SUPERVISED CLUSTERING WITH STRUCTURAL SVMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Thomas W. Finley

January 2009

SUPERVISED CLUSTERING WITH STRUCTURAL SVMS

Thomas W. Finley, Ph.D.

Cornell University 2009

Supervised clustering is the problem of training clustering methods to produce desirable clusterings. Given sets of items and complete clusterings over these sets, a supervised clustering algorithm learns how to cluster future sets of items in a similar fashion, typically by changing the underlying similarity measure between item pairs. This work presents a general approach for training clustering methods such as correlation clustering and k-means/spectral clustering able to optimize to task-specific performance criteria using structural SVMs. We empirically and theoretically analyze our supervised clustering approach on a variety of datasets and clustering methods. This analysis also leads to general insights about structural SVMs beyond supervised clustering. Specifically, since clustering is a NP-hard task and the corresponding training problem likewise must make use of approximate inference during training of the parameters, we present a detailed theoretical and empirical analysis of the general use of approximations in structural SVM training.

BIOGRAPHICAL SKETCH

Thomas Finley graduated from Duke University (Durham, NC) in 2002 with Bachelor's of Science degrees in both Computer Science and Mathematics, graduating with High Distinction in both, with thesis work overseen by Dr. Susan Rodger and Dr. William Allard in both fields, respectively. He also minored in Economics. In the fall of 2003, Thomas began graduate studies at Cornell University (Ithaca, NY), with a major field in Computer Science and a minor field of Information Science. Dr. Thorsten Joachims advised his doctoral research in Computer Science in the general field of machine learning and the more specific field of discriminative structural learning. Thomas is completing requirements for Doctor of Philosophy in the fall of 2008. In the fall of 2008, he will begin a position as Research Software Development Engineer for Microsoft (Redmond, WA).

To Meg, Mom, and Dad!

ACKNOWLEDGEMENTS

The research presented in this dissertation is supported by the Nation Science Foundation under Award IIS-0412894 and IIS-0713483, gifts from Yahoo! Inc., and the KD-D grant.

I'd like to extend thanks and acknowledgements to my thesis advisor, Dr. Thorsten Joachims, for direction, collaboration, and encouragement during the past five years, and further to my collaborators and advisormates Filip Radlinski, Yisong Yue, and Chun-Nam Yu, with whom I have collaborated with on works not appearing in this thesis. I'd also like to extend my thanks to my officemates and near-officemates over the years, and the members of the Machine Learning Discussion Group for many fun conversations. On a far more general level, I'd like to thank Cornell, Ithaca, and central New York State for being a great home these past five years. Finally, I am quite grateful for the more personal encouragement I receive from my wife Megan, and my parents, John and Sue. Thanks for all the love and proofreading!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION TO SUPERVISED CLUSTERING

Clustering is an important data mining task employed in dataset exploration and other settings where one wishes to partition a set of items into clusters of related items. However, there may be many possible ways of breaking up a set of items since there are many different ways one can define "related items" for a given task. For example, if one were clustering news articles, there are many possible ways one could group the articles. One could group them as being relevant to the same location, or being written by the same author, or being about the same news story. However, in many applications of clustering, one may be interested in one particular way of splitting up a set of items; if one wishes to group news articles by their story, then a clustering function that favors grouping news articles by authorship would be less useful than a clustering function that favors splitting news articles apart by those that refer to the same news story. There are many applications where one wishes to partition input item sets in a particular fashion. For instance:

- Noun-phrase coreference, where, given the noun phrases in a text, a clustering algorithm predicts sets of noun-phrases that co-refer, i.e., noun-phrases that refer to the same entity [76, 78, 100]. For example, consider the text "the $\boxed{\text{dog}_a}$ put the $\boxed{\text{ball}_b}$ in $\boxed{\text{his}_c \text{ mouth}_d}$." Items a and c would be grouped together since they both refer to the dog. However, item b would be in its own group, as would d, as there are no other noun phrases in this text referring to the ball, or the dog's mouth.
- Image segmentation or perceptual grouping, where, given an image (considered a collection of pixels), a procedure finds groups of pixels corresponding

to regions in the image where one object or another is present [47, 73, 86, 98].

- News story clustering, where, given a set of news articles for a day, cluster those that cover the same story [40]. Automated news aggregators like Google News use clustering procedures to group news articles into stories.
- Speech segmentation, where, given a sound file from a single microphone recording of multiple speakers, the voice of each individual speaker is recovered [6].

In situations where a particular type of grouping of a set of items is desired, it may be necessary to adjust a clustering algorithm so that the clustering algorithm outputs the desired partitioning. Further, with the understanding that it may not be possible to get an algorithm always to return with exactly the desired clustering, we would like the clusterings returned by the clustering algorithm to be as close to a desirable clustering as possible.

The natural question is, how can one adjust a clustering algorithm so that it produces desirable clusterings? If we suppose that items are described by multiple attributes, finding the right combination or even weighting of attributes that lead to a desirable clustering may be difficult and time consuming to do manually, even when there are only a dozen attributes. In situations where one could exploit thousands of item attributes to produce a clustering, the infeasibility of doing manual adjustment, and the attraction of automatic adjustment of a clustering function, becomes obvious. In many situations it may be easier or ultimately more effective to provide a procedure with examples of good clusterings of data, and let this procedure automatically infer a parameterization of a clustering function so as to change what clusterings it will produce for given input item sets.

Also, in practice one cannot expect to learn a parameterization such that a clus-

tering algorithm will return the "right clustering" every time under every possible input. Furthermore, different tasks may have widely differing measures of what is considered a "close enough" answer. Therefore, it follows that a parameterization must be chosen taking into account some penalty or loss, so the risk incurred by using a given parameterization of a clustering algorithm is low relative to other possible parameterizations.

This thesis describes tasks and methods for supervised clustering. A supervised clustering method is a means of automatic adjustment of a clustering function. The clustering algorithm is adapted by a supervised machine learning procedure, which makes use of a training set of example input sets and output partitionings of those sets. This chapter discusses the problem of supervised clustering, explains what it is, clarifies what it is not, and relates it to other closely related fields of machine learning. Chapter 2 presents the structural SVM learning framework, which we use in Chapter 3 and Chapter 4 to derive a supervised correlation clustering algorithm and supervised k-means/spectral clustering algorithms, respectively. In contrast to existing approaches to supervised clustering and other closely related methods, the principles of the proposed framework (a) are very general and can parameterize a variety of clustering procedures, (b) optimize clustering performance directly rather than relying on heuristics and assumptions about the distribution of the data, (c) optimize parameterizations to problem specific loss functions, and (d) are demonstrably efficient, both theoretically and empirically. We derive these methods, theoretically characterize them, and empirically analyze them on a variety of supervised clustering datasets covering a range of applications. Since these learning frameworks require the use of approximations in training, we theoretically and empirically analyze the use of approximations in structural SVMs in Chapter 5. We also present various software tools written to support this work in the

appendices.

1.1 Different Types of Clustering

Let us first start our discussion by defining what we consider to be a clustering algorithm: a clustering algorithm is a procedure that outputs a partitioning of a given input set. In the larger sense, though, cluster analysis is an extremely broad term referring to many variants on this general theme. While all concern taking sets of items and finding groupings of these items, many go beyond simply producing a partitioning.

One of the most popular clustering algorithms that does not produce a simple partition of items is hierarchical clustering. Hierarchical clustering schemes produce a recursive partitioning of the items called a dendrogram. In a dendrogram, the root partition contains all data, but is itself partitioned into subgroups, with each subgroup partitioned recursively into subgroups, until a cluster has one item and is therefore indivisible [55]. The well known single-link, complete-link, and average-link agglomerative clustering methods fall into this category.

Soft clustering assigns items to clusters with a certain confidence (where this confidence is often but not necessarily stated as a probability), so for a single item, its score of belonging to any given cluster may be nonzero for multiple clusters [9, 37]. Closely related to soft clustering is the field of clustering with mixture models, where the input set $\mathbf{x}$ is assumed to be generated from a type of distribution. This generating distribution is characterized by a series of mixture models, where each component of the mixture corresponds to a different cluster [34, 42, 68]; i.e., all elements from one cluster come from one component of a mixture, where

all elements from another cluster come from a different component of the mixture. The task of the clustering algorithm in such a setting may be to find the parameterizations of the components of the mixture model that lead to the generation of the data $\mathbf{x}$. For example, if the generating distribution of the input set $\mathbf{x}$ were a mixture of Gaussian distributions, then the mixture model clusterer's task is to find the mean of the mixture distributions (typically algorithms assume fixed variance [34]). While inferring the components of the mixture model does not give us a partition, the probability of each point being generated by each mixture is known, so assigning points to the group corresponding to their most likely generating component is trivial. Furthermore, the probabilities of each item belonging to each component's group are often used as a soft clustering scheme.

While in principle supervised learning procedures could be used to parameterize any such method, and future supervised clustering work could very well expand its view of clustering to include these other definitions, this thesis concerns learning parameterizations for clustering procedures that produce simple partitions of input sets. The clusters are therefore flat (i.e., non-hierarchical insofar as the clusters have no identified sub-clusters) and hard (i.e., an item's cluster membership is unambiguous). Correspondingly, this work often treats the terms clustering and partitioning interchangeably, and cluster and partition interchangeably.

For clarity, let us phrase this more precisely. In this setting our inputs are $\mathbf{x}$, where $\mathbf{x}$ is a set of m items

$$\mathbf{x} = \{x_1, x_2, \ldots, x_m\}. \tag{1.1}$$

The m is not fixed across sets $\mathbf{x}$, that is, two different sets $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$ may have different numbers of items so $|\mathbf{x}| \neq |\mathbf{x}'|$. We suppose the size m we are talking about will be obvious in context, or simply use $|\mathbf{x}|$. Furthermore, suppose all pairs

of items $x_i, x_j \in \mathbf{x}$ have some pairwise similarity $\phi_{ij} \in \mathbb{R}$. A clustering algorithm produces a clustering or partitioning

$$\mathbf{y} = \{y_1, y_2, \ldots, y_k\} \tag{1.2}$$

of k clusters or partitions (this k may be fixed a priori or determined dynamically, depending on the clustering algorithm, and as with m for $\mathbf{x}$, may vary across different $\mathbf{y}, \mathbf{y}' \in \mathcal{Y}$). This $\mathbf{y}$ is a partition of $\mathbf{x}$, such that

$$\cup_{y \in \mathbf{y}} y = \mathbf{x} \tag{1.3}$$

and

$$\sum_{y \in \mathbf{y}} |y| = |\mathbf{x}|. \tag{1.4}$$

We view this clustering procedure as a function

$$h : \mathcal{X} \to \mathcal{Y}, \tag{1.5}$$

where $\mathcal{X}$ is the set of all possible item sets, and $\mathcal{Y}$ the set of all possible partitionings of the item sets of $\mathcal{X}$. Put somewhat inexactly, the goal of the clustering procedure is to produce a clustering $\mathbf{y}$ of an item set $\mathbf{x}$ so that some function of item pair similarity ϕ_{ij} for pairs of items $x_i, x_j \in y_\ell$ for $y_\ell \in \mathbf{y}$ is maximized. The precise function that the clustering procedure attempts to maximize will differ from algorithm to algorithm.

1.2 Supervised Clustering

Now that we have defined clustering more precisely, let us clarify our definition of supervised clustering. Supervised clustering describes the parameterization of a clustering algorithm so that the clustering algorithm will tend to produce desirable

clusterings. The user informs the clustering algorithm of what is desirable by providing training examples, where each example consists of both a set of items and a completely specified partition of that set.

In a supervised clustering setting, we take the above view of the clustering function

$$h_{\mathbf{w}} : \mathcal{X} \rightarrow \mathcal{Y}, \tag{1.6}$$

where $\mathcal{X}$ is the set of all possible item sets, and $\mathcal{Y}$ is the set of all possible partitionings of the item sets of $\mathcal{X}$. Note the use of the $\mathbf{w}$ subscript for h in (1.6) versus (1.5). This indicates that the clustering procedure is parameterized through some model parameterization $\mathbf{w}$. By changing $\mathbf{w}$, we may change the clustering $\mathbf{y} = h_{\mathbf{w}}(\mathbf{x})$ this function $h_{\mathbf{w}}$ will produce for some input item set $\mathbf{x}$.

This parameterization $\mathbf{w}$ is set through a learning procedure that makes use of a training set

$$\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\} \in (\mathcal{X} \times \mathcal{Y})^n, \tag{1.7}$$

that we assume is drawn i.i.d. from some unknown distribution $P(\mathbf{x}, \mathbf{y})$. This training set $\mathcal{S}$ consists of n examples of input item sets $\mathbf{x}_i \in \mathcal{X}$ and partitionings of those sets $\mathbf{y}_i \in \mathcal{Y}$. It is important to note that these item sets $\mathbf{y}_i$ are complete partitionings of their input sets $\mathbf{x}_i$; consequently, supervised clustering is concerned with applications where one could provide complete example partitionings, such as those listed earlier (e.g., noun-phrase coreference, image segmentation, news story clustering, speech segmentation).

A supervised clustering method will have some notion of a loss function

$$\Delta : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}, \tag{1.8}$$

a function measuring to what extent two clusterings differ. A goal common to nearly all machine learning methods, including supervised clustering, is to minimize risk,

$$\mathcal{R}_P(h_{\mathbf{w}}) = \int_{\mathcal{X}\times\mathcal{Y}} \Delta(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x}))\ dP(\mathbf{x}, \mathbf{y}) \tag{1.9}$$

which is, intuitively, the expected loss incurred by a hypothetical future random example $(\mathbf{x}, \mathbf{y})$ drawn from the unknown distribution $P(\mathbf{x}, \mathbf{y})$ when the input $\mathbf{x}$ is run through the parameterized clustering algorithm $h_{\mathbf{w}}$. In reality, the goal of directly minimizing risk is impossible since $P(\mathbf{x}, \mathbf{y})$ is unknown, so different machine learning methods minimize different criteria, as explained more fully in Chapter 2.

The exact form of the clustering function $h_{\mathbf{w}}$, parameterization $\mathbf{w}$, loss function Δ, and whatever other miscellaneous criteria the supervised clustering algorithm takes into account when choosing $\mathbf{w}$ depend upon the supervised clustering algorithm.

How precisely should $\mathbf{w}$ parameterize a clustering algorithm? Many types of clustering algorithms can be phrased as an attempt to find a clustering $\mathbf{y}$ that maximizes some joint function $f_{\mathbf{w}} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ over similarities between pairs of items of an input set $\mathbf{x}$. More formally, for item set $\mathbf{x}$ and any pair of items $x_i, x_j \in \mathbf{x}$ in the item set, we suppose a similarity measure $\phi_{x_i,x_j} \in \mathbb{R}$ is defined, and the clustering procedure $h_{\mathbf{w}}(\mathbf{x})$ finds a clustering $\mathbf{y}$ to maximize some function $f_w(\mathbf{x}, \mathbf{y})$ of these ϕ_{x_i,x_j} pairwise similarities. Typically, pairs of items with high similarity scores would tend to be placed in the same cluster, while pairs with low similarity scores would tend to be placed in a different cluster. Correspondingly, $\mathbf{w}$ parameterizes a similarity measure between item pairs. By changing the similarity measure, we can change what clustering maximizes the objective function $f_{\mathbf{w}}$

The use of the phrase similarity *measure* is important: we use this to indicate that this function is not necessarily a kernel, nor is any distance measure induced by this similarity measure necessarily a metric. However, both metrics and kernels could be considered similarity measures, and methods of supervised clustering that are restricted to learning metrics and kernels are perfectly valid.

1.3 Supervised Clustering as Pairwise Classification

If the goal of supervised clustering is to form items into groups, and clustering algorithms form groups according to which items are most similar, logically it seems plausible that one may learn a classifier over pairs of points to learn an "in the same group" versus "not in the same group" classification. During classification, there could be inconsistencies in such a scheme (i.e., for three points A, B, C, the classifier could think $A - B$ and $A - C$ are grouped, but $B - C$ are not), but one could subsequently run some clustering algorithm over these outputs from the classifier to enforce a consistent partitioning.

More particularly, one may take all pairs of items in all training sets, describe each pair in terms of a feature vector, and let positive examples be those pairs in the same cluster and negative examples be those pairs in different clusters. Then, one trains a classifier on this training set. When one wants to cluster a new set of items $\mathbf{x}$, they could run all pairwise similarity vectors ϕ_{ij} for items $x_i, x_j \in \mathbf{x}$ through this learned classifier. The output values from the classifier are the pairwise similarity values; positive and negative outputs indicate a pair should or should not be in the same cluster, respectively. Then, one could cluster based on this matrix of output similarities to find the final clustering $\mathbf{y}$.

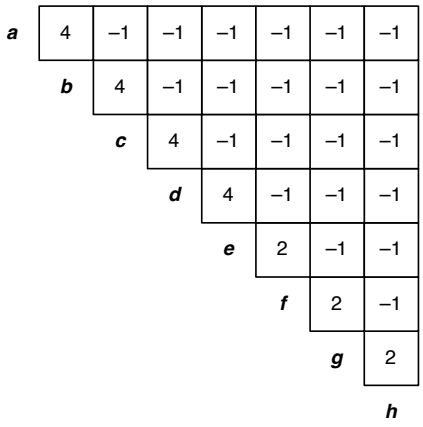

Figure 1.1: An example of a similarity matrix between eight items (denoted here a through h). For example, the item pairs (a, b) and (a, c) have similarity 4 and -1, respectively.

One might view this as a supervised clustering algorithm, though of a different and less direct type than the subjects in this work and those summarized in Section 1.4. If one could learn the pairwise decision function perfectly, the resulting judgments would correspondingly partition object sets perfectly. However, in the likely event that such a perfect scheme will not be learned, there are significant shortcomings to this approach.

First, the pairwise classifier, which is usually optimized for accuracy of judgments on the training set, is optimizing the wrong thing. Optimizing for pairwise performance accuracy is not ideal for supervised clustering. As an example, suppose that we have eight items $a, b, \ldots, h$ that form a training example with a true partitioning into two clusters, (a, b, c, d, e), and (f, g, h). Suppose, further, that our clustering scheme finds the partitioning such that the sum of similarities between all item pairs in the same partition over all partitions is maximized. (This objective happens to be identical to correlation clustering [33], treated in more depth in Chapter 3.)

Now suppose that we have some parameterization $\mathbf{w}$ that leads to the pairwise similarity scores shown in the similarity matrix of Figure 1.1. How can we evaluate the quality of this parameterization? From the point of view of our hypothetical pairwise classification learner, this parameterization would seem poor: only 18 of 28 possible pairwise relationships have the correct pairwise classification, so for a pairwise learner this represents 35.7% training error. From the point of view of a supervised clustering learner, however, this scheme is perfect: the optimal partition under the clustering scheme and the training partition are identical, so this would have zero training error.

Second, in clustering applications, often the number of pairs in a cluster is relatively small, e.g., only 3.6% of pairs in the MUC-6 test set represent items in the same cluster [77]. The training imbalance could lead to an understatement of pairwise similarity if we are optimizing for accuracy.

Third, some supervised clustering tasks are associated with a performance measure, e.g., the model-theoretic MITRE score for MUC noun-phrase coreference [108] described in Section 3.3.2. A pairwise classifier that optimizes for accuracy may yield inferior performance, since the tradeoffs it makes may not accurately reflect tradeoffs appropriate for clustering performance. For example, in MUC-6, since only 3.6% of pairs are coreferent in the test set, a model that simply identifies all pairs as being non-coreferent would have a 96.4% accuracy across all documents, which looks attractive to a pairwise learner trained for accuracy. However, for any practical purpose that rule would be totally useless, and would actually have loss of 100, i.e., the worst one could possibly do, under the MITRE loss.

Fourth, and most important, a pairwise classifier that assumes pairs are i.i.d. cannot take advantage of dependencies between item pairs. Consider this small

"A Balrog," muttered Gandalf$_a$. "Now I understand." He faltered and leaned heavily on his staff. "What an evil fortune! And I am already weary."

⋮

"Mithrandir we called him in elf-fashion," said Faramir, "and he was content. Many are my names in many countries, he said. Mithrandir$_b$ among the Elves, Tharkûn to the Dwarves; Olórin I was in my youth in the West that is forgotten, in the South Incánus, in the North Gandalf$_c$; to the East I go not."

⋮

"Mithrandir$_d$!" he cried. "Mithrandir!" "Well met, I say to you again, Legolas!" said the old man.

Figure 1.2: An example of three passages from *The Lord of the Rings*, with four noun-phrases highlighted. (Note that there are, in reality, very many more than four noun-phrases in these examples.)

document: "Bush$_a$ ate some tacos. ... President Bush$_b$ likes tacos. ... He$_c$ said tacos are good food for himself$_d$, the president$_e$." We want a, b, c, d, e to be in the same cluster. With pairwise features as in [100] or [78], it is probably easy to learn that (a, b), (b, c), (c, d), and (d, e) are coreferent and should have positive similarity, but difficult to learn that pairs (a, c), (a, e), and (c, e) are coreferent. However, a clustering algorithm would still come up with the correct clustering even if the hard pairs were kept as unknowns (perhaps with similarity kept slightly less than 0) and only the easy pairs were learned. A learner could exploit these transitive dependencies to learn more effectively, since we do not actually need to learn all the difficult relationships. Insisting that we do may diminish the overall effectiveness of the hypothesis.

To illustrate this point, consider the example of Figure 1.2. In this example, we have highlighted four noun-phrases. Learning a model that a and c and b and d

are coreferent is trivial assuming there are features regarding string matches, and b and c quite possible with pairwise features that capture sentence level syntactic relations or other proper choices of linguistic features [100], but it is highly unlikely that one could reasonably directly link a to b or d, nor c to d. However, a learning algorithm that incorporates clustering could take advantage of the fact that getting every single pairwise relationship right is unnecessary (as seen in the example of Figure 1.1).

Another more practical consideration of using a pairwise trainer for classification is that the sheer number of training examples entailed by such a scheme may be unwieldy, though recent years have seen advances in learning with large training set sizes [52, 97].

Nonetheless, with the aid of heuristics and domain specific knowledge about the problem, pairwise classification has been employed to learn distance functions. Some methods employ heuristics to train the classifier only on selected item pairs. The hope is that a properly chosen heuristic will compensate for some of these weaknesses.

1.3.1 Canopies Heuristic

In terms of computation time, work in [21] adapted the canopy technique, originally intended for unsupervised clustering [75], to methods for selecting training pairs for supervised clustering. Given an a priori similarity measure, that is, one known prior to any training at all taking place, the canopy method will select only those item pairs whose a priori similarity is within a certain interval. The pairwise classifier is then trained on the selected pairs. This a priori measure is not the

similarity we are hoping to learn, but something that is "close enough" to the truth to be useful to select training sample pairs. The idea is that the training set will be composed of those pairs that are close enough to the point where learning a distance metric over them is considered useful, but far enough away to the point where they are an "interesting" training example. So, for a set $\mathbf{x}$ of size $m = |\mathbf{x}|$, instead of having a training sample composed of all $\binom{m}{2}$ possible pairs, one may have a training sample composed of roughly $\frac{\alpha m}{2}$ possible pairs, where the constant α is the average number of points in the interesting region across all points.

1.3.2 Noun-Phrase Chain Heuristic

An heuristic for leveraging pairwise training specific to noun-phrase coreference appears in [78]. In short, each noun-phrase x_b and its closest preceding non-anaphoric coreferent noun-phrase x_a form a positive training pair, while all non-coreferent noun-phrases in between x_a and x_b are paired with x_b as a negative training example. This approach yields excellent performance for the NP coreference task, but was built with expert domain knowledge and is not applicable to other tasks.

In the noun phrase coreference problem, there is a document with a series of noun phrases. Each noun-phrase refers to a particular object. The idea is to partition the noun phrases so that any two partitions refer to different entities, and all noun phrases within a partition refer to the same entity. Of course, a way to partition a set is by clustering over that set. Soon and Ng [100] describe an approach to use supervised clustering for noun-phrase coreference. This approach is refined and extended in [78]. Both papers have algorithms built to accept the type of input that is provided in the MUC-6 and MUC-7 coreference tasks.

In both approaches, they do not view the set of noun phrases as a set exactly, but rather as an ordered linearized list, so that each document has its noun phrases enumerated as an ordered list $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, not as an unordered set. In both papers, each noun phrase x_i is considered by a series of attributes including gender, whether it is a pronoun, whether it appeared after the word "the," and many other features. A pair of noun phrases x_i, x_j has an associated vector ϕ_{ij} containing features describing how compatible they are. The idea is to train a classifier over these ϕ_{ij} vectors to see if a pair (x_i, x_j) is coreferent.

Soon and Ng [100] do not merely take all pairs of NPs and train on them. First of all, this is not suitable to the domain, as partially illustrated in Figure 1.2. Attempts to use clustering to solve noun-phrase coreference has mainly centered around using something resembling single link clustering. The clustering algorithm they use is specially adapted to the task, given that this is an ordered collection of NPs. For each x_j, the clusterer works its way backwards, checking x_{j-1}, x_{j-2}, and so forth until it discovers an x_i (where $i < j$) that the learned classifier considers as coreferent. The two are then linked in the output cluster. At this point the clusterer, moves on to x_{j+1}, and performs the same backwards search. Once all the noun phrases x_j have been checked in this fashion, whatever two noun phrases are connected through a chain of links are considered coreferent.

The method they use for choosing the similar and dissimilar sets $\mathcal{S}$ and $\mathcal{D}$ is made to reflect how this clustering is performed. For each noun phrase x_j, its closest previous non-pronoun coreferent noun phrases x_i is found. (If there is no such x_i, the selector moves on to noun phrase x_{j+1}.) When x_i is found, the pair (x_i, x_j) is added to $\mathcal{S}$, and for all intervening $x' \in \{x_{i+1}, \cdots, x_{j-1}\}$ that are not coreferent with x_j, the pair (x', x_j) is added to $\mathcal{D}$.

A weakness of this approach is that some important training examples are overlooked. Consider noun phrases with no coreferent noun-phrases, and noun phrases that are the first mention of an entity. Neither appear as the second element of a pair in either $\mathcal{S}$ or $\mathcal{D}$. This could be problematic because both of these classes of noun phrase are likely to be constructed in different ways than noun phrases that refer to something that has already been referenced. When it comes time to do the backward search, the classifier could get confused if it is searching backwards from a unique noun-phrase or a first mention, as this is a case the model was never prepared for in training.

1.4 Previous Supervised Clustering Methods

As seen in our list of applications of supervised clustering, the field of supervised clustering exists in many forms under many names depending upon the application of interest. However, some work treats the problem of supervised clustering and learning partitioning functions as a general problem of interest, rather than being application-driven work specific to image segmentation, or noun-phrase coreference, or speech segmentation, and so on. This includes work by this thesis' author—specifically, the work on supervised clustering for correlation clustering based methods [40] and k-means and spectral type methods [41], which are more fully described in Chapter 3 and Chapter 4, respectively. While we save a full discussion for the appropriate chapter, these methods both use a structural SVM based approach to learn a parameterized similarity function, directly optimizing to a loss function Δ of interest to the particular application. In work by Haider et al., the correlation clustering method has proven effective as an online algorithm in the clustering of incoming e-mails [45].

In addition, McCallum and Welner solve the noun-phrase coreference problem with a clustering method phrased as a graphical model [76]. In an input set of noun-phrases $\mathbf{x}$, the partitioning $\mathbf{y}$ of $\mathbf{x}$ is stated as a collection of $\binom{|\mathbf{x}|}{2}$ variables y_{ij}, where $y_{ij} = 0$ or 1 depending upon whether $x_i, x_j \in \mathbf{x}$ are non-coreferent or coreferent, respectively. The likelihood function of a partitioning $\mathbf{y}$ given the input set $\mathbf{x}$ is

$$P(\mathbf{y}|\mathbf{x}) = \frac{1}{Z_{\mathbf{x}}} \exp\left(\sum_{i,j,\ell} \lambda_\ell f_\ell(x_i, x_j, y_{ij}) + \sum_{i,j,k,\ell'} \lambda_{\ell'} f_{\ell'}(y_{ij}, y_{jk}, y_{ik})\right). \tag{1.10}$$

The first term is effectively the ℓ-th feature function for the item pair x_i, x_j, where the λ_ℓ parameter is the weight for that given feature. The second term is a notational trick for stating consistency of a partition probabilistically (e.g., the probability of $y_{jk} = 0$ should be very low given $y_{ij} = 1$ and $y_{ik} = 1$); in actual implementation, the inference algorithm simply ignores inconsistent partitions. It is easy to see that the most likely partitioning $P(\mathbf{y}|\mathbf{x})$ is that which maximizes the sum of the pairwise similarities among all item pairs, which is the objective function of correlation clustering. This work makes it very close to the correlation clustering work defined by Chapter 3, except that the parameters λ are learned through maximum likelihood instead of minimization of empirical risk. However, actually computing the maximum likelihood parameterization is intractable since the gradient over parameters involves computing a sum over all possible clusterings. Consequently, they elect instead to use a structural perceptron [23].

Bach and Jordan [5, 6] present a means of learning parameterization of spectral clustering. It is difficult to summarize this work without understanding spectral clustering, and a treatment of spectral clustering will not appear until Chapter 4. However, suffice to say, for a given item set $\mathbf{x}$, if we phrase the similarity matrix $A \in \mathbb{R}^{|\mathbf{x}|\times|\mathbf{x}|}$ where $A_{ij} = \phi_{x_i,x_j}$ for $x_i, x_j \in \mathbf{x}$ where $\phi_{x_i,x_j} \in \mathbb{R}$ is the real valued item pair

similarity, spectral clustering methods perform an eigenanalysis on A to determine a partition of $\mathbf{x}$. The learning scheme in [5, 6] searches for a parameterization $\mathbf{w}$ leading to a similarity matrix A as close to an "ideal" similarity matrix as possible. The ideal similarity matrix is one which would lead to the correct partitioning. In searching for the parameter $\mathbf{w}$, they minimize a function that they prove upper bounds the loss Δ (for a special form of Δ) that would be incurred were one to run spectral clustering on the parameterized matrix A. In other words, they find $\mathbf{w}$ to directly minimize an upper bound on the empirical risk for a certain Δ. A limitation of the method as provided is that it is applicable to only one Δ loss function. A second limitation is the restriction of finding $\mathbf{w}$ through spectral clustering. While spectral clustering and k-means are closely related insofar as they are two different algorithms to solve the same problem [6, 36], an algorithm that directly optimizes for k-means could be more useful in situations where the upper bound implied by this spectral relaxation is too loose.

Daumé and Marcu [30] provide a generative view of clustering based on Dirichlet processes. The set of items and its clustering are assumed to come from a Dirichlet process. The advantage that they have in using a Dirichlet process is that it can be used to define an infinite mixture model, which is an advantage in the case of supervised clustering since in many applications there may be an unknown and potentially unlimited number of clusters, which a Dirichlet process can model gracefully. Their primary motivating application is paper citation matching, where a set of items is a set of citations, and a cluster would correspond to multiple citations of the same paper. Informally, when encountering a new citation, a Dirichlet based clusterer would either assign the citation to an existing cluster (i.e., this is a paper whose citations have been seen before), or assign the citation to a new cluster (i.e., this is a paper whose citations we have not seen before). In

such a setting, the exact number of papers would hardly be known a priori. The proposed supervised clustering algorithm sets parameters on the Dirichlet in order to control how willing the clusterer is to assign a given item to some other existing cluster, versus to its own cluster. The weakness of this approach is its generative nature; they suppose that the process generating the data is known. While they offer arguments as to why Dirichlet distributions may model some of the data they are interested in, modeling assumptions remain a weakness in this sort of model, especially if one were interested in including many possibly dependent features in the learned similarity measure. Also, while the clustering procedure resulting from this model is evaluated on a number of different clustering loss functions Δ, the training algorithm is not able to choose a parameterization to minimize the training loss for these evaluation losses. The learning procedure also suffers from being computationally inefficient.

Kamishima et al. [56] derive another generative scheme for supervised clustering. The paper characterizes clustering as finding some partition that maximizes some probability, which depends upon features of the items, item pairs, and partitions. The supervised clustering learning procedure correspondingly learns parameters of these probabilistic functions to make the correct clusterings encountered in the training data most likely. As with the previous generative work, to gain tractability, this relies on independence assumptions about the points, and the features. While the paper briefly argues that one can choose features that are "as independent as possible" to overcome this weakness, in practice this ensuring independence of features is not easy to do. Furthermore, the clustering function and method of training, while interesting, are somewhat peculiar to the paper, and it is not at all obvious that application to a more familiar proven clustering scheme would be feasible under this framework.

1.5 Supervised Clustering Is Not Multiclass Classification

Repeatly, there has been confusion on the distinction between supervised clustering and simple multiclass classification. Though we were surprised the first few times this confusion arose, it is understandable: clustering and classification appear quite similar. In both cases, when they are predicting the outputs for a set, they produce a partition of an input set.

Indeed, in many types of machine learning algorithms, the line between clustering and classification is very thin. Utilizing clustering in classification is the basis for the majority of the semi-supervised and transductive classification research. In these settings, the target hypothesis is either a multiclass or binary classification rule, and the training data consists both of labeled and unlabeled points. Informally, the goal of these algorithms is that the learned classification hypothesis should be consistent with the labeled data, but the learned hypothesis should also obey the cluster assumption. If we interpret examples as existing in a vector space, then the decision boundary for the learned hypothesis should pass through regions of the space that have low density (or, depending on the type of hypothesis that is being learned, that class centers should be in regions of high density) considering both labeled and unlabeled points [19, 18, 88, 96]. In intuitive motivation, mathematical formulation, and algorithmic interpretation, these methods use clustering-like techniques to guide the learning and application of semi-supervised classification schemes [43].

It is worth noting that this technique is applicable to situations other than multiclass and binary classification. Similar techniques have also been applied to cases where the learned hypothesis function produces outputs that are more complex and structured [2, 14, 20, 49]. However, in these settings, the intuitive notion

of the cluster assumption of points lying in space becomes less compelling since there is no longer an easily visualizable "vector space" model for these complex objects, so these efforts often adopt different terminology.

Despite intersecting in some applications, there are important differences in the types of problems the two are appropriate for and what concepts they are capable of representing, which we discuss in detail here.

1.5.1 Dynamic Clusters versus Static Classes

One major difference between classification versus clustering is supervised clustering schemes learn to partition sets of items, whereas multiclass classification schemes learn to partition sets of items into static, defined partitions.

A simple example might help illustrate this difference: Consider the problem of clustering marbles, and suppose that we have as training data a set consisting of red marbles, green marbles, and blue marbles, as pictured in Figure 1.3(a). The training data consists of the partition of these marbles into those three colors. The features for each marble include the "color angle" of this marble's hue on a color wheel (red is at 0°, green is at 120°, blue is at 240°), as well as various other features such as size, weight, clarity, and other features that turn out to be irrelevant to this task.

Were we to consider this a multiclass classification problem, then such an algorithm would learn how to classify future items fed into the algorithm as being in a red, green, or blue class, which was learned during the training phase. For example, for a given input marble, the classifier would view red, green, or blue as the most likely classification depending on how close the input marble's hue is

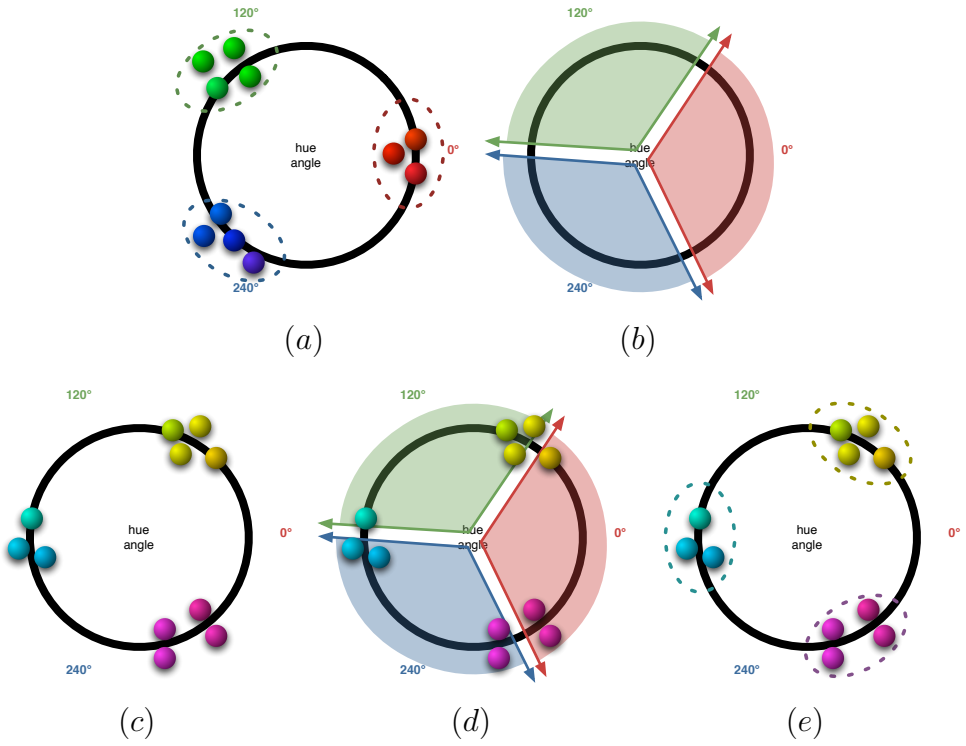

Figure 1.3: This illustration serves as an example of the difference between clustering marbles, and classifying marbles.

to 0°, 120°, or 240°, respectively. Were we to consider this as a supervised classification problem, then the learned model would learn how to partition a set of items, perhaps learning that difference in hue angle and likelihood of being in the same cluster are inversely related. The classification learner wants to learn how to partition future items into the groups indicated by the training data, e.g., learn what areas in the space will correspond to membership in what class, as indicated in Figure 1.3(b).

Suppose that we train either a supervised clustering or multiclass classification algorithm on the red-green-blue marbles, but in prediction (either for supervised clustering or multiclass classification) we are fed marbles that are roughly in groups of yellow, cyan, and magenta marbles, as shown in Figure 1.3(c). The multiclass

classification, having learned regions corresponding to the red, green, and blue marbles, will have a tendency to "split" the natural yellow, cyan, and magenta groups since the classification regions have a decision boundary through each of these groups, a Figure 1.3(d). Alternately, however, the supervised clustering method, having learned to partition items according to color, but *not* to put them into any predefined bins, can identify the natural color regions as shown in Figure 1.3(e).

This is not to say that the partitioning according to Figure 1.3(d) is wrong; in many applications this is precisely what one wants. It is also not to say that the application of any given supervised clustering algorithm scheme would result in the scenario leading up to Figure 1.3(e). This merely illustrates a basic difference between the two types of tasks for which these methods are appropriate.

1.5.2 Large Numbers of Unknown Groups

Another difference between clustering and classification becomes obvious when one considers that classification assumes one knows a priori what classes a set of objects could be partitioned into, which is not always the case.

For instance, consider a task like noun-phrase coreference. Recall that in noun-phrase coreference one takes all of the noun phrases in a document, and partitions them according to what noun-phrases refer to the same entity. If we were to apply multiclass classification to this scheme, we would have to have a separate class for each entity. This would be impossible because the sheer number of classes implied by having to produce a class description of every entity that has been encountered and ever could be encountered is prohibitive, and these entities are also unknown. Also consider the problem of clustering news articles as being about the same news

story or not: it would be impossible to anticipate what "classes" corresponding to stories that the news articles are going to fall into in future days, or else it would not be news.

Furthermore, in common noun-phrase tasks, few of the entities referred to in the evaluation set actually occurred in the training set [77], and it is unclear how one can generalize knowledge about how to "group" items in the context of pure classification of individual noun-phrases. If one's training data talks about Anne, Bob, and Clarence, what is the algorithm to think when it encounters, in application, noun-phrases referring to a previously unknown entity Doug?

1.5.3 Different Appropriate Choices of Features

In addition to these fundamental differences in purpose, there are practical differences that separate what types of parameterizations are acceptable for supervised classification versus supervised clustering. At issue is that classification learns over individual features, whereas supervised clustering, in parameterizing a pairwise similarity score, has features describing two points jointly, that is, pairwise features. To take an example, in a vector space model, consider two points $x_i, x_j \in \mathbf{x}$ that are also real valued vectors $x_i, x_j \in \mathbb{R}^N$. As a classification task, the natural instinct would be to take the vectors as is. In contrast, a pairwise feature vector as used in clustering would involve some synthesis of the two, perhaps their difference $|x_i - x_j|$ or a componentwise product $x_i \circ x_j$. As an example, if we were trying to learn a concept like the marble clustering example of Section 1.5.1, a classifier woudl find the hue angle as a useful feature, but a clusterer would get more use from a pairwise feature of the difference in hue angle.

A more subtle difference between the two feature representations becomes clear when one considers what types of features are helpful versus harmful in both settings. Suppose a hypothetical individual is working with noun-phrases, and a training set of documents that talk about the entities Anne, Bob, and Clarence. If this individual wishes to use a multiclassification scheme, i.e., classify new noun-phrases as referring to either Anne, Bob, or Clarence, then many binary features indicating that the noun-phrase in question is the character sequence `Anne`, or `Bob`, or `Clarence` become extraordinarily helpful. In the case of supervised clustering, though, features that are this specific can become harmful, since the goal is to be able to group noun-phrases no matter what entity they refer to, whether they refer to these three individuals or someone completely different; a model that depends heavily on these simple features would be unable to transfer to a new, unseen entity. Features specific to a token that appears in text would be difficult to help learn a general model [15]. They would allow easily fitting the training data while being nearly useless for data on unseen entities. This is not to suggest such features could not be useful—as a practical matter it seems humans must do something like this to connect names and titles to entities—but such features with very limited training data would be harmful for generalization performance.

1.6 Relation to Semi-Supervised Clustering

As one might guess from the name, semi-supervised clustering is related to supervised clustering, but is typically applied to very different settings. In semi-supervised clustering, a clustering algorithm is likewise parameterized, but usually for application to a single large set of items (rather than multiple smaller sets of items) where information on the clustering structure of the input items is incom-

plete. As such, there is often no interest in learning a model parameterization that could be applied to new sets of items, whether it be in the form of a learned metric or other transferable learned knowledge. This is a key component of supervised clustering.

Semi-supervised clustering methods augment an unsupervised clustering algorithm with information about how some of the items being clustered should relate to each other. In this setting one does not get the complete clustering of the set as described in the purely supervised case. Rather, information is incomplete, usually in the form of pairwise constraints, e.g., "items a and b should be in the same cluster" or "should not be in the same cluster." When clustering the data, the semi-supervised clusterer attempts to fulfill the constraints as best it can. This is distinct from supervised clustering, since in supervised clustering one has sets of items and complete partition information on these training sets, rather than incomplete information covering only a certain subset of pairs within a single input set.

Some of these semi-supervised clustering methods modify a clustering algorithm so it incorporates this supervision information, but does not parameterize a distance or similarity measure. For example, Aggarwal et al. [1] describe a minimal approach based on cluster seeds. The k-means algorithm is implemented by starting with seed cluster centroids that are iteratively refined in a greedy fashion to minimize intracluster distance (or alternatively maximize intracluster similarity), and has a strong tendency to find local minima for its objective function. Aggarwal et al. take advantage of this tendency to fall into local minima by having the initial cluster seeds be the same as those centroids seen in the training data, thus leaving the clustering algorithm predisposed to finding clusters close to the

initial starting points [1]. (Interestingly, were the k-means algorithm not so prone to fall into local optima, this algorithm would be ineffective.) Wagstaff et al. [110] propose an algorithm that likewise does not modify the distance metric at all, but directly constrains the k-means clustering algorithm so as to respect constraints about what sets of points should or should not be together; in the event the clusterer comes up with a cluster that violates the constraints in the k-means iteration, the items are reassigned to satisfy constraints and the algorithm continues until convergence.

De Bie and Cristianini present a method on learning a metric, with the stated purpose of clustering [32]. It works through defining a metric parameterized through a matrix W, where the metric between two points x_i and x_j is

$$(x_i - x_j)^T W W^T (x_i - x_j) \tag{1.11}$$

where the W is derived through an eigenanalysis of the vectors and their cluster constraints. This is remarkably similar to metric learning techniques described in Section 1.7 both in formulation and in algorithmic process, but the eigenanalysis would capture information about the global structure of how the data would cluster, despite clustering not being used directly in the optimization procedure.

Cohn et al. incorporate user feedback of clusterings of documents the form "these two documents should (not) be in the same cluster," and use these constraints to improve the distance metric between pairs of elements [22]. The clustering procedure used is a mixture of distributions, where each cluster corresponds to a different distribution, and each document has a probability of being generated according to that distribution. A document's probability is modeled as a weighted product of the probabilities of the words. By changing weights corresponding to each word, one modifies the distribution generating the document. The approach

taken in this paper is to, through iterative hillclimbing, choose a weighting so that the KL divergence of the distribution for two documents is small or large depending upon whether these documents should or should not be clustered.

Some methods do directly include the clustering procedure in the metric optimization procedure for semi-supervised clustering. Bilenko et al. [10] and Basu et al. [8] produce algorithms called, respectively, MPCK-Means and Hidden Markov Random Field k-Means (HMRF K-Means), which both incorporate must-link and cannot-link constraints through an EM procedure. These procedures first cluster data, and second modify the distance measure to "fix" any mistakes that occurred during the clustering. The algorithms then iterate over these two steps until convergence. The first works through application of a matrix update procedure and the second works through MAP inference on a graphical model, but the two methods appear almost identical in intent. An additional paper by Kulis et al. [60] refines Basu et al. [8] for the case of kernel clustering, where points do not necessarily exist as individual points in an explicit vector space in which they are clustered. This would be the case in, for example, typical representations of noun-phrase coreference [100]. These methods all modify both the clustering procedure to respect constraints, and also parameterize the distance metric as they perform clustering.

To summarize, semi-supervised clustering methods may seem closely related to supervised clustering methods, but the natural consequence of the typical target application, the clustering of a single dataset for which there is incomplete information and the lack of concern for transfer to new clustering, leads to very different problem formulations that are often inappropriate for the supervised clustering setting.

1.7 Relation to Metric and Kernel Learning

The preceeding material on learning distance and similarity measures for clustering may bring to mind the field of metric and kernel learning. This work could be and has been used to learn distance metrics for the purpose of clustering, but there is a substantial difference between "learning a metric" and "learning a metric so that a clustering algorithm will perform well," as argued in Section 1.3. Also, the primary application of these metric learning papers is to improve kNN classifiers [24]. Despite these differences, however, it is nonetheless a closely related field. As we shall see, these metric learning algorithms almost uniformly learn a similar type of distance metric.

Davis et al. [31] describe learning Mahalanobis distances, which generalize Euclidean distances through admission of linear scaling and rotations of the feature space. The algorithm is phrased in terms of learning a metric parameterized by a matrix A so as to minimize the distance between similar and dissimilar points. For example, two points x_i, x_j in a vector space would have distance

$$(x_i - x_j)^T A(x_i - x_j). \tag{1.12}$$

For example, if $A = I$, then this is standard squared Euclidean distance between x_i and x_j. If A is diagonal, this is Euclidean distance with corresponding feature weights. If A is not diagonal, the measure allows for correlation between features.

The matrix A is parameterized so that pairs of points that are similar and dissimilar have this distance less than a certain threshold and greater than a different threshold, respectively. Furthermore, this is subject to regularization of the form that the KL divergence between A and a certain prior parameterization A_0 is minimized, so the algorithm works to satisfy the constraints while keeping the

transformation as close to a priori notions of what the "right" parameterization should look like.

Weinberger et al. [111] propose learning a distance metric through a linear weighting of terms. The linear weights are learned through an optimization problem that simultaneously punishes long distances between points in the same group and short distances between points in different groups. The optimization procedure minimizes the sum of the distances (or inverses of the distances) for dissimilar (or similar) points, where the contribution of the distances is weighted according to some learning meta-parameters that the user of this learning procedure must set.

Lanckriet et al. [65, 66] present a procedure to employ semi-definite programming to maximize the alignment between a learned kernel matrix (really a weighted sum of provided kernel functions, where a weighting is the learned parameterization) and the labels assigned to points. Though phrased for transductive classification, nothing prevents this method from being used for other applications where learning a kernel function would be appropriate.

Xing et al. [112] describe an elegant approach. The data consists of a set of points we want to cluster $\{x_i : i \in 1..n\}$ with $x_i \in \mathbb{R}^N$. As in the typical semi-supervised learning setting, there are two constraint sets $\mathcal{S}$ and $\mathcal{D}$, where $\mathcal{S}$ contains pairs that should be similar and $\mathcal{D}$ contains pairs that should be different in the learned metric. The paper considers a distance metric $d_A(x, y)$ parameterized by a positive semidefinite matrix A, identical to that shown in [31] up to a squaring.

$$d_A(x, y) = \|x - y\|_A = \sqrt{(x - y)^T A (x - y)} \tag{1.13}$$

We can then produce the following optimization problem:

Optimization Problem 1. (Xing et al.'s Distance Learning)

$$\text{argmin}_A \quad \sum_{(x_i,x_j)\in\mathcal{S}} \|x_i - x_j\|_A^2 \tag{1.14}$$

$$s.t. \quad \sum_{(x_i,x_j)\in\mathcal{D}} \|x_i - x_j\|_A \geq 1. \tag{1.15}$$

The remainder of the algorithmic description focuses on establishing ways to make this learning problem tractable for the case where A is not diagonal.

In a similar vein, Tsang and Kwak introduce a kernel learning algorithm [105]. They suppose that for two patterns x_i, x_j in the input space $\mathbb{R}^p$, there is an inner matrix product $\langle x_i, x_j \rangle = s_{ij} = x_i^T M x_j$, where $M \in \mathbb{R}^{p\times p}$ is a positive semi-definite matrix. Since M is s.p.d. it can be factored as a product of a matrix and its transpose, so they rewrite s_{ij} as $s_{ij} = x_i^T A A^T x_i$ where A is a $p \times p$ matrix. In their learning framework, A is some learned matrix for a learned metric $\tilde{d}$, whereas M corresponds to an original metric d:

$$d_{ij}^2 = (\phi(x_i) - \phi(x_j))^T M (\phi(x_i) - \phi(x_j)) \tag{1.16}$$

$$\tilde{d}_{ij}^2 = (\phi(x_i) - \phi(x_j))^T A A^T (\phi(x_i) - \phi(x_j)) \tag{1.17}$$

The algorithm tries to learn an A with the following optimization problem, for $\mathcal{S}$ and $\mathcal{D}$ as sets of pairs of elements that are supposed to be similar or different, respectively:

Optimization Problem 2. (Tsang and Kwak Distance Learning)

$$\text{argmin}_{A,\gamma,\xi_{ij}} \ \frac{1}{2}\|AA^T\|^2 + C_{\mathcal{S}}\left(\frac{1}{|\mathcal{S}|}\sum_{(x_i,x_j)\in\mathcal{S}}\tilde{d}_{ij}^2\right) + C_{\mathcal{D}}\left(-\nu\gamma + \frac{1}{|\mathcal{D}|}\sum_{(x_i,x_j)\in\mathcal{D}}\xi_{ij}\right) \quad (1.18)$$

$$s.t. \quad \forall (x_i,x_j)\in\mathcal{D} \quad \tilde{d}_{ij}^2 - d_{ij}^2 \geq \gamma - \xi_{ij} \quad (1.19)$$

$$\xi_{ij} \geq 0. \quad (1.20)$$

Here, $C_{\mathcal{S}}$, $C_{\mathcal{D}}$, and ν are tunable positive valued parameters. The $\|AA^T\|^2$ term is used to encourage the rank of A to be low for sparsity. The larger $C_{\mathcal{S}}$ is, the more the algorithm attempts to make the learned distance measure for $(x_i, x_j) \in \mathcal{S}$ low. The ξ_{ij} serve a similar function to slack variables in a generic SVM in that their minimization punishes pairs in $\mathcal{D}$ from being closer than the threshold γ, and the larger $C_{\mathcal{D}}$ is, the less the program tolerates large slack. Finally, the larger ν is, the larger the optimization program tries to make the margin γ between distances of pairs in $\mathcal{S}$ and distances of pairs in $\mathcal{D}$. For a more intuitive explanation, it chooses an A such that close pairs are close, while distant pairs are far apart.

Schultz and Joachims describe a different way to think about learning a distance metric [95]. Instead of the $\mathcal{S}$ and $\mathcal{D}$ sets that say, "these elements are similar/different," constraints in [95] are of the form "a is closer to b than a is to c." In this way, the desired closeness is scaled in terms of relative preferences. Relative constraints have been unnecessary for the hard clusterings we have so far considered, but this type of distance learning measure may be useful in situations where one needs to tune a distance metric with more finesse than is allowed by absolute binary relationships of "similar" and "different."

Similar to other formulations, we have a metric $d_{A,W}$ parameterized by matrices

A and W:

$$d_{A,W}(x,y) = \sqrt{(x-y)^T AWA^T(x-y)}. \tag{1.21}$$

W is a positive diagonal matrix whose diagonal entries are learned by this algorithm. A is a real matrix provided a priori. The paper discusses two possible choices for A. One is $A = I$, of course. The other is $A = \Phi$ with the ith column equal to $\phi(x_i)$, that is, training vector x_i projected into the feature space; this A allows one to use kernel functions representing products within this feature space provided by ϕ.

An optimization problem to learn this metric is given in OP 3.

Optimization Problem 3. (Schultz and Joachims Distance Learning)

$$\begin{aligned}
\min \quad & \frac{1}{2}\|AWA^T\|_F^2 + C\sum_{i,j,k}\xi_{ijk} && (1.22)\\
s.t. \quad \forall (i,j,k) \in P_{train}. \quad & (x_i - x_k)^T AWA^T(x_i - x_k) - (x_i - x_j)^T AWA^T(x_i - x_j) \\
& \geq 1 - \xi_{ijk} && (1.23)\\
& \xi_{ijk} \geq 0 && (1.24)\\
& W_{ii} \geq 0 && (1.25)
\end{aligned}$$

In conclusion, methods in this field learn a metric so that points which are similar and different are kept close and far in a learned metric, respectively. They all learn some sort of matrix inner product $\langle x, y\rangle = x^T By$, where the form of B and how it is learned differs from paper to paper. Even in this cursory survey, we have seen a tremendous variety of methods in this area, all with different opinions about the proper optimization criteria. In this way, supervised clustering work could be viewed as another metric learning problem, except the criteria for

optimization for a supervised clusterer is that the metric or measure learned is such that a clusterer will perform well in partitioning the data when run on the similarity matrix. However, the converse does not hold, as metric learning does not by itself constitute a supervised clustering method since the optimization criteria are typically much different.

1.8 Summary

To summarize this chapter, we introduced the problem of supervised clustering. In supervised clustering, one wishes to learn a clustering function to produce desirable partitions of input sets, with applications in noun-phrase coreference, image segmentation, news clustering, speech segmentation, and others. We can phrase the learning problem more technically as learning a parameterization $\mathbf{w}$ for a clustering function $h_{\mathbf{w}} : \mathcal{X} \rightarrow \mathcal{Y}$ through a training set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\} \in (\mathcal{X} \times \mathcal{Y})^n$, drawn from the set of all possible item sets $\mathcal{X}$ and partitions of that item set $\mathcal{Y}$. The training example $(\mathbf{x}_i, \mathbf{y}_i)$ consists of an item set, and a complete partitioning of this set. The goal in choosing $\mathbf{w}$ is to choose one such that, informally, the clustering algorithm will perform well over future training examples, or more formally, such that the risk $\mathcal{R}_P(h_{\mathbf{w}}) = \int_{\mathcal{X} \times \mathcal{Y}} \Delta(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x}))\ dP(\mathbf{x}, \mathbf{y})$ is minimized for the unknown generating distribution $P(\mathbf{x}, \mathbf{y})$. The advocated form of the parameterization $\mathbf{w}$ is to parameterize pairwise similarity measures for items $x_i, x_j \in \mathbf{x}$ so that, when clustering $\mathbf{x}$ on this similarity measure, the desired partition $\mathbf{y} = h_{\mathbf{w}}(\mathbf{x})$ is produced. Section 1.4 discussed existing prior work in this field.

While there are existing methods of learning similarity measures, the goal of

supervised clustering is to learn this parameterization with an aim of producing desirable clustering, which is distinct from learning local pairwise similarities, or a general similarity measure as argued in Section 1.3 and Section 1.7. They are not optimizing to the right criteria.

Furthermore, existing approaches in semi-supervised learning are insufficient for this task. Semi-supervised clustering concerns learning how to cluster a single data set with incomplete information on that data set, and is unconcerned with transferring this knowledge to clustering new sets of items, the primary concern of supervised clustering.

The next chapter, Chapter 2, will introduce the basic machine learning framework that forms the basis of our implementation of supervised clustering methods. These methods will parameterize similarity measures directly optimized to cluster performance as measured by our loss.

CHAPTER 2

STRUCTURED LEARNING

Before we leap into the discussion of supervised clustering methods of Chapter 3 and Chapter 4, we must introduce the machine learning frameworks which will form the basis of that work. This chapter will discuss discriminative methods for learning functions for structured outputs. Rather than discussing structured learning for clustering specifically, which is a particular type of structured output, the algorithms for learning structured outputs shall be presented generally. Presentation of the techniques for utilizing these methods for supervised clustering specifically will be presented and analyzed in Chapter 3 and Chapter 4. Absolutely critical to understanding is Section 2.1's material on structural support vector machines, with material on other learning methods being important for a deep appreciation.

Speaking generally, no matter the application, nearly all machine learning methods learn a function. A learned function can output a binary label for an input document as in binary classification, or produce of a parse tree for an input sentence, or provide a protein alignment, or even output a partitioning of an input set as in clustering. Machine learning is rife with examples of functions (trained in the case of supervised machine learning) to produce certain outputs for inputs. Functions $h : \mathcal{X} \rightarrow \mathcal{Y}$ produce an output $\mathbf{y} \in \mathcal{Y}$ from a range of possible outputs $\mathcal{Y}$ given an input $\mathbf{x} \in \mathcal{X}$ from a domain of possible inputs $\mathcal{X}$.

Structured learning concerns learning functions where the $\mathcal{X}$ and $\mathcal{Y}$ are potentially complex structured outputs. In applications like binary or multiclass classification, $\mathcal{Y}$ is a very small collection of scalar labels (e.g., "yes" or "no" in binary classification). In structured learning, we may potentially be trying to learn

functions with much more complicated functions, including functions which output sequence labels or parse trees for sentences, translations for sentences, or even clusterings for a given set of items.

Quite generally, in many machine learning applications these functions take the form of finding an output to achieve a maximization of a discriminant function f. The function $h : \mathcal{X} \to \mathcal{Y}$ maps the input $\mathbf{x} \in \mathcal{X}$ to some output $\mathbf{y} \in \mathcal{Y}$ such that some joint discriminant function $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is maximized.

$$h(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{argmax}} f(\mathbf{x}, \mathbf{y}) \tag{2.1}$$

To be clear, functions that are usually phrased as minimizations can be phrased as maximizations through an inversion of the discriminant function. For example, the minimum spanning tree problem is to find the sub-tree containing all nodes of a connected graph such that the sum of the included edges' weights are minimized, or, alternately, such that the negated sum of the included edges' weights are maximized.

In machine learning applications, many of the most popular methods, including those summarized in this chapter, parameterize the discriminant function by some model parameterization $\mathbf{w}$, and can phrase the discriminant function in this form

$$h_{\mathbf{w}}(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{argmax}} f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \underset{\mathbf{y} \in \mathcal{Y}}{\operatorname{argmax}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle \tag{2.2}$$

Despite the use of the inner product $\langle \cdot, \cdot \rangle$, it is worth noting that this inner product $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ is by no means necessarily a linear inner product, but could potentially be kernelized, so that $\mathbf{w}$ is a collection of $\mathbf{\Psi}$ vectors and associated α

coefficients such that $\mathbf{w} = \sum_{(i)} \alpha_{(i)} \mathbf{\Psi}(\mathbf{x}_{(i)}, \mathbf{y}_{(i)})$, so that

$$\begin{aligned} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle &= \left\langle \sum_{(i)} \alpha_{(i)} \mathbf{\Psi}(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}), \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \right\rangle && (2.3) \\ &= \sum_{(i)} \alpha_{(i)} \left\langle \mathbf{\Psi}(\mathbf{x}_{(i)}, \mathbf{y}_{(i)}), \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \right\rangle && (2.4) \\ &= \sum_{(i)} \alpha_{(i)} \mathcal{K}\left((\mathbf{x}_{(i)}, \mathbf{y}_{(i)}), (\mathbf{x}, \mathbf{y})\right) && (2.5) \end{aligned}$$

However, in typical applications and in most practice this $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ inner product does wind up being a perfectly straightforward linear product, with both $\mathbf{\Psi}(\cdot, \cdot)$ and $\mathbf{w}$ literally interpretable as real vectors in some vector space $\mathbb{R}^N$.

In supervised machine learning, this parameterization $\mathbf{w}$ is learned with the help of a training set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\} \in (\mathcal{X} \times \mathcal{Y})^n$.

2.1 Structural Support Vector Machines

Suppose that for a given supervised learning task we are attempting to learn some function $h_{\mathbf{w}} : \mathcal{X} \to \mathcal{Y}$ as described above. Suppose further that for our task we have some loss function $\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ which in principle measures the extent to which two outputs differ. The intended use of Δ is to gauge how far any output differs from some known correct output. Though the exact specification of a loss is strongly task dependent, a loss Δ typically has the following characteristics:

1. $\forall \mathbf{y} \in \mathcal{Y}$, it is the case that $\Delta(\mathbf{y}, \mathbf{y}) = 0$, that is, an output compared against itself incurs no loss.
2. $\forall \mathbf{y} \in \mathcal{Y}, \forall \hat{\mathbf{y}} \in \mathcal{Y} \setminus \mathbf{y}$, then $\Delta(\mathbf{y}, \hat{\mathbf{y}}) > 0$, that is, for unequal outputs, some loss is incurred.
3. Informally, it is generally desirable that $\Delta(\mathbf{y}, \hat{\mathbf{y}}) \leq \Delta(\mathbf{y}, \bar{\mathbf{y}})$ for $\bar{\mathbf{y}}$ which would

be a worse output than $\hat{\mathbf{y}}$ if $\mathbf{y}$ were the correct output. For example, in a typical sequence tagging task, an output that has a greater proportion of the sequence labels differing should be greater.

None of these characteristics is, strictly speaking, a requirement, though it is difficult to imagine many scenarios where violating them would be attractive.

Finally, suppose that in our task, input-output pairs are generated according to some fixed distribution $P(\mathbf{x}, \mathbf{y})$. Then, a possible goal for selecting a hypothesis $h_{\mathbf{w}}$ is one such that risk

$$\mathcal{R}_P(h_{\mathbf{w}}) = \int_{\mathcal{X}\times\mathcal{Y}} \Delta(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x}))\ dP(\mathbf{x}, \mathbf{y}) \tag{2.6}$$

is minimized, e.g., the expected value of the loss Δ for future inputs is minimized with the chosen $h_{\mathbf{w}}$. Since P is an unknown distribution and minimizing (2.6) is consequently impossible, the approach instead is to take a training sample $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y} : i = 1, \ldots, n\}$ which we assume is generated i.i.d. according to P, and approximate $\mathcal{R}_P(h_{\mathbf{w}})$ with the empirical risk

$$\mathcal{R}_{\mathcal{S}}(h_{\mathbf{w}}) = \frac{1}{n}\sum_{i=1}^{n} \Delta(\mathbf{y}_i, h_{\mathbf{w}}(\mathbf{x}_i)). \tag{2.7}$$

As we shall see, the structural SVM will optimize a bound on empirical risk subject to regularization criteria.

2.1.1 Structural SVM Optimization Problem

The structural SVM is a method which attempts to minimize $\mathcal{R}_{\mathcal{S}}(h_{\mathbf{w}})$. Given a discriminant function of the form $f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$, with hypotheses of the form $h_{\mathbf{w}}(\mathbf{x}) = \text{argmax}_{\mathbf{y}}\, f(\mathbf{x}, \mathbf{y})$, where $h_{\mathbf{w}}$ is a hypothesis parameterized by $\mathbf{w}$ (we

often just use h for simplicity when which parameterization $\mathbf{w}$ is being used is obvious in context) with training pairs in the form $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{X} \times \mathcal{Y} : i = 1, \ldots, n\}$, a hypothesis $h_{\mathbf{w}}$ may be learned with this quadratic program:

Optimization Problem 4. (Margin-Scaled Structural SVM QP)

$$\min_{\mathbf{w},\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \tag{2.8}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.9}$$

$$\forall i, \forall \mathbf{y} \in \mathcal{Y}: \quad \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle \geq \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + \Delta(\mathbf{y}_i, \mathbf{y}) - \xi_i \tag{2.10}$$

Note that the loss $\Delta(\mathbf{y}_i, \mathbf{y})$ of the constraint's associated output $\mathbf{y}$ is incorporated as the margin between the discriminant function for the correct output $\langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle$ and the incorrect output $\mathbf{y}$'s discriminant function $\langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle$. There is also an alternate formulation proposed that scales the slack by the loss instead of the margin, to wit:

Optimization Problem 5. (Slack-Scaled Structural SVM QP)

$$\min_{\mathbf{w},\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{n}\sum_{i=1}^{n}\xi_i \tag{2.11}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.12}$$

$$\forall i, \forall \mathbf{y} \in \mathcal{Y}: \quad \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle \geq \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + 1 - \frac{\xi_i}{\Delta(\mathbf{y}_i, \mathbf{y})} \tag{2.13}$$

These optimization problems learn a model which upper bounds empirical risk $\mathcal{R}_{\mathcal{S}}(h_{\mathbf{w}})$ as seen in the following theorem.

Theorem 1. (Structural SVM Empirical Risk Bound)

Under either OP 4 or OP 5, let $\boldsymbol{\xi}(\mathbf{w}) = \{\xi_i : i = 1, \ldots, n\}$ *be any set of slack*

variables feasible for any given $\mathbf{w}$*. Then*

$$\frac{1}{n}\sum_{i=1}^{n}\xi_i \geq \mathcal{R}_{\mathcal{S}}(h_{\mathbf{w}}). \tag{2.14}$$

In other words, the slack term of the optimization problem is C times an upper bound on empirical risk of the hypotheses parameterized by $\mathbf{w}$. The proof can be understood by reviewing the constraints in either program: given that the hypotheses function is of the form $h_{\mathbf{w}}(\mathbf{x}) = \text{argmax}_{\mathbf{y}} f_{\mathbf{w}}(\mathbf{x}, \mathbf{y})$, any loss in the empirical risk for a training example $(\mathbf{x}_i, \mathbf{y}_i)$ incurred through $\Delta(\mathbf{y}_i, h_{\mathbf{w}}(\mathbf{x}_i))$ must be the result of $f_{\mathbf{w}}(\mathbf{x}_i, h_{\mathbf{w}}(\mathbf{x}_i)) \geq f_{\mathbf{w}}(\mathbf{x}_i, \mathbf{y}_i)$. By working from (2.10) or (2.13), depending on whether we are using the margin or slack scaled structural SVM, since there must be a constraint associated with any $h_{\mathbf{w}}(\mathbf{x}_i)$, assuming all constraints are respected, this means $\xi_i \geq \Delta(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x}))$ which, if plugged into (2.7), results in the desired bound.

For the sake of completeness, it is worth noting that there are two other forms of the structural SVM optimization problem which instead has a squared slack in the slack penalty term of the objective function, e.g., an L2 norm instead of the L1 norm:

Optimization Problem 6. (Margin-Scaled Quadratic Slack Structural SVM QP)

$$\min_{\mathbf{w},\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{2n}\sum_{i=1}^{n}\xi_i^2 \tag{2.15}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.16}$$

$$\forall i, \forall \mathbf{y} \in \mathcal{Y}: \quad \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle \geq \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + \sqrt{\Delta(\mathbf{y}_i, \mathbf{y})} - \xi_i \tag{2.17}$$

Optimization Problem 7. (Slack-Scaled Quadratic Slack Structural SVM QP)

$$\min_{\mathbf{w},\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + \frac{C}{2n}\sum_{i=1}^{n}\xi_i^2 \tag{2.18}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.19}$$

$$\forall i, \forall \mathbf{y} \in \mathcal{Y}: \quad \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle \geq \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + 1 - \frac{\xi_i}{\sqrt{\Delta(\mathbf{y}_i, \mathbf{y})}} \tag{2.20}$$

Though these L2-slack variants of structural SVMs are an important part of the work on structural SVMs, they are rarely used in practice, and this work does not make use of the squared-slack variants.

2.1.2 Cutting Plane Algorithm

The obvious problem with OP 4 and OP 5 is that there are as many constraints as there are possible labels. In most structural learning problems, the number of possible labelings of each example, and consequently the number of constraints required in OP 4 or OP 5, is typically at least exponential in the size of an input $\mathbf{x}$. For example, in sequence tagging, for a sequence of size m where there are ℓ possible labels for each sequence item, there would be ℓ^m possible labelings for that sequence.

To take the example more germane to this work, let us consider the case of clustering for an item set $\mathbf{x}$ of size $|\mathbf{x}| = m$. If the number of clusters of our input is fixed at $|\mathbf{y}| = k$ then the number of possible clusterings is k^m. If the number of clusters is not fixed then the number of possible clusterings would be given by the m-th Bell number (without going into details, Bell numbers grow faster than

exponential but slower than factorial) [89]. Consequently, we cannot solve either of these optimization problems directly, despite their desirable properties. The approach taken by the structural SVM is to employ a cutting plane algorithm to dynamically generate and introduce violated constraints.

(Structural SVM Cutting Plane Algorithm)

Input: $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, C, ϵ
$S_i \leftarrow \emptyset$ for all $i = 1, \ldots, n$
repeat
  for $i = 1, \ldots, n$ **do**
    $H(\mathbf{y}) \equiv \Delta(\mathbf{y}_i, \mathbf{y}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle$ for margin scaling (OP 4)
    $H(\mathbf{y}) \equiv (\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + 1)\, \Delta(\mathbf{y}_i, \mathbf{y})$ for slack scaling (OP 5)
    compute $\hat{\mathbf{y}} = \operatorname{argmax}_{\mathbf{y} \in \mathcal{Y}} H(\mathbf{y})$
    compute $\xi_i = \max\{0, \max_{\mathbf{y} \in S_i} H(\mathbf{y})\}$
    if $H(\hat{\mathbf{y}}) > \xi_i + \epsilon$ **then**
      $S_i \leftarrow S_i \cup \{\hat{\mathbf{y}}\}$
      $\mathbf{w} \leftarrow$ optimize primal over $\bigcup_i S_i$
    end if
  end for
until no S_i has changed during iteration

Algorithm 1: Cutting plane algorithm to solve OP 4 or OP 5.

The cutting plane optimization algorithm is shown in Algorithm 1. To summarize, one would start with an empty set of constraints for each example, iteratively find the most violated constraint, introduce this constraint into a "working set," and reoptimize the quadratic program with this additional constraint [106, 107]. By most violated constraint for training example $(\mathbf{x}_i, \mathbf{y}_i)$, we mean the constraint in the full QP that requires the highest slack ξ_i. In order to find the example associated with the most violated constraint, since each individual constraint by itself requires a slack defined by a cost function H where

$$H(\mathbf{y}) \equiv \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}) \tag{2.21}$$

or, for slack scaling as in OP 5,

$$H(\mathbf{y}) \equiv \left(\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle + 1\right) \Delta(\mathbf{y}_i, \mathbf{y}), \tag{2.22}$$

it suffices to solve $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} H(\mathbf{y})$. Note that these cost functions $H(\mathbf{y})$ in (2.21) and (2.22) are derived from solving for the slack ξ_i in the constraints (2.10) and (2.13), respectively. Thus, this procedure finds the output associated with the constraint in the full quadratic program requiring the greatest slack, i.e., the most violated constraint. If the resulting constraint requires a slack that violates the current ξ_i by more than a predefined tolerance ϵ, then the constraint is added, and otherwise it is ignored. Upon an iteration where no valid constraint is added, the algorithm terminates, and of course as the constraint derived was the most violated constraint, no constraints in the full QP are violated more than ϵ in this iterative QP.

2.1.3 Theoretical Properties

This algorithm has several interesting theoretical properties which we present here. Not only are the properties themselves interesting insofar as they concern the correctness and practical application of Algorithm 1, but understanding why these properties are true is also important for understanding the algorithms of Chapter 3 and Chapter 4. In particular, a general understanding of the proofs will be critical to even a basic understanding of Chapter 5.

One property is that the resulting problem is correct with respect to the full quadratic problem, and respects the empirical risk bound of Theorem 1 up to tolerance ϵ.

Theorem 2. (Algorithm 1 Correctness)

By applying Algorithm 1 to solve either OP 4 or OP 5, the final solution $\mathbf{w}, \boldsymbol{\xi}$ *from Algorithm 1 will respect all constraints in the corresponding optimization problem within* ϵ*, and have an objective function value that does not exceed that from the original full problem.*

This is easy to see, since if any constraint *were* violated by more than ϵ, i.e., there is some constraint for example $(\mathbf{x}_i, \mathbf{y}_i)$ which requires slack $\hat{\xi}_i > \xi_i + \epsilon$, then it must be found and introduced by the algorithm. Further, since Algorithm 1 works over a subset of constraints relative to the original problems of OP 4 or OP 5, its objective function value cannot be greater than these original problems since the optimal solution to OP 4 or OP 5 is at least feasible in Algorithm 1.

Theorem 3. (Algorithm 1 Empirical Risk Bound)

By applying Algorithm 1 to solve either OP 4 or OP 5, let the resulting final solution's slack vector be $\boldsymbol{\xi} = [\xi_1, \ldots, \xi_n]^T$*. Then*

$$\epsilon + \frac{1}{n} \sum_{i=1}^{n} \xi_i \geq \mathcal{R}_{\mathcal{S}}(h_{\mathbf{w}}).$$

In other words, the slack bound on empirical risk is respected within ϵ under Algorithm 1. This is also easy to see. From Theorem 2 we know constraints are satisfied within ϵ, so for any slack variable ξ_i found by Algorithm 1, $\xi_i + \epsilon$ must be a feasible slack in the original OP 4 or OP 5, and by working from Theorem 1 we see the truth of Theorem 3.

Consequently, the solution found by Algorithm 1 must respect the empirical risk bound within ϵ. Though this is within a tolerance of ϵ, as a purely practical matter

this tolerance is meaningless since solvers for constrained quadratic problems find a solution within a certain tolerance anyway.

One of the most important and practically essential properties of this algorithm is that, despite arising from a quadratic program with typically, in its full form, an exponential or even infinite number of constraints, this algorithm converges within a polynomial number of iterations. Put more formally in the language of [107],

Theorem 4. (Algorithm 1 Iteration Complexity)

With $\bar{R} = \max_{\mathbf{y}_i, \mathbf{y}} \|\mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) - \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\|_2$*, and* $\bar{\Delta} = \max_{\mathbf{y}_i, \mathbf{y}} \Delta(\mathbf{y}_i, \mathbf{y})$*, and for a given* $\epsilon > 0$*, Algorithm 1 terminates after incrementally adding at most*

$$\max\left\{\frac{2n\bar{\Delta}}{\epsilon}, \frac{8C\bar{\Delta}\bar{R}^2}{\epsilon^2}\right\} \tag{2.23}$$

constraints in the margin scaling case and

$$\max\left\{\frac{2n\bar{\Delta}}{\epsilon}, \frac{8C\bar{\Delta}^3\bar{R}^2}{\epsilon^2}\right\} \tag{2.24}$$

constraints in the slack scaling case.

The proof of this is not as evident as the previous theoretical properties. Ignoring the mathematics, the idea of the proof is based on a few simple observations: (1) when we start our dual objective is 0 as we have no primal constraints initially, (2) by adding a constraint only when it is violated by more than ϵ, we guarantee that this dual objective must increase by a certain minimum amount, and (3) the dual objective value is upper bounded by the primal objective value which in turn is upper bounded by the objective value, corresponding to the trivial feasible point with $\mathbf{w} = \mathbf{0}$ and the slack variables for each example each set to the maximum possible loss function value.

Some existing applications of structural SVMs of this variety include label sequence learning, natural language parsing with weighted context free grammars [107], learning alignment models in computational biology [50, 114], collective classification of a test set allowing for optimization for multivariate performance measures [51], and learning ranking functions for search engines [115, 116].

2.1.4 1-Slack Structural SVM

A relatively recent advance in structural support vector machines is the 1-slack structural support vector machine [53]. This is a substantial improvement over the original structural SVM insofar as the training procedure runs in time linear in the number of training examples and desired precision. It follows from a reformulation of the structural SVM quadratic problem which, in effect, "combines" examples in a training set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$ into a single training example. In such a case, there is not a slack vector $\boldsymbol{\xi} = \xi_1, \ldots, \xi_n$ for every training example, but rather a single scalar slack variable ξ, hence it being termed the 1-slack variant.

Optimization Problem 8. (1-Slack Margin-Scaled Structural SVM QP)

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\xi \tag{2.25}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.26}$$

$$\forall(\bar{\mathbf{y}}_1, \ldots, \bar{\mathbf{y}}_n) \in \mathcal{Y}^n: \quad \left\langle \mathbf{w}, \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \right\rangle \geq \left\langle \mathbf{w}, \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{\Psi}(\mathbf{x}_i, \bar{\mathbf{y}}_i) \right\rangle + \frac{1}{n}\sum_{i=1}^{n} \Delta(\mathbf{y}_i, \bar{\mathbf{y}}_i) - \xi \tag{2.27}$$

Optimization Problem 9. (1-Slack Slack-Scaled Structural SVM QP)

$$\min_{\mathbf{w},\xi} \frac{1}{2}\|\mathbf{w}\|^2 + C\xi \tag{2.28}$$

$$s.t.\ \forall i: \quad \xi_i \geq 0, \tag{2.29}$$

$$\forall(\bar{\mathbf{y}}_1,\ldots,\bar{\mathbf{y}}_n) \in \mathcal{Y}^n: \ \left\langle \mathbf{w}, \frac{1}{n}\sum_{i=1}^{n} \Delta(\mathbf{y}_i,\bar{\mathbf{y}}_i)\boldsymbol{\Psi}(\mathbf{x}_i,\mathbf{y}_i) \right\rangle \geq \left\langle \mathbf{w}, \frac{1}{n}\sum_{i=1}^{n} \Delta(\mathbf{y}_i,\bar{\mathbf{y}}_i)\boldsymbol{\Psi}(\mathbf{x}_i,\bar{\mathbf{y}}_i) \right\rangle + \frac{1}{n}\sum_{i=1}^{n} \Delta(\mathbf{y}_i,\bar{\mathbf{y}}_i) - \xi \tag{2.30}$$

The general idea of OP 8 and OP 9 is that there is a constraint for every single combination of outputs across all training examples. This is in contrast to OP 4 and OP 5, which has a family of constraints for each training example, with one constraint per example per output. We can make similar theoretical statements about this formulation.

Theorem 5. (Equivalence of OP 4 with OP 8, and OP 5 with OP 9)

Any solution $(\mathbf{w}^*, \xi^*)$ *of OP 8 or OP 9 has an analogous solution* $(\mathbf{w}^*, \boldsymbol{\xi}^0)$ *of OP 4 or OP 5, respectively, with the* $\xi^* = \frac{1}{n}\|\boldsymbol{\xi}^0\|_1$ *(and vice versa).*

The general idea of the proof in [53] works by arguing through straightforward algebraic manipulation that for any given $\mathbf{w}$, the required slack scalar variable ξ in OP 8 or OP 9 and the required slack vector variable $\boldsymbol{\xi}$ in OP 4 or OP 5 related through $\xi = \frac{1}{n}\|\boldsymbol{\xi}\|_1$ lead to the same objective function value in all optimization problems, and consequently the two have the same optima.

In Algorithm 2, we present a cutting plane algorithm, analogous to Algorithm 1, to solve either OP 8 or OP 9. As with Algorithm 1, this cutting plane algorithm correctly solves OP 8 or OP 9 up to a tolereance ϵ.

(1-Slack Structural SVM Cutting Plane Algorithm)

Input: $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, C, ϵ
$S \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  {set up cost functions}
  $H_i(\mathbf{y}) \equiv \Delta(\mathbf{y}_i, \mathbf{y}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle$ for margin scaling (OP 8)
  $H_i(\mathbf{y}) \equiv (\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + 1)\, \Delta(\mathbf{y}_i, \mathbf{y})$ for slack scaling (OP 9)
end for
repeat
  for $i = 1, \ldots, n$ **do**
    compute $\hat{\mathbf{y}}_i = \operatorname{argmax}_{\mathbf{y} \in \mathcal{Y}} H_i(\mathbf{y})$
  end for
  compute $\xi = \frac{1}{n} \max_{(\bar{\mathbf{y}}_1, \ldots, \bar{\mathbf{y}}_n) \in S} \sum_{i=1}^{n} \max(0, H_i(\bar{\mathbf{y}}_i))$
  if $\frac{1}{n} \sum_{i=1}^{n} H_i(\hat{\mathbf{y}}_i) > \xi + \epsilon$ **then**
    $S \leftarrow S \cup \{(\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_n)\}$
    $\mathbf{w} \leftarrow$ optimize primal over S
  end if
until S has not changed during iteration

Algorithm 2: Cutting plane algorithm to solve the 1-slack structural SVM OP 8 or OP 9.

Theorem 6. (Algorithm 2 Correctness)

For any training set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$ *and* $\epsilon > 0$*, Algorithm 2 returns a solution* $(\mathbf{w}, \xi)$ *with a better objective function value than the optimal* $(\mathbf{w}^*, \xi^*)$ *for OP 8 (or OP 9), and for which* $(\mathbf{w}, \xi + \epsilon)$ *is feasible in OP 8 (or OP 9).*

Theorem 7. (Algorithm 2 Iteration Complexity)

With $\bar{R} = \max_{\mathbf{y}_i, \mathbf{y}} \|\mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) - \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\|_2$*, and* $\bar{\Delta} = \max_{\mathbf{y}_i, \mathbf{y}} \Delta(\mathbf{y}_i, \mathbf{y})$*, and for any* $0 < C$*, tolerance* $0 < \epsilon < 4\bar{R}^2 C$*, and training sample* $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$*, Algorithm 2 terminates after at most*

$$\left\lceil \log_2 \left(\frac{\bar{\Delta}}{4\bar{R}^2 C} \right) \right\rceil + \left\lceil \frac{16\bar{R}^2 C}{\epsilon} \right\rceil \tag{2.31}$$

iterations for the margin-scaled variant of Algorithm 2, or

$$\left\lceil \log_2 \left(\frac{1}{4\bar{R}^2 \bar{\Delta} C} \right) \right\rceil + \left\lceil \frac{16\bar{R}^2 \bar{\Delta}^2 C}{\epsilon} \right\rceil \tag{2.32}$$

iterations for the slack-scaled variant of Algorithm 2.

At first glance, this may seem like little more than an interesting but undesirable transformation, as OP 8 or OP 9 requires a great many more constraints, roughly $|\mathcal{Y}|^n$ instead of $n|\mathcal{Y}|$. However, theoretical results of [53] show this formulation is relatively sparse in the dual SVM problem, dual solution density being independent of training sample size. Importantly, note Theorem 7's independence on training set size n, versus the analogous Theorem 4 for Algorithm 1. In the case of learning linear parameterizations $\mathbf{w}$, which covers much of the work in structured prediction, including the work in this thesis, the resulting training procedure is linear in the number of training examples. When learning a linear model parameterization $\mathbf{w}$, the training procedure is extraordinarily faster. Detailed theoretical and empirical analyses appear in [52, 53].

This one-slack formulation has substantial theoretical and practical advantages, and is closer to the actual implementation of the structural SVM as used in much of this thesis. However, owing to the greater intuitive appeal of the original formulation, we still use the original structural SVM formulation as it appears in OP 4, OP 5, and Algorithm 1 in our discussions. Due to the equivalence of the two programs, identical requirements for practitioners to exploit either structural learning algorithm, and the similarity of the theoretical results, we can hold such discussions without too many compromises. When appropriate or relevant, we will clarify which variant is being used in actual practice.

2.1.5 Approximations in Structural SVMs

The theoretical results of Section 2.1.3 and Section 2.1.4 give us confidence in applying structural support vector machines to structured prediction problems, but they rely upon the separation oracle $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ being tractable, i.e., we can find the most violated constraint. However, in some structured prediction problems, particularly those where the prediction problem is intractable, we can no longer guarantee a tractable $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$. Without this guarantee, many of the existing proofs of the theoretical properties no longer hold. However, as we shall see in the proposed frameworks of Chapter 3 and Chapter 4, these methods still produce desirable results empirically, and we shall treat the problem of approximations and structural SVMs in great detail in Chapter 5.

2.2 Maximum Margin Markov Networks

Maximum margin Markov networks (M^3N) represent a different approach to solve the structural SVM quadratic program in OP 4 [104]. In order to achieve tractability, it restricts its attention to a significant subcase of structural learning: supervised learning over Markov networks. In a Markov network, we have an undirected graph $G = (V, E)$ with each node in V corresponding to one in a set of random variables X, and an edge $\{u, v\} \in E$ representing a dependency between the variables u and v, and a collection of non-negative potential functions ϕ_k for each clique k in G. The joint distribution of the network is given as

$$P(X = x) = \frac{1}{Z} \prod_{k \in cliques(G)} \phi_k(x_{\{k\}}) \tag{2.33}$$

where Z is the normalizing partition function so that the sum of the probabilities of all different assignments to X sums to 1, specifically:

$$Z = \sum_{\hat{x}} \left[\prod_{k \in cliques(G)} \phi_k(\hat{x}_{\{k\}}) \right] \tag{2.34}$$

where $\hat{x}$ is enumerated over all possible assignments to $\hat{x}$. Let us further suppose that all potential functions ϕ_k in log space take the form of

$$\log \phi_k(x_{\{k\}}) = \left\langle \mathbf{w}, \psi(k, x_{\{k\}}) \right\rangle \tag{2.35}$$

where $\mathbf{w}$ is some weight vector shared amongst all the potential functions, and ψ is a function taking two inputs: the clique k, and values for the variables in the clique $x_{\{k\}}$. Naturally, when one does induction over this structure to assign values to variables given a network with potentials, one is interested in finding $\text{argmax}_x P(X = x)$.

To give the common canonical example, for the problem of part-of-speech tagging with a standard sequence tagger, the nodes V would represent words in a sentence, edges would exist between adjacent words in the sentence, the variable assignments to X would represent the part of speech assigned to each word, and the $\psi(k, x_{\{k\}})$ would, in the typical implementation, select out the weights in $\mathbf{w}$ relevant to the likelihood that the words in k would have the parts of speech indicated by $x_{\{k\}}$ and that these two parts of speech would be adjacent.

In this formulation, the familiar $\mathbf{x}, \mathbf{y}$ input-output pairs are of the form where $\mathbf{x}$ represents some structure from which one may induce a Markov network (e.g., the sequence of words in a sentence $\mathbf{x}$ inducing a chain Markov network of the same length), and the $\mathbf{y}$ represents the variable assignments in that network. Let us restrict our attention to pairwise Markov networks for now (i.e., all cliques are edges). Then, for an input pattern $\mathbf{x}$ inducing a graph structure $G_{\mathbf{x}} = (V_{\mathbf{x}}, E_{\mathbf{x}})$,

recall that the potential for the edge $\{i,j\} \in E_{\mathbf{x}}$ with corresponding variable assignments y_i, y_j is $\phi_{\{i,j\}} = \exp\left[\langle \mathbf{w}, \psi(i,j,y_i,y_j)\rangle\right]$, with the overall distribution

$$P(\mathbf{y}|\mathbf{x}) = \frac{1}{Z}\exp\left[\sum_{\{i,j\}\in E_{\mathbf{x}}} \langle \mathbf{w}, \psi(i,j,y_i,y_j)\rangle\right] = \frac{1}{Z}\exp\left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x},\mathbf{y})\rangle\right]. \tag{2.36}$$

In the language of the structural SVM, the $\mathbf{\Psi}(\mathbf{x},\mathbf{y}) = \sum_{\{i,j\}\in E_{\mathbf{x}}} \psi(i,j,y_i,y_j)$, with the log probability given as

$$\log P(\mathbf{y}|\mathbf{x}) = -\log Z + \left[\sum_{\{i,j\}\in E_{\mathbf{x}}} \langle \mathbf{w}, \psi(i,j,y_i,y_j)\rangle\right] = -\log Z + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x},\mathbf{y})\rangle, \tag{2.37}$$

so our hypothesis as in the case of the structural SVM is of the form $h_{\mathbf{w}}(\mathbf{x}) = \text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x},\mathbf{y})\rangle$.

Unlike structural SVMs, M^{3}Ns require a loss function $\Delta(\mathbf{y},\hat{\mathbf{y}})$, which decomposes over elements in $\mathbf{y}$ and $\hat{\mathbf{y}}$. As $\mathbf{\Psi}$ is a sum of local feature functions ψ, for a given input pattern $\mathbf{x}$, Δ becomes a sum of local losses δ over all vertices $i \in V_{\mathbf{x}}$, with

$$\Delta(\mathbf{y},\hat{\mathbf{y}}) = \sum_{i\in V_{\mathbf{x}}} \delta(i, y_i, \hat{y}_i) \tag{2.38}$$

as the proportion of predictions within $\mathbf{y}$ and $\hat{\mathbf{y}}$ that differ between teh two inputs, that is, $\delta(i,y_i,\hat{y}_i) = \frac{1}{|V_{\mathbf{x}}|}\mathbb{1}_{y_i=\hat{y}_i}$, where $\mathbb{1}_\cdot$ is the indicator function returning 1 or 0 if its input is true or false, respectively.

In the full structural SVM quadratic program, we have one dual variable $\alpha_{\mathbf{x}}(\mathbf{y})$ for every wrong labeling $\mathbf{y}$ of every example $\mathbf{x}$. While the work of [106] deals with this exponentially sized body of constraints by iteratively selecting and introducing the dual variables associated with the most violated constraint, in contrast, the work of [104] takes advantage of the special structure of the Markov network and reformulates the dual program with "marginal" dual variables $\mu_{\mathbf{x}}(y_i) = \sum_{\mathbf{y}\sim[y_i]} \alpha_{\mathbf{x}}(\mathbf{y})$

and $\mu_{\mathbf{x}}(y_i, y_j) = \sum_{\mathbf{y} \sim [y_i, y_j]} \alpha_{\mathbf{x}}(\mathbf{y})$. Here, $\mathbf{y} \sim [y_i, y_j]$ denotes the set of all labelings $\mathbf{y}$ with the variable assignments y_i, y_j in positions i, j, respectively. Given our training set S, we can then pose an alternate dual quadratic program as follows:

$$\begin{aligned}
\max \quad & \sum_{(\mathbf{x}_i, \mathbf{y}_i) \in S} \sum_{u \in V_{\mathbf{x}}} \sum_{y_u} \mu_{\mathbf{x}_i}(y_u) \delta(u, y_{iu}, y_u) \\
& -\frac{1}{2} \sum_{\substack{(\mathbf{x}_i, \mathbf{y}_i), \\ (\mathbf{x}_j, \mathbf{y}_j) \in S}} \sum_{\substack{(u,v) \in E_{\mathbf{x}_i} \\ y_u, y_v}} \sum_{\substack{(r,s) \in E_{\mathbf{x}_j} \\ y_r, y_s}} \mu_{\mathbf{x}_i}(y_u, y_v) \mu_{\mathbf{x}_j}(y_r, y_s) \left\langle \psi(u, v, y_u, y_v), \psi(r, s, y_r, y_s) \right\rangle \ . \\
\text{s.t.} \quad & \textstyle\sum_{y_u} \mu_{\mathbf{x}}(y_u, y_v) = \mu_{\mathbf{x}}(y_v), \sum_{y_u} \mu_{\mathbf{x}}(y_u) = C, \mu_{\mathbf{x}}(y_u, y_v) \geq 0
\end{aligned}$$

In this formulation, we now have a number of dual variables polynomial in the length of the sequences and number of possible local labelings, and in the event where the Markov networks together form a forest, this formulation reaches the same solution as the original structural quadratic program.

In the event where one has 3-cliques, one can introduce even more marginal dual variables defined over these cliques, and with loops, one can "triangularize" the dependency graph. Of course, triangularization and subsequent introduction of 3-clique dual variables leads to an exponential number of dual variables in the size of both loops and cliques, but on certain classes of problems, the loops and cliques are small enough so that this is a reasonable suggestion. However, in a case where the graphical model holds a larger clique, or a very large loop as is common in some applications, the number of variables required in the optimization problem can become very large to the point where solving the problem becomes intractable.

The suggestion in this intractable case is to simply solve the QP with its pairwise marginal dual variables, as a "relaxed" version of the full problem, e.g., ignore any loops and just focus on enforcing local consistency. Though the theoretical guarantees of equivalence to OP 4 no longer hold, they empirically demonstrate the effectiveness of this method on the WebKB data [104]. In this problem each node

represents a web page, and each edge represents a link between the two pages. The web pages do not comprise a tree nor a graph that can be tractably triangularized, so the collective classification of the web pages relies upon the workings of this relaxation.

Closely related work features a grid Markov Random Field employed to segment 3D scan data, with model parameters used through a max margin framework [4]. In this work, they take the original OP 4. They reformulate the "family" of linear constraints consisting of a single constraint for each possible wrong answer

$$\forall i, \forall \mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i : \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle \geq \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}) - \xi_i \tag{2.39}$$

and reformlate it into the single non-linear constraint

$$\forall i : \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + \xi_i \geq \max_{\mathbf{y} \in \mathcal{Y} \setminus \mathbf{y}_i} \left(\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}) \right) \tag{2.40}$$

This constraint has the inference procedure in the maximization term. In this case, the maximization procedure for the Markov random field can be shown to be equivalent to an integer linear program, which is relaxed to a real LP. By "folding" this LP back into the non-linear term of the constraint, with some algebraic manipulation the authors derive a modified quadratic program that implicitly has the non-linear constraint. Of course, a real relaxation to compute this max term would produce an answer greater than or equal to the original integer linear program, leading to a QP possibly "overconstrained" with respect to OP 4. Though used specifically for the scan segmentation problem setting, this "folding" strategy could be used in any structural learning problem with an inference mechanism that can be expressed as a linear program, in line with [103]. Mathematically speaking, the resulting learning algorithm should be mathematically equivalent to our learning method for the special case where the separation oracle is computed though a linear-program.

Some applications that utilize methods derived from M^{3}N include sequence tagging [104], image segmentation [4], alignment models for translation [63, 70], and general translation [69].

Related to M^{3}N networks are maximum margin Bayesian networks [44]. Such methods based on directed models must satisfy normalization constraints that M^{3}N's, based on undirected Markov fields, need not obey, i.e., some of the probabilities must sum to 1. Though with general network topologies parameter inference in training and inference with the models is approximate, they do show improved performance when the directedness of the model encodes valuable information.

2.3 Search and Learn (SEARN)

Recall that the basic idea behind OP 4 as used in both structural SVMs and M^{3}Ns is, loosely, for each training example $(\mathbf{x}, \mathbf{y})$ to make the discriminant function f for the correct output greater than the discriminant function for any incorrect output $\hat{\mathbf{y}}$ so that $f(\mathbf{x}_i, \mathbf{y}_i) > f(\mathbf{x}_i, \hat{\mathbf{y}})$. Obviously, in the exact case where our problem setting allows non-approximate inference method which can exactly solve $h(\mathbf{x}) = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{x}, \mathbf{y})$ to find the maximizing $\mathbf{y}$, this will minimize empirical risk on the training sample.

There are two major problems that arise in the inexact case. First, in the work in the structural SVM and the M^{3}N, we observe that it is often no longer possible to ensure that the discriminant function f is maximized for correct inputs versus incorrect inputs. Second, if the inference procedure used in computing $\text{argmax}_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{x}, \mathbf{y})$ is no longer tractable, *even in cases where we can solve the first problem to our satisfaction*, merely ensuring that f is maximized for correct inputs

versus incorrect inputs no longer has any guarantee of minimizing empirical risk!

Consider, for example, some inference procedure in h that works via greedy search, with straightforward local decisions in attempting to approximate a maximizing argument $\mathbf{y}$ for $f(\mathbf{x}, \mathbf{y})$. We could definitely have the situation where the desirable output $\mathbf{y}^{opt}$ maxmizes f, but that h will instead find some suboptimal $\bar{\mathbf{y}}$ with $f(\mathbf{x}, \bar{\mathbf{y}}) < f(\mathbf{x}, \mathbf{y}^{opt})$ through convergence to a local minimum. However, it is also possible that if our learning procedure had been, in some sense, aware of the local decisions in the greedy algorithm, then a different parameterization $\mathbf{w}$ of f could have been found that would have led the local decisions to a final output of $\mathbf{y}^{opt}$.

Recent work by Daumé, called SEARN (a portmanteau of "search" and "learn"), implicitly incorporates information about inexact inference processes capable as being phrased as a form of greedy or beam search into the training method [27, 28, 29]. Though primarily motivated as a way to simplify and speed structural predictors and training, this may also provide benefits over other methods that have no consideration for the particular eccentricities of an inexact inference process.

It is important to note that the SEARN system is not limited in application to inexact inference methods (though inexact inference is what is primarily covered in its introductory literature, as the search in the provided examples is greedy), nor does the underlying learner have to be an SVM or any of its derivatives. One may apply it to any structured prediction problem which can be phrased as a search problem, the search can be any beam search, and make use of any multiclass classifier.

The method works by rephrasing the structured prediction problem as a search

problem. The search moves from state to state by means of a policy, and at the end of the search one has a "state" representing a complete structured prediction. For example, to take a simple example of NLP tagging, one may have states $(\mathbf{x}, \hat{y}_{1:t-1})$ consisting of the input sequence of words in the sentence $\mathbf{x}$, a sequence $\hat{y}_{1:t-1} = \hat{y}_1, \ldots, \hat{y}_{t-1}$ of already tagged words, and the "policy" (in reality a multiclass classifier) of finding the next state $(\mathbf{x}, \hat{y}_{1:t})$ where the word at position t now has its tag $\hat{y}_t$. (One may view a maximum entropy Markov model [74] for sequence prediction as a very special restricted case of a SEARN learner.)

As the learning algorithm focuses on training a model parameterization to make local decisions that lead to the correct global output, it is, in some sense, integrated into the search procedure, so it has the potential to be sensitive to peculiar tendencies of a method to fall into local minima.

2.4 Conditional Random Fields

Conditional random fields (CRFs) share similarities with M^{3}N learning, in that both are intended for the class of problem where, given an input pattern $\mathbf{x}$, one finds a labeling $\mathbf{y}$ of nodes in a probabilistic graphical model [64, 109].

Given an input $\mathbf{x} \in \mathcal{X}$ and an output $\mathbf{y} \in \mathcal{Y}$, we have the following $\mathbf{w}$ parameterized distribution for the probability of $\mathbf{y}$ given $\mathbf{x}$. (Note that this form is the same as the M^{3}N conditional distribution of (2.36).)

$$P(\mathbf{y}|\mathbf{x}; \mathbf{w}) = \exp\left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle - z(\mathbf{w}|\mathbf{x})\right] \tag{2.41}$$

Note that $\mathbf{\Psi}$ retains a very similar meaning as in the M^{3}N network, in that $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ is the sum of feature vectors for all cliques k in the underlying graphical model G

for which we are finding the node labels $\mathbf{y}$, so

$$\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \sum_{k \in cliques(G)} \psi(\mathbf{x}, \mathbf{y}_{\{k\}}) \tag{2.42}$$

Here, $\mathbf{y}_{\{k\}}$ represents the label configuration in $\mathbf{y}$ for the nodes in clique k. The value of the clique potential function for a clique k is therefore $\left\langle \mathbf{w}, \psi(\mathbf{x}, \mathbf{y}_{\{k\}}) \right\rangle$ Further, similar to the Z normalizing constant in the M^{3}N conditional distribution, we have the log partition function

$$z(\mathbf{w}|\mathbf{x}) = \log \left[\sum_{\hat{y}} \exp \left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \hat{\mathbf{y}}_i) \rangle \right] \right] \tag{2.43}$$

With this conditional probability for $P(\mathbf{y}|\mathbf{x}; \mathbf{w})$, we may write the conditional likelihood of the entire training sample $\mathcal{S} = ((\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n))$ with $\mathbf{x}_{[\mathcal{S}]} = \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ and $\mathbf{y}_{[\mathcal{S}]} = \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$ as

$$P(\mathbf{y}_{[\mathcal{S}]}|\mathbf{x}_{[\mathcal{S}]}; \mathbf{w}) = \prod_{i=1}^{n} P(\mathbf{y}_i|\mathbf{x}_i; \mathbf{w}) = \exp \left[\sum_{i=1}^{n} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle - z(\mathbf{w}|\mathbf{x}_i) \right] \tag{2.44}$$

Where a CRF differs substantially from the M^{3}N method is that instead of learning the parameters $\mathbf{w}$ with an aim of maximizing margin, what the goal is instead is to find the most likely parameterization $\mathbf{w}$ of the model given the training set $\mathcal{S}$, to wit:

$$P(\mathbf{w}|\mathbf{x}_{[\mathcal{S}]}, \mathbf{y}_{[\mathcal{S}]}) = P(\mathbf{w}) P(\mathbf{y}_{[\mathcal{S}]}|\mathbf{x}_{[\mathcal{S}]}, \mathbf{w}) \tag{2.45}$$

For their prior distribution over the parameters, they choose a zero mean Gaussian $P(\mathbf{w}) \propto \exp\left[-\frac{1}{2\sigma^2}\|\mathbf{w}\|^2\right]$. The goal in training is to find the most likely parameterization $\mathbf{w}^*$ given the training sample $\mathcal{S}$ (i.e., the posterior of the parameters), specifically:

$$\mathbf{w}^* = \operatorname*{argmax}_{\mathbf{w}} P(\mathbf{w}|\mathbf{x}_{[\mathcal{S}]}, \mathbf{y}_{[\mathcal{S}]}) \tag{2.46}$$

How can we calculate this? Note that according to Bayes' rule,

$$P(\mathbf{w}|\mathbf{x}_{[\mathcal{S}]}, \mathbf{y}_{[\mathcal{S}]}) \propto P(\mathbf{w}) P(\mathbf{y}_{[\mathcal{S}]}|\mathbf{x}_{[\mathcal{S}]}; \mathbf{w}) \tag{2.47}$$

Let $\mathcal{L}(\mathbf{w})$ be the negative log-posterior of the parameters $\mathbf{w}$, specifically:

$$\begin{aligned} \mathcal{L}(\mathbf{w}) &= -\log P(\mathbf{w}|\mathbf{x}_{[\mathcal{S}]}, \mathbf{y}_{[\mathcal{S}]}) + (\text{some constant}) && (2.48) \\ &= \frac{\|\mathbf{w}\|^2}{2\sigma^2} - \sum_{i=1}^{n} [\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle - z(\mathbf{w}|\mathbf{x}_i)] && (2.49) \end{aligned}$$

As $\mathcal{L}(\mathbf{w})$ is the negative log of the posterior, we can maximize this posterior by finding $\mathbf{w}$ that minimizes $\mathcal{L}(\mathbf{w})$.

The method of minimization employed in CRF training is typically some form of gradient descent on $\mathcal{L}$. The gradient is given as

$$\frac{\delta}{\delta \mathbf{w}} \mathcal{L}(\mathbf{w}) = \frac{\mathbf{w}}{\sigma^2} - \sum_{i=1}^{n} \left[\mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) - \overbrace{\sum_{\mathbf{y} \in \mathcal{Y}} P(\mathbf{y}|\mathbf{x}_i; \mathbf{w}) \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y})}^{E} \right] \tag{2.50}$$

The interesting portion of computing the gradient at each step is the term labeled E. This sum may be computed in time exponential in the size of the largest clique in the optimally triangularized version of the underlying graphical model G. This marginal term is calculated through the sum/product belief propagation algorithm. This requirement of a marginal over all possible outputs is a weakness of CRFs. In the case of graphical models, we have the sum-product algorithm to compute this marginal, but in other applications, computing a function over all possible inputs may be either intractable, or add complexity to the learning procedure, as it requires another algorithm aside from the inference step.

In the case of graphical models, the sum $\sum_{\mathbf{y} \in \mathcal{Y}} P(\mathbf{y}|\mathbf{x}_i; \mathbf{w}) \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y})$ in (2.50) may be computed in time exponential in the size of the largest clique in the optimally triangularized version of the underlying graphical model G. Chains and trees have maximal clique size of 2, but in cases where G has large cliques or loops it will no longer be tractable to do exact computation of the E term. For example, in the case where G takes the form of a grid or lattice (as is common in image processing

applications, for example), exact computation of E for the gradient is no longer possible.

In the case when G is not a general graphical model, one typically employs some form of approximation in computing this gradient, leading to approximate training of model parameters. In [109], a stochastic gradient descent method is employed which makes use of approximations of the gradient. [47] also uses gradient descent, utilizing contrastive divergence [48] to approximate the gradient in computing the step at each iteration. Bayesian CRFs, a method closely related to CRFs, in training utilizes an approximation of the posterior of the model parameters [85]. Discriminative random fields, another method closely related to CRFs, uses psuedolikelihood to estimate model parameters [61, 62].

Another interesting innovation relating to conditional random fields is that it might even be possible for a learning method based on approximate inference to, in some cases, do better than a CRF model built for exact inference, with a locally trained model giving better sequence predictions. In particular, a CRF that is trained in a piecewise fashion in some cases appears to perform better than a globally trained CRF [101]. The ability of a locally and, in some sense, "inexactly" trained sequence model to perform comparably to globally trained models was a feature in [84, 92] as well.

In particular, in [92] is a paper about the use of CRFs for sequence predictions in the case where one has constraints on the output that one knows a priori. For example, consider a simple semantic role labeling task, where one has a sentence and wishes to discover the verb-argument structure, where each "verb" has a single argument, and each argument itself is one of several types. Then one can have constraints difficult or impossible to include in standard Viterbi: for exam-

ple, one would want exactly one argument label, the active verb is provided as input, various verbs disallow certain types of arguments from being used, etc. The suggestion is to phrase the Viterbi sequence inference procedure as instead being an instance of an integer linear program. The flexibility of being an ILP allows them to include a more general class of constraints than can be accommodated by a Viterbi like algorithm. The paper is interesting and relevant to this work in two respects. First, the constrained inference procedure is not used in training, leading to a machine learning procedure which is, in some respect, "relaxed," as the evaluation inference mechanism differs, in some sense, from the inference mechanism for which the training algorithm is trying to optimize. Instead of training for a constrained sequence predictor, they train the model as a vanilla CRF for an unconstrained sequence predictor. Second, going even further, they utilize a training method which does not learn a model as a sequence *at all*, i.e., effectively just learning a multiclass classifier. Performance of the purely locally trained model without the ILP constraints is quite low, though with the inclusion of constraints the performance rises rapidly, even to the point of exceeding the performance of the "properly" globally trained model once all constraints are active.

2.5 Local Learning, Global Inference

There is also considerable work on models which are trained locally, but used to perform some global inference task. The primary thrust of the University of Illinois at Urbana-Champaign (UIUC) Cognitive Computation Group (CCG) led by Dan Roth is to apply machine learning techniques for reasoning and inference over natural language in a unified fashion. However, "unified" should not be taken to mean "non-modular," in contrast to other work that attempts to learn a truly

single "end to end" classifier without reliance upon a pipeline. What do we mean by a pipeline? Suppose one wants to do semantic role labeling on text as a small part of a larger information retrieval system, where various phrases are labeled according to what "role," they play in the sentence, that is, for "Anne saw Bob," we have "Anne" as the subject noun, "saw" is the action, and "Bob" is the object. What modules can we break this down into? We have POS tagging, which supports attempts to build a parse tree, which further support word sense disambiguation, which in turn supports semantic role labeling. While a somewhat more modern and fashionable approach might be to go directly from text to SRL outputs in a single learning framework with almost no intermediate representation, papers from the CCG quite typically employ subinference modules trained locally, but during inference these modules are arranged in a pipeline, and the outputs of these modules are selected with an aim of increasing global pipeline performance. This allows interactions among these modules up and down the pipeline. This practice of locally trained models combined together to perform inference, often in the form of a pipeline, is a theme endemic throughout a great deal of the CCG work [90, 82, 83, 84, 91]. This is in contrast to typical pipelined module framework which takes each successive stage of the pipeline as "correct" input for the next stage of the pipeline, leading to increased compounding of errors. This body of work may be viewed as a situation where inference could be exact (though as a practical nature it is not), but the learning process is approximate in that the individual pipeline elements are trained only locally.

2.6 Summary

This section introduced structured machine learning, the field of learning parameterizations for functions $h : \mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ could be complex structured inputs and outputs. This is in contrast to, for instance, the common task of binary classification with $\mathcal{Y} = \{-1, +1\}$. However, machine learning can be applied to far more sophisticated functions which can output, for instance, output sequence labels, or parse trees for sentences, translations for sentences, or even clusterings for a given set of items, tasks where structured prediction methods have been successful. This chapter summarized the major discriminative structured prediction frameworks, including conditional random fields, maximum margin Markov networks, but in most detail the structural support vector machine.

The structural support vector machine learner learns a parameterization $\mathbf{w}$ for a hypothesis function $h_{\mathbf{w}} : \mathcal{X} \rightarrow \mathcal{Y}$ which can be phrased in terms of maximizing some $\mathbf{w}$-parameterized discriminant function $f_{\mathbf{w}} : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}$, i.e., $h_{\mathbf{w}}(\mathbf{x}) = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} f_{\mathbf{w}}(\mathbf{x}, \mathbf{y})$. To find this parameterization, the structural SVM utilizes a quadratic problem with many constraints for each training example $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}$ for a training set $\mathcal{S}$, such that the discriminant function of the correct output $f_{\mathbf{w}}(\mathbf{x}_i, \mathbf{y}_i)$ is separated from that of any incorrect output $f_{\mathbf{w}}(\mathbf{x}_i, \mathbf{y})$. In the margin scaling variant this separation must be at least the loss $\Delta(\mathbf{y}_i, \mathbf{y})$ between the two with any violation punished with a slack variable. In the slack scaling variant, the separation must be 1, but the slack variable is scaled by $\Delta(\mathbf{y}_i, \mathbf{y})$ to punish violations of high loss examples more severely. Since this requires constraints for every possible wrong output, a cutting plane algorithm iteratively finds and introduces the most violated constraint until convergence. The algorithm is demonstrably correct theoretically, and terminates in a polynomial number of iterations. These methods will form the

basis for the methods in Chapter 3 and Chapter 4 for learning parameterizations for correlation clustering, k-means clustering, and spectral clustering.

CHAPTER 3

SUPERVISED CORRELATION CLUSTERING

Clustering techniques are often leveraged for any application where we wish to group sets of items. For example, in the noun-phrase coreference task, a single document's noun-phrases are clustered by which noun-phrases refer to the same entity [77], and in news article clustering, a single day's worth of news articles are clustered by topic [40]. However, it is often difficult to make these clustering methods produce desirable clusterings. Chapter 1 introduced the notion of supervised clustering, where a clustering algorithm is parameterized to produce desirable clusterings.

This chapter provides a supervised clustering method for correlation clustering [7, 33]. Correlation clustering's goal is to, given an item $\mathbf{x}$, find the clustering $\mathbf{y}$ which maximizes the sum of all pairwise similarities of items $x_i, x_j \in \mathbf{x}$ in the same cluster in $\mathbf{y}$. The attraction of correlation clustering lies in its ability to choose the number of clusters, its simplicity, and, though finding the optimal clustering under the correlation clustering criteria is an NP-hard problem, the clustering solution can be approximated efficiently.

The supervised correlation clustering method, based on the structural SVM learning methods of Section 2.1, parameterizes the pairwise similarity through supervised learning. By changing the parameterization, we change which clustering is optimal under the correlation clustering criteria. The supervision takes the form of a training set, where users provide complete clusterings of a few of these sets to express their preferences, e.g., provide a few complete clusterings of several documents' noun-phrases, or several days' news articles. From these training examples, we learn a function to cluster future item sets using the learning techniques de-

scribed in Section 2.1. We derive a method based on these techniques that learns an item-pair similarity measure as described in Section 1.2. The method we derive is capable of directly optimizing correlation clustering performance for multiple problem specific loss functions, and is computationally efficient.

To review the basic supervised clustering learning problem, the method receives a set $\mathcal{S}$ of n training examples $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\} \in (\mathcal{X} \times \mathcal{Y})^n$, all drawn i.i.d. from some distribution $P(X, Y)$, with the random variables X and Y taking values from the set $\mathcal{X}$ and $\mathcal{Y}$ respectively. $\mathcal{X}$ is the set of all possible sets of items and $\mathcal{Y}$ is the set of all possible clusterings (partitionings) of these sets. For any $(\mathbf{x}, \mathbf{y})$, $\mathbf{x} = \{x_1, x_2, \ldots, x_m\}$ is a set of m items, and $\mathbf{y} = \{y_1, y_2, \ldots, y_c\}$ with $y_i \subseteq \mathbf{x}$ is the partitioning of $\mathbf{x}$ into c clusters. The goal is to learn a clustering function $h_{\mathbf{w}} : \mathcal{X} \rightarrow \mathcal{Y}$ that can accurately cluster new sets of items.

Given a loss function that compares two clusterings $\Delta : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$, the training error for a clustering function $h_{\mathbf{w}}$ on an example $(\mathbf{x}, \mathbf{y})$ is $\Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y})$. The goal is to find $h_{\mathbf{w}}$ to minimize risk $Err_P(h_{\mathbf{w}}) = \int_{\mathcal{X} \times \mathcal{Y}} \Delta(h_{\mathbf{w}}(\mathbf{x}), \mathbf{y}) \; dP(\mathbf{x}, \mathbf{y})$, which we instead approximate by empirical risk $Err_S(h_{\mathbf{w}}) = \frac{1}{n} \sum_{i=1}^{n} \Delta(h_{\mathbf{w}}(\mathbf{x}_i), \mathbf{y}_i)$ since the distribution $P(\mathbf{x}, \mathbf{y})$ is unknown.

The approach taken here is to modify the similarity measure to encourage parameterizations $\mathbf{w}$ of our correlation clustering function $h_{\mathbf{w}}$ so it performs well under Δ. Modifying the similarity measure has some intuitive appeal: if you want news articles clustered by topic, a great clustering method using author similarity will probably produce worse results than a mediocre clustering method using topic similarity.

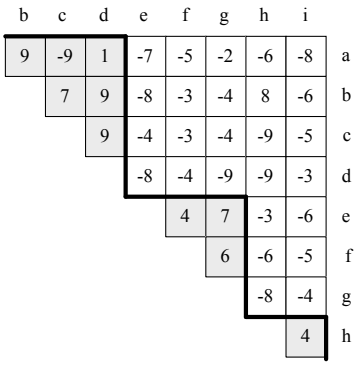

Figure 3.1: Correlation clustering on a matrix of similarities for items x_a through x_i, where shaded boxes indicate that a pair is considered to be in the same cluster. This represents the "optimal" clustering, e.g., x_a through x_d are joined, x_e through x_g are joined, and x_h and x_i are joined.

3.1 Correlation Clustering

For our clustering method, we use *correlation clustering* [7, 33]. The ideal correlation clustering of a set of items $\mathbf{x}$ is the clustering $\mathbf{y}$ maximizing the sum of similarities for item pairs in the same cluster, where K is a matrix of pairwise similarities. The objective function $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ is

$$f(\mathbf{x}, \mathbf{y}) = \sum_{y \in \mathbf{y}} \sum_{x_i, x_j \in y} K_{i,j} \tag{3.1}$$

with the ideal correlation clustering is the maximizing $\mathbf{y}$ for this objective function, that is, $\text{argmax}_{\mathbf{y}} f(\mathbf{x}, \mathbf{y})$.

Bansal et al. [7] originally introduced correlation clustering where all elements $K_{ij} \in \{-1, +1\}$, Joachims and Hopcroft [54] considered the case where $K_{ij} \in \{-1, 0, +1\}$, but we consider the more general correlation clustering where $K_{ij} \in \mathbb{R}$, a setting also explored by other authors [33, 102]. As shown in Figure 3.1, pairs considered dissimilar can appear in the same cluster if the net effect of including

them is positive (e.g., x_a and x_c, despite having a negative similarity $K_{ac} = -9$, are joined in the optimal clustering), and pairs considered similar should not be in the same cluster if the net effect of including them is negative (e.g., x_b and x_h, despite having similarity $K_{bh} = 8$, are not joined in the optimal clustering).

Correlation clustering could be optimized through many means, but we first consider a very simple greedy algorithm given in Algorithm 3. This algorithm starts with a trivial partition $\mathbf{y}$ with each item of the input set $\mathbf{x}$ in its own cluster, and iteratively joins the two clusters in $\mathbf{y}$ that most increase the correlation clustering objective function, until no joining will increase the correlation clustering objective function or there is nothing to join.

(GREEDY CORRELATION CLUSTERING)

Input: An input set of items $\mathbf{x}$, inferring similarity matrix $K \in \mathbb{R}^{|\mathbf{x}|\times|\mathbf{x}|}$
$\mathbf{y} \leftarrow \{\{x_i\} : x_i \in \mathbf{x}\}$
let $\textsc{Merge}(\mathbf{y}, y, y') \equiv (\mathbf{y} \setminus \{y, y'\}) \cup \{y \cup y'\}$
repeat
  $\bar{y}, \bar{y}' \leftarrow \text{argmax}_{y,y' \in \mathbf{y}: y \neq y'} \sum_{x_i \in y} \sum_{x_j \in y'} K(i, j)$
  if $\sum_{x_i \in \bar{y}} \sum_{x_j \in \bar{y}'} K(i, j) > 0$ **then**
    $\mathbf{y} \leftarrow \textsc{Merge}(\mathbf{y}, \bar{y}, \bar{y}')$
  end if
until $\mathbf{y}$ has not changed during an iteration, or $|\mathbf{y}| = 1$
return $\mathbf{y}$

Algorithm 3: A greedy approximation to correlation clustering.

An alternative to simple greedy approximation is a real relaxation approximation, either in the form of a linear [33] or semidefinite program [102]. We use a linear program equivalent to work appearing in [33]. In the linear program, each pair of items $x_i, x_j \in \mathbf{x}$ has a corresponding variable e_{ij} indicating the degree to which x_i and x_j are in the same cluster. For all the e_{ij} variables which we collectively term $\mathbf{e}$, the program is:

Optimization Problem 10. (Relaxed Correlation Clustering)

$$\max_{\mathbf{e}} \sum_{e_{ij} \in \mathbf{e}} e_{ij} \cdot K_{ij} \tag{3.2}$$

$$s.t. \quad e_{ij} \in [0,1], \quad e_{ij} = e_{ji}, \quad e_{ij} \geq e_{jk} + e_{ik} - 1. \tag{3.3}$$

By itself this relaxation does not produce a clustering since the e_{ij} may be fractional, but techniques exist for deriving a proper partitioning [33, 102].

We have defined correlation clustering and algorithms and methods to provide an approximate correlation clustering $\mathbf{y}$ given a set of items $\mathbf{x}$. We next discuss how to parameterize correlation clustering with the structural SVM.

3.2 Supervised Correlation Clustering with SVMs

This section describes our supervised correlation clustering algorithm. We define our model, summarize the structural SVM algorithm [106, 107], and then describe how to adapt the algorithm to clustering. We begin to describe how to apply structural SVMs to the problem of supervised correlation clustering, by first phrasing the supervised correlation clustering problem in terms of a structural SVM. We refer to the resulting method as SVM-CC (SVM supervised correlation clustering).

Our supervised correlation clustering method will modify the similarity measure so that the correlation clustering method presented in Section 3.1 produces desirable clusterings. Recall that our similarity matrix $K_{\mathbf{w}}$, which we now suppose is parameterized by $\mathbf{w}$, has entries corresponding to the similarity of pairs of items. This similarity is a real number indicating how similar the corresponding

pair is; positive values indicate the pair is alike and should be clustered, whereas negative values indicate negative evidence for co-cluster membership. Our choice of parameterization is to let each pair of different items $x_i, x_j \in \mathbf{x}$ have a feature vector $\psi(x_i, x_j) \equiv \psi_{i,j}$ to describe the pair. The entry in the similarity matrix K is then $K_{\mathbf{w}}(i,j) = \langle \mathbf{w}, \psi_{i,j} \rangle$.

How can we learn this parameterization? The structural SVM algorithm described in Section 2.1 provides a general framework for learning functions with complex structured output spaces [106, 107]. In order to phrase supervised correlation clustering as a structural SVM problem as shown in OP 4 or OP 5, we must first devise an appropriate $\Delta(\mathbf{y}, \hat{\mathbf{y}})$ function to indicate how "different" clusterings are from each other, as well as come up with an appropriate $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ feature function to relate input sets $\mathbf{x}$ and output clusterings $\mathbf{y}$ in such a fashion that the $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ is equivalent to the $\mathbf{w}$-parameterized correlation clustering objective function $f_{\mathbf{w}}(\mathbf{x}, \mathbf{y})$.

$\Delta(\mathbf{y}, \hat{\mathbf{y}})$ indicates a non-negative real-valued loss between a true cluster $\mathbf{y}$ and a predicted cluster $\hat{\mathbf{y}}$. $\Delta(\mathbf{y}, \hat{\mathbf{y}}) = 0$ if $\mathbf{y} = \hat{\mathbf{y}}$, and Δ rises as the two clusters become more dissimilar. In our experimental section we use two loss functions Δ: a loss based on the MITRE precision and recall score for noun-phrase coreference, and a "pairwise" loss that counts the number of pairwise cluster relationships the clusterings disagree on. More details of these loss functions appear in Section 3.3.

We must also phrase our $\mathbf{w}$-parameterized discriminant function $f_{\mathbf{w}}(\mathbf{x}, \mathbf{y})$ of (3.1) as

$$f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle \tag{3.4}$$

as shown in Section 2.1. We can rewrite the correlation clustering objective func-

tion $f(\mathbf{x}, \mathbf{y})$ of (3.1) as follows:

$$f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \sum_{y \in \mathbf{y}} \sum_{x_i, x_j \in y} K_{\mathbf{w}}(i, j) \tag{3.5}$$

$$= \sum_{y \in \mathbf{y}} \sum_{x_i, x_j \in y} \langle \mathbf{w}, \psi(x_i, x_j) \rangle \tag{3.6}$$

$$= \left\langle \mathbf{w}, \left(\sum_{y \in \mathbf{y}} \sum_{x_i, x_j \in y} \psi(x_i, x_j) \right) \right\rangle. \tag{3.7}$$

The objective function is an inner product of our parameterization $\mathbf{w}$, and a sum of ψ vectors. So, if we begin working from (3.7), we can derive the $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ combined feature function of the input $\mathbf{x}$ and output $\mathbf{y}$ as

$$\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \frac{1}{|\mathbf{x}|^2} \sum_{y \in \mathbf{y}} \sum_{x_i, x_j \in y} \psi(x_i, x_j) \tag{3.8}$$

Since $f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ is the correlation clustering objective, we may phrase a $\mathbf{w}$-parameterized correlation clustering for a set of items $\mathbf{x}$ as follows, with $\mathbf{\Psi}$ taking the form given in (3.8):

$$h_{\mathbf{w}}(\mathbf{x}) = \operatorname*{argmax}_{\mathbf{y}} f_{\mathbf{w}}(\mathbf{x}, \mathbf{y}) = \operatorname*{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle \tag{3.9}$$

In terms of the structural SVM formulations in terms of OP 4, by phrasing correlation clustering in this fashion we may apply the program to find a $\mathbf{w}$ such that, for every training example $(\mathbf{x}_i, \mathbf{y}_i)$, and every possible wrong clustering $\mathbf{y}$, we will have SVM-CC find the vector $\mathbf{w}$ to make the value of the objective for the *correct* clustering be greater than the value of the objective for this *incorrect* clustering by at least a margin of the loss between $\mathbf{y}_i$ and $\mathbf{y}$. Note that $\sum_{i=1}^{n} \xi_i$ upper bounds the training loss.

Of course, merely specifying the correlation clustering objective and a loss function is insufficient for the practical application of either OP 4 or OP 5. These problems themselves are still intractable given the large number of constraints they

entail: they require as many constraints for a given example $(\mathbf{x}_i, \mathbf{y}_i)$ as there are different partitions of the set $\mathbf{x}_i$, which itself equals the $|\mathbf{x}_i|^{\text{th}}$ Bell number [89]. The implementation of Algorithm 1 instead applies the cutting plane algorithm of Algorithm 1 by finding some $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y}} H(\mathbf{y})$ which is the maximization of the function $H(\mathbf{y})$ given by (2.21) (if trying to solve OP 4) or (2.22) (if trying to solve OP 5). For the case of margin scaling, recall that (2.21) is

$$H(\mathbf{y}) \equiv \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}) \tag{3.10}$$

By solving $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y}} H(\mathbf{y})$, the algorithm finds the clustering $\hat{\mathbf{y}}$ associated with the most violated constraint for $(\mathbf{x}_i, \mathbf{y}_i)$, i.e., the output that requires the greatest slack. Since $H(\hat{\mathbf{y}})$ is the necessary slack for $\hat{\mathbf{y}}$ under the current $\mathbf{w}$, if $H(\hat{\mathbf{y}}) > \xi_i + \epsilon$, the constraint is violated by more than ϵ, so we introduce the constraint and re-optimize. The algorithm repeats this process until no new constraints are introduced. The proof of convergence and correctness of this algorithm appeared in Section 2.1.3.

3.3 Loss Functions

Many learning tasks already have existing performance measures. For example, performance on noun-phrase coreference is often evaluated with the MITRE score [108]. While many learning methods optimize to some implicit performance measure, good performance on this learning measure may not translate into good performance on the desired measure. In this section we test whether SVM-CC's ability to optimize to a particular loss function is beneficial. We use SVM-CC with two loss functions, described in the sequel.

3.3.1 Pairwise Loss Δ_P

Δ_P (Pairwise Loss) is

$$\Delta_P(\mathbf{y}, \bar{\mathbf{y}}) = 100\frac{W}{T}, \tag{3.11}$$

where T is the total number of pairs of items in the set $\mathbf{x}$ which is being partitioned by $\mathbf{y}$ and $\bar{\mathbf{y}}$, i.e., $T = \binom{|\mathbf{x}|}{2}$. W is the total number of pairs where $\mathbf{y}$ and $\bar{\mathbf{y}}$ disagree about their cluster membership. This loss is scaled from 0 to 100, where $\Delta_P(\mathbf{y}, \bar{\mathbf{y}}) = 0$ indicates that all the pairwise relationships are equal, that is, $\mathbf{y} = \bar{y}$, and $\Delta_P(\mathbf{y}, \bar{\mathbf{y}}) = 100$ indicates that all pairwise relationships are flipped, which is not actually possible except in cases where $\mathbf{y}$ and $\bar{\mathbf{y}}$ are the clusters where every item is in its own cluster, or where all items are in one cluster (or vice versa). This is the complement of the Rand index [87].

3.3.2 MITRE Loss Δ_M

Δ_M (MITRE Loss) is $\Delta_M(\mathbf{y}, \bar{\mathbf{y}}) = 100\frac{2RP}{R+P}$ where R and P are the MITRE recall and precision scores respectively [108]. Though this measure is difficult to describe succinctly and accurately, the MITRE measures R and P can be briefly summarized and understood in terms of the number of operations to transform $\mathbf{y}$ into $\bar{\mathbf{y}}$. Suppose we consider two operations: merge two clusters in a given clustering $\mathbf{y}$ to form one cluster, or split one cluster in $\mathbf{y}$ to form two clusters. The compliment of recall $1 - R$ and precision $1 - P$ are proportional to how many merges and splits are needed to transform $\bar{\mathbf{y}}$ into $\mathbf{y}$.

Since merges are the inverse of splits, R and P get flipped if we flip which of $\mathbf{y}$ or $\bar{\mathbf{y}}$ we consider as being the "correct" clustering; which is to say, recall of $\mathbf{y}$ with respect to $\bar{\mathbf{y}}$ is precision of $\bar{\mathbf{y}}$ with respect to $\mathbf{y}$. Because these functions R and

P can be considered mathematically equivalent with just an inversion of function arguments, we focus on describing recall of $\bar{\mathbf{y}}$ with respect to truth $\mathbf{y}$, $R(\mathbf{y}, \bar{\mathbf{y}})$.

Though we shall provide a formal expression for $R(\mathbf{y}, \bar{\mathbf{y}})$ in (3.13) in the sequel, it is not obvious where the expression comes from, so some intuitive understanding prior to the expressions presentation is desirable. The expression can be viewed in terms of the number of "joins" we would need to do on the clusters within $\bar{\mathbf{y}}$ to get it into form like $\mathbf{y}$. To explain further, for each cluster $y \in \mathbf{y}$, we count how many clusters in $\bar{\mathbf{y}}$ the elements in y are scattered across. If y's elements appear within $C = |\{\bar{y} \in \bar{\mathbf{y}} : \bar{y} \cap y \neq \emptyset\}|$ clusters in $\bar{\mathbf{y}}$, then $C - 1$ joins would be necessary to make these elements into one cohesive cluster. By summing over all $y \in \mathbf{y}$, we get the number of joins required for all of $\bar{\mathbf{y}}$. (Viewing this really in terms of minimum required operations, by doing joins of two clusters in $\bar{\mathbf{y}}$ we might render another join redundant, but we do not concern ourselves with it: imagine that all the "splits" of clusters in $\bar{\mathbf{y}}$ happened first.) Of course, recall is not the measure of the joins that need to happen, but rather the joins that did happen that should have. In total, to build $\mathbf{y}$, we would need the number of elements in $\mathbf{y}$'s partitions minus the number of partitions, e.g., $\sum_{y \in \mathbf{y}} (|y| - 1)$.

$$R(\mathbf{y}, \bar{\mathbf{y}}) = \frac{\sum_{y \in \mathbf{y}} (|y| - 1) - \sum_{y \in \mathbf{y}} \left(|\{\bar{y} \in \bar{\mathbf{y}} : \bar{y} \cap y \neq \emptyset\}| - 1 \right)}{\sum_{y \in \mathbf{y}} (|y| - 1)} \tag{3.12}$$

$$= \frac{\sum_{y \in \mathbf{y}} (|y| - |\{\bar{y} \in \bar{\mathbf{y}} : \bar{y} \cap y \neq \emptyset\}|)}{\sum_{y \in \mathbf{y}} (|y| - 1)} \tag{3.13}$$

In the case where the denominator of (3.13) is 0, which happens when there are as many clusters in $\mathbf{y}$ as there are items being clustered, then $R(\mathbf{y}, \bar{\mathbf{y}}) = 1$ for any $\bar{\mathbf{y}}$ since, intuitively, if all items should be in their own cluster, then no joins need to happen at all, so every join that should happen must have all ready have happened, regardless of what $\bar{\mathbf{y}}$ is.

Precision $P(\mathbf{y}, \bar{\mathbf{y}}) = R(\bar{\mathbf{y}}, \mathbf{y})$. The final MITRE loss Δ_m is

$$\Delta_M(\mathbf{y}, \bar{\mathbf{y}}) = \frac{2P(\mathbf{y}, \bar{\mathbf{y}})R(\mathbf{y}, \bar{\mathbf{y}})}{P(\mathbf{y}, \bar{\mathbf{y}}) + R(\mathbf{y}, \bar{\mathbf{y}})}, \tag{3.14}$$

which is the harmonic mean between recall and precision.

3.4 Approximate Inference for the Separation Oracle

In this section we describe the difficulty of finding the most violated constraint in $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ and suggest methods for approximately finding the most violated constraint with two clustering methods.

Consider the cost function H for loss margin scaling (2.21) as in OP 4.

$$H(\mathbf{y}) \equiv \Delta(\mathbf{y}_i, \mathbf{y}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle \tag{3.15}$$

The last term is a constant, and so can be ignored since it does not change the maximum. The cost function is a loss Δ between the true labeling $\mathbf{y}_i$ and prediction $\mathbf{y}$ plus the correlation clustering objective function. Finding the $\mathbf{y}$ to maximize the correlation clustering objective function is NP-complete [7], and the addition of the loss term is unlikely to help tractability, so finding $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ is intractable, just as the basic inference problem is. Fortunately, algorithms exist for approximately maximizing these clustering objectives, and $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$. These approximations will not solve $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ exactly, but are possibly close enough that SVM-CC still learns something reasonable. Applying a similar margin maximizing framework to perform collective classifications, [103] inferred approximated constraints with a linear relaxation. Approximate inference may work for clustering as well.

However, recall that we had our theoretical guarantees of Section 2.1.3, which ensure that Algorithm 1 terminates in reasonable times and with a solution which

is close to the solution we desire in OP 4 or OP 5. How are the termination and the correctness of the structural SVM algorithm affected if one uses approximate maximization of $H(\mathbf{y})$? The proof of polynomial time termination in Theorem 4 still holds. The proof does not depend upon finding $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ exactly, but rather that new introduced constraints are violated by more than ϵ, and so cause the quadratic objective to increase by a minimum amount. The proof of correctness for Theorem 2 no longer holds. Without finding $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ exactly, either violated constraints may remain undetected, or the objective may be raised. We consider two approximations: a simple greedy approach $\mathcal{C}_G$ corresponding to Algorithm 3, and a real relaxation of correlation clustering $\mathcal{C}_R$ corresponding to OP 10. We consider how they impact the correctness of the algorithm in the sequel, and later in Section 3.6 empirically evaluate their performance. Later, in Chapter 5, we treat the problem of using these types of approximation in much greater detail.

3.4.1 Greedy Approximation to Clustering, $\mathcal{C}_G$

To greedily approximate $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$, we can adapt Algorithm 3, with the simple modification that the merges that take place are those that most increase the cost function, not those that most increase the objective function. The advantage of this algorithm is that it can incorporate any loss Δ at all, although some Δ functions may be of such a form that the maximization $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ is not approximated well by such a simple greedy search.

Fairly formal pseudocode for this algorithm is given in Algorithm 4, but its workings are intuitively easy to understand: Start with an initial partitioning $\mathbf{y}$ with every item of $\mathbf{x}$ in its own cluster. Repeatedly find and merge the two clusters $y_i, y_j \in \mathbf{y}$ that would maximally increase $H(\mathbf{y})$. Halt and return $\mathbf{y}$ when no merge

(Greedy Correlation Clustering Separation Oracle)

Input: An input example $(\mathbf{x}_i, \mathbf{y}_i)$, current model parameterization $\mathbf{w}$.
$\hat{\mathbf{y}} \leftarrow \{\{x_j\} : x_j \in \mathbf{x}_i\}$
let $H(\mathbf{y}) \equiv \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y})$ for margin scaling (OP 4)
let $H(\mathbf{y}) \equiv (\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle + 1)\, \Delta(\mathbf{y}_i, \mathbf{y})$ for slack scaling (OP 5)
let $\textsc{Merge}(\mathbf{y}, y, y') \equiv (\mathbf{y} \setminus \{y, y'\}) \cup \{y \cup y'\}$
repeat
  $\bar{y}, \bar{y}' \leftarrow \text{argmax}_{y,y' \in \hat{\mathbf{y}}} H(\textsc{Merge}(\hat{\mathbf{y}}, y, y'))$
  if $H(\textsc{Merge}(\hat{\mathbf{y}}, y, y')) > H(\hat{\mathbf{y}})$ **then**
    $\hat{\mathbf{y}} \leftarrow \textsc{Merge}(\hat{\mathbf{y}}, y_i, y_j)$
  end if
until $\hat{\mathbf{y}}$ has not changed during an iteration, or $|\hat{\mathbf{y}}| = 1$
return $\hat{\mathbf{y}}$

Algorithm 4: The greedy approximation to the correlation clustering separation oracle $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ for example $(\mathbf{x}_i, \mathbf{y}_i)$.

increases $H(\mathbf{y})$.

Given the close relation between Algorithm 3 and Algorithm 4, we refer to both with the shorthand $\mathcal{C}_G$, with the understanding that when we are talking about training a model we are referring to the separation oracle variant Algorithm 4, and when referring to prediction we are talking about the prediction variant Algorithm 3.

Corollary 1. *The greedy approximation $\mathcal{C}_G$ leads to an underconstrained program with respect to OP 4, with an objective value not greater than OP 4's objective.*

Proof. Suppose the true $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ is $\hat{\mathbf{y}}$, but the approximate $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ found with this greedy approximation is $\mathbf{y}^*$, so that $H(\hat{\mathbf{y}}) \geq H(\mathbf{y}^*)$. Some constraints from the full QP OP 4 violated by more than ϵ might not be found and introduced. This leads to an optimization program which is underconstrained relative to OP 4, i.e., the solution found may be infeasible by more than the ϵ tolerance.

Since the underconstrained program's feasible region contains the solution to OP 4, the objective cannot be greater than OP 4's objective. □

3.4.2 Relaxation Approximation to Clustering, $\mathcal{C}_R$

The relaxation of OP 10 can also be used as a separation oracle in the case where our loss function is the pairwise loss Δ_P, and we are using the margin scaled structural SVM learning framework of OP 4. For a given training example $(\mathbf{x}, \mathbf{y})$, the linear program that serves as a "relaxed" separation oracle for Δ_P under the margin scaled structural SVM producing a relaxed most violated constraint $\mathbf{e}$ is

Optimization Problem 11. (Relaxed Correlation Clustering)

$$\max_{\mathbf{e}} \sum_{e_{a,b} \in \mathbf{e}} e_{a,b} \cdot \left(\langle \mathbf{w}, \psi_{a,b} \rangle + (1 - 2 \cdot \mathbb{1}_{\exists y \in \mathbf{y}. x_a \in y \wedge x_b \in y}) \cdot \frac{100}{\binom{|\mathbf{x}|}{2}} \right) \tag{3.16}$$

$$s.t. \quad e_{a,b} \in [0,1], \qquad e_{a,b} = e_{b,a}, \qquad e_{a,b} \geq e_{b,c} + e_{a,c} - 1 \tag{3.17}$$

The $\mathbb{1}_{\cdot}$ indicator function tests the indicated condition, which in the case of its use in (3.16) is a test for the existence of a cluster $y \in \mathbf{y}$ holding x_a and x_b, i.e., a test that the two items should be in the same cluster according to $\mathbf{y}$. The second term of which this indicator function is a part represents the loss incurred by (or avoided by) setting $e_{a,b}$ to a non-zero value.

However, $\mathbf{\Psi}$ and Δ are defined for discrete clusterings $\mathbf{y}$, not relaxed clusterings $\mathbf{e}$. We can use the relaxed solution in the constraints by extending (3.8) to incorporate a relaxed solution $\mathbf{e}$ instead of the discrete solution $\mathbf{y}$.

$$\mathbf{\Psi}(\mathbf{x}, \mathbf{e}) = \frac{1}{|\mathbf{x}|^2} \sum_{e_{i,j} \in \mathbf{e}} e_{i,j} \cdot \psi(x_i, x_j) \tag{3.18}$$

Note that (3.18) is equivalent to (3.8) if all $e_{i,j}$ are integral. We can further extend the Δ_P to accommodate relaxed clusterings through

$$\Delta_P(\mathbf{y}, \mathbf{e}) = 100 \frac{\sum_{e_{i,j} \in \mathbf{e}} \left| \mathbb{1}_{\exists y \in \mathbf{y}. x_i \in y \wedge x_j \in y} - e_{i,j} \right|}{\sum_{e_{i,j} \in \mathbf{e}} 1}, \tag{3.19}$$

which is essentially the average percentage over all $e_{i,j}$ variables of how far off from the true clustering $\mathbf{y}$ they are. In the case where all $e_{i,j} \in \{0, 1\}$, (3.19) is equivalent to (3.11).

One interesting characteristic of this separation oracle is that it finds the most violated constraint, but over an expanded space that admits fractional solutions to the correlation clustering procedure, with the original search space $\mathcal{Y}$ as a subset of this expanded space. Though we shall explore the implications of relaxations as separation oracles more fully in Section 5.3.2, for now we present this result.

Corollary 2. *The relaxed approximation $\mathcal{C}_R$ leads to an overconstrained program with respect to QP OP 4, with an objective not less than than OP 4's objective.*

Proof. The feasible region of the LP relaxation contains the integer solution to $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$. This means the relaxed solution $\mathbf{e}$ forms an upper bound on $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$, i.e., $H(\hat{\mathbf{y}}) \leq H(\mathbf{e})$. If $\hat{\mathbf{y}}$'s corresponding constraint would be introduced, $\mathbf{e}$'s corresponding constraint must also be introduced. So, at the end of the iterations no constraint in the QP OP 4 is significantly violated, and as additional constraints not in the OP 4 may have been introduced, the QP is potentially overconstrained with respect to OP 4. Since the extra constraints may exclude OP 4's solution from the feasible region, the objective cannot be less than QP OP 4's objective. □

The incorporation of Δ_P into the linear objective of the predictor OP 10 to derive the separation oracle OP 11 was just a matter of incrementing and decre-

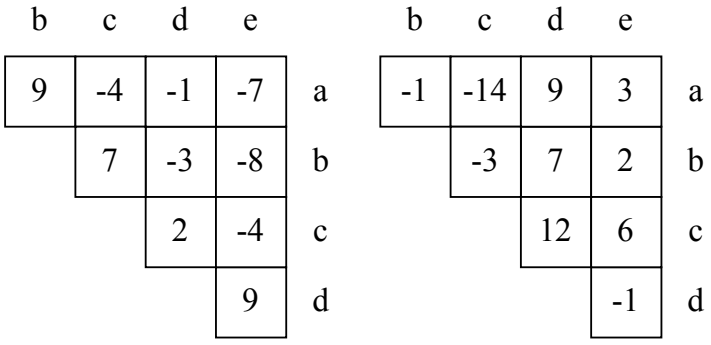

Figure 3.2: Correlation clustering on a matrix of similarities for the item set $\mathbf{x}_i = \{x_a, x_b, x_c, x_d, x_e\}$ with clustering $\mathbf{y}_i = \{\{x_a, x_b, x_c\}, \{x_d, x_e\}\}$. The left matrix holds the raw similarities as would be used in computing $\text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle$, whereas the right matrix holds the adjusted simiarities that would be used in computing the $\mathbf{y}$ corresponding to the most violated constraint, $\text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta_P(\mathbf{y}_i, \mathbf{y})$.

menting pairwise similarity scores depending on whether a given pair should not or should be in the same cluster, respectively. Since Δ_P is decomposable in the same fashion $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ is, any technique used for correlation clustering can be used to approximate $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ for margin scaling under Δ_P. More specifically, if we consider correlation clustering as an operation over a similarity matrix K inferred from an item set $\mathbf{x}$, Δ_P is incorporated by incrementing (or decrementing) elements of K by the amount of loss caused (or avoided) by joining the corresponding pair in a clustering. OP 11 is simply OP 10 adjusted to incorporate this incrementing and decrementing.

In Figure 3.2 we see an example of this adjustment for a set of five items $\mathbf{x}_i = \{x_a, x_b, x_c, x_d, x_e\}$ with true clustering $\mathbf{y}_i = \{\{x_a, x_b, x_c\}, \{x_d, x_e\}\}$. In this case the pairwise scores are adjusted downward or upward $\frac{100}{\binom{5}{2}} = \frac{100}{10} = 10$, since there are 10 pairwise relationships, i.e., $\binom{5}{2}$, among five items, so each adjustment is either 10 (for pairs not in the same cluster in $\mathbf{y}_i$) or -10 (for pairs in the same

cluster in $\mathbf{y}_i$). With the pairwise scores thus modified, the clustering procedure is run as normal.

While incorporating Δ_P is straightforward, the MITRE loss Δ_M is another story since it is not linearizable in the same fashion that Δ_P is, so incorporating it into the linear program of OP 10 is impossible while maintaining it as a linear program. For this reason, we do not use the relaxed separation oracle $\mathcal{C}_R$ when optimizing for MITRE loss Δ_M.

Similar to the greedy approximation, given the close relation between OP 10 and OP 11, we refer to both with the shorthand $\mathcal{C}_R$, with the understanding that in the context of training or prediction we are referring to the separation oracle or prediction variant, respectively.

3.4.3 Discretized Relaxation to Clustering, $\mathcal{C}_R^*$

For evaluation on the test set, we employ $\mathcal{C}_R^*$, a discretized version of $\mathcal{C}_R$. $\mathcal{C}_R^*$ forces a relaxed solution $\mathbf{e}$ into discrete clusters with a simple technique: Start with an initial partitioning $\mathbf{y}$ that has every item in $\mathbf{x}$ in its own cluster. Iterate over all $x_a \in \mathbf{x}$. If x_a is currently in a singleton cluster in $\mathbf{y}$, iterate through all other $x_b \in \mathbf{x}$, put x_a in x_b's cluster for the first x_b that satisfies $e_{a,b} > 0.7$. Algorithm 5 provides pseudocode for this procedure.

This discretization procedure is very simple compared to others in the literature [33], but in actual practice, the correlation clustering linear program rarely produces non-integer solutions. In fact, in our experiments, at no time was there a non-integer solution to the linear program during prediction with a learned model, so even the simplest discretization procedure would have sufficed. Had this not

(Discretized Relaxation Correlation Clustering, $\mathcal{C}_R^*$)

Input: An input set of items $\mathbf{x}$
$\mathbf{e} \leftarrow$ the solution to OP 10
$\mathbf{y} \leftarrow \{\{x_i\} : x_i \in \mathbf{x}\}$
let $\textsc{Merge}(\mathbf{y}, y, y') \equiv (\mathbf{y} \setminus \{y, y'\}) \cup \{y \cup y'\}$
let $\textsc{FindCluster}(\mathbf{y}, x) \equiv y \in \mathbf{y}$ such that $x \in y$
for $x \in \mathbf{x}$ **do**
    if $|\textsc{FindCluster}(\mathbf{y}, x)| > 1$, continue
    find any $x' \in \mathbf{x} - x$ such that $e_{x,x'} > 0.7$
    if no such x' exists, continue
    $\mathbf{y} \leftarrow \textsc{Merge}(\mathbf{y}, \{x\}, \textsc{FindCluster}(\mathbf{y}, x'))$
end for
return $\mathbf{y}$

Algorithm 5: The discretization procedure $\mathcal{C}_R^*$.

been the case, we would have been motivated to do something less simple.

3.5 Training Algorithm

Algorithm 6 gives the training algorithm for learning the parameterization $\mathbf{w}$ of correlation clustering given the training sample $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$ with regularization parameter C and tolerance ϵ. This algorithm is Algorithm 1, instantiated with correlation clustering as described in the previous matter. The symbol $\mathbf{ye}$ is used to indicate either a discrete clustering $\mathbf{y}$ or a soft clustering $\mathbf{e}$ as returned by the relaxation $\mathcal{C}_R$. The clustering algorithms $\mathcal{C}_G$ and $\mathcal{C}_R$ are augmented to simultaneously maximize the appropriate loss function Δ as shown in (3.10) and described algorithmically in Section 3.3.

(Supervised Correlation Clustering Algorithm)

Input: $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, C, ϵ
$S_i \leftarrow \emptyset$ for all $i = 1, \ldots, n$
repeat
  for $i = 1, \ldots, n$ **do**
    if learning using relaxations **then**
      $\hat{\mathbf{e}} \leftarrow$ output of OP 11 given $(\mathbf{x}_i, \mathbf{y}_i)$
    else
      $\hat{\mathbf{y}} \leftarrow$ output of Algorithm 4 given $(\mathbf{x}_i, \mathbf{y}_i)$
    end if
    $\hat{\mathbf{ye}} \leftarrow \hat{\mathbf{y}}$ or $\hat{\mathbf{e}}$ as appropriate
    compute $\xi_i = \max\{0, \max_{\mathbf{ye} \in S_i} \Delta(\mathbf{y}_i, \mathbf{ye}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{ye}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle\}$
    if $\Delta(\mathbf{y}_i, \hat{\mathbf{ye}}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \hat{\mathbf{ye}}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle > \xi_i + \epsilon$ **then**
      $S_i \leftarrow S_i \cup \{\hat{\mathbf{ye}}\}$
      $\mathbf{w} \leftarrow$ optimize primal over $\bigcup_i S_i$
    end if
  end for
until no S_i has changed during iteration

Algorithm 6: Cutting plane algorithm for the supervised correlation clustering problem, based on the cutting plane algorithm for structural SVMs Algorithm 1.

3.6 Empirical Analysis

This section describes experiments to test the ability of SVM-CC to exploit dependencies in data, to examine the importance of the loss function during optimization, and to examine the different approximations to $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$. We evaluate SVM-CC's performance on noun-phrase clustering and news article clustering.

3.6.1 Datasets

For the MUC-6 noun-phrase coreference task, there are 60 documents with their noun-phrases assigned to coreferent clusters. Each document had an average of

101 clusters, with an average of 1.48 noun-phrases per cluster; there are many single element clusters. The first 30 documents form the training set. The last 30 form the evaluation set. The pairwise feature vectors for pairs of noun-phrases are those used in [78]. Each feature vector contains 53 features, e.g., whether the noun-phrases appear to have the same gender, how many sentences apart they are, whether either one is the subject in a sentence, etc.

The news article clustering data set is a new data set we derived by trawling Google News. Google News *itself* works by clustering news articles, but presumably their clustering method is sufficiently sophisticated that teaching an unsophisticated clustering method how to cluster in the same fashion is interesting. For each day for 30 days (starting July 14 2004 through August 12 2004), at most 10 topics from the "World" category were selected, and from each topic at most 15 articles were selected. The topics form our true reference clusters. The first 15 days are the training set, and the last 15 days are the test set.

We have various simple heuristics for extracting the article text, quoted article text, headline, and title. These extraction procedures were hand coded and far from perfect, but seemed to work well on the majority of the data. Given that the text was extracted in 2004, which is long past the age where the "real text" of a page and its textual formatting can be more or less reliably determined just from the HTML, the basis of the procedure was to render the indicated web page in a virtual web browser and pick out the text which appeared to conform to certain formatting with respect to darkness, size, and placement on the page, which differed depending upon which type of element we were attempting to extract (e.g., page title, headline, text, quoted text). The extraction procedure is highly noisy, however, and a significant portion of the entries were badly extracted (e.g.,

extraction of unrelated side text, banners, menu text, advertisement text, etc.).

Each article has 30 TFIDF weighted vectors for unigrams, bigrams, and trigrams of the text appearing in the title, the headline according to two extraction methods, article text, and article text in quotations, and for all of these there are Porter stemmed and non-stemmed versions of the vectors. The pairwise feature vector $\psi_{a,b}$ for two articles $x_a, x_b \in \mathbf{x}$ are the 30 cosine similarities between these entities corresponding vectors in x_a and x_b, plus one feature which is always the constant 1. For example, feature 11 is the cosine similarity among TFIDF bigrams in unstemmed text.

3.6.2 Experimental Setup

With these data sets, we trained and tested several supervised correlation clustering models. A model consists of the learned similarity weights $\mathbf{w}$. In all cases, the C regularization parameter was chosen from several values based on k-fold cross validation on the training set ($k = 10$ for NP-coreference, $k = 5$ for news article clustering). Significance tests between the results for two models use the paired two-tailed T-test. Performance is considered significantly different for p values less than 0.05.

For our baseline, we use PCC (pairwise classification correlation clustering), the naïve approach described and critiqued in Section 1.3. In summary, to learn a similarity measure for clustering, this PCC method will take all pairs of items in all training sets, take each pairwise feature vector as an input vector, and let positive examples for this classification learning algorithm be those pairwise vectors in the same cluster, and negative examples be those pairwise vectors in different clusters.

Table 3.1: Results for NP Coreference, with columns corresponding to different constraint inference methods used in training, and rows corresponding to different loss functions used in testing.

	$\mathcal{C}_G$	PCC	Default
Test with $\mathcal{C}_G, \Delta_M$	41.3	51.6	51.0
Test with $\mathcal{C}_G, \Delta_P$	2.89	3.15	3.59

With such a model learned, when you want to cluster a new set of items, one would simply run all pairs in this new set through the learned classifier. The output values are the pairwise similarity values. Positive and negative outputs indicate a pair should or should not be in the same cluster, respectively. Then, cluster based on these output similarities. PCC uses SVM^{light} as the pairwise classifier, and clusters with correlation clustering.

3.6.3 Supervised Correlation Clustering vs. the Pairwise Learner

Section 1.3 outlines problems with a method like PCC. We supposed SVM-CC would be able to handle transitive dependencies better than a simple pairwise classifier. How does SVM-CC compare to PCC?

Table 3.1 shows a comparison on the noun-phrase task. The $\mathcal{C}_G$ column contains results of two models trained on SVM-CC using the greedy $\mathcal{C}_G$ approximation, with the first optimized and tested with respect to the MITRE loss Δ_M, the second with respect to the pairwise loss Δ_P. Both tests used greedy $\mathcal{C}_G$ clustering on the test set with the learned similarity measure. The PCC column contains analogous results

Table 3.2: Results for News Articles, with columns corresponding to different constraint inference methods used in training, and rows corresponding to different loss functions used in testing.

	$\mathcal{C}_G$	$\mathcal{C}_R$	PCC	Default
Test with $\mathcal{C}_G, \Delta_P$	2.36	2.43	2.45	9.45
Test with $\mathcal{C}_R^*, \Delta_P$	2.04	2.08	1.96	9.45

for PCC. The default column contains results for a model that either puts each item in its own cluster (for Δ_P), or all in one cluster (for Δ_M).

The SVM-CC model performs significantly better. While the Δ_M performance could be explained as optimization to a loss which PCC cannot do, the Δ_P loss, as the proportion of pairwise relationships that are wrong, is analogous to pairwise accuracy, which is what PCC's classifier optimizes. Even under this configuration, SVM-CC performs significantly better.

What happens for item sets without complex transitive dependencies between items? Consider the case where you view two noun-phrases in isolation, versus two news articles in isolation. While it is often very difficult to tell whether two noun-phrases co-refer by just looking at two noun-phrases taken out of context, it is usually quite easy to tell if two news articles are about the same topic just by viewing the two articles. For this reason, it seems less helpful to exploit dependencies in a task like news article clustering. In Table 3.2, we see the results of a comparison between SVM-CC and PCC. The $\mathcal{C}_G$ and $\mathcal{C}_R$ columns refer to the clusterers used in the cost function approximation in SVM-CC. The two rows show the performance of the learned similarity measure with different clustering methods. Though results seem mixed, the results among the different methods in each row

are not statistically different from one another. These empirical results suggest that SVM-CC is more effective than the nave PCC approach when the data contains transitive dependencies, and that both methods perform comparably when not.

3.6.4 Effects of Optimizing to the Correct Loss

The SVM-CC algorithm has the ability to optimize to specific loss functions. How important is it to use the correct loss function during training? We address this question in an experiment that evaluates how a model optimized for one loss function performs when evaluated under a different loss function.

Table 3.3 shows evaluation results on the NP-task for models optimized to different losses (corresponding to columns) and evaluated on different losses (corresponding to rows). The performances in the first row for the MITRE loss Δ_M are not significantly different for models optimized to Δ_M and Δ_P. Interestingly, when optimized under the pairwise loss Δ_P, there is a great difference; indeed, models optimized to Δ_M are not even significantly different from the default clustering shown in Table 3.1. We conclude that optimization to the appropriate loss function can make a significant and substantial difference in clustering accuracy.

3.6.5 Importance of Loss in the Separation Oracle

The cost function H includes a loss function, but when computing $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$, sometimes including the loss function is difficult or impossible for computational reasons, e.g., including the MITRE score in the linear objective for correlation clus-

Table 3.3: Training and testing on separate losses on the noun-phrase coreference task. Columns represent the particular Δ function that was optimized during training of the model in question, while rows represent the Δ used in evaluation.

	Opt. to Δ_M	Opt. to Δ_P
Performance on Δ_M	41.3	42.8
Performance on Δ_P	4.06	2.89

Table 3.4: Comparison of performance when loss was not used in the $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$, versus when it was included. NP-coreference experiments used $\mathcal{C}_G$ clustering. News experiments used Δ_P loss.

	w/ loss	w/o loss
NP-coreference, Δ_M	41.3	41.1
NP-coreference, Δ_P	2.89	2.81
News, train $\mathcal{C}_G$, test $\mathcal{C}_G$	2.36	2.42
News, train $\mathcal{C}_R$, test $\mathcal{C}_R^*$	2.08	2.16

tering. Can we sometimes get away with not including the loss in the $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$? Note, we do still include the loss when introducing a new QP constraints; however, the method to choose which constraint to introduce would no longer necessarily find the best constraint.

A comparison of SVM-CC models that differed only in whether the loss is or is not included in the cost function is seen in Table 3.4. No two results in a row of this table differ significantly. This bodes well for situations where including the cost in the $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ approximation is difficult.

Table 3.5: Comparison of performance on the news dataset when different clustering methods were used to approximate $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$.

	Train $\mathcal{C}_G$	Train $\mathcal{C}_R$
Test $\mathcal{C}_G$	2.36	2.43
Test $\mathcal{C}_R^*$	2.04	2.08

3.6.6 Greedy vs. Relaxed Clustering in Training

For clustering, finding the exact $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ present in Line 6 of the algorithm requires solving an NP-hard problem, so we instead use greedy and relaxation approximations. How do these approximations compare?

The different clustering methods $\mathcal{C}_G$ and $\mathcal{C}_R$ are used in training models for the news article task. In Table 3.5 we compare models that differ only in which approximation was used during training. The test results are not significantly different. This comparison was run only on the news story task: the off-the-shelf linear solver used in the correlation clustering implementation could not handle some problem sizes in the noun-phrase MUC-6 task. These results provide no basis to prefer either the greedy underconstrained approximation or relaxation overconstrained approximation.

3.6.7 Discussion of the Model's Learned Weights

Aside from comparative end performance figures, which provide a macroscopic view of the algorithm, it is also of interest to examine in greater detail the workings of the supervised correlation clustering procedure on a dataset. In particular, we shall

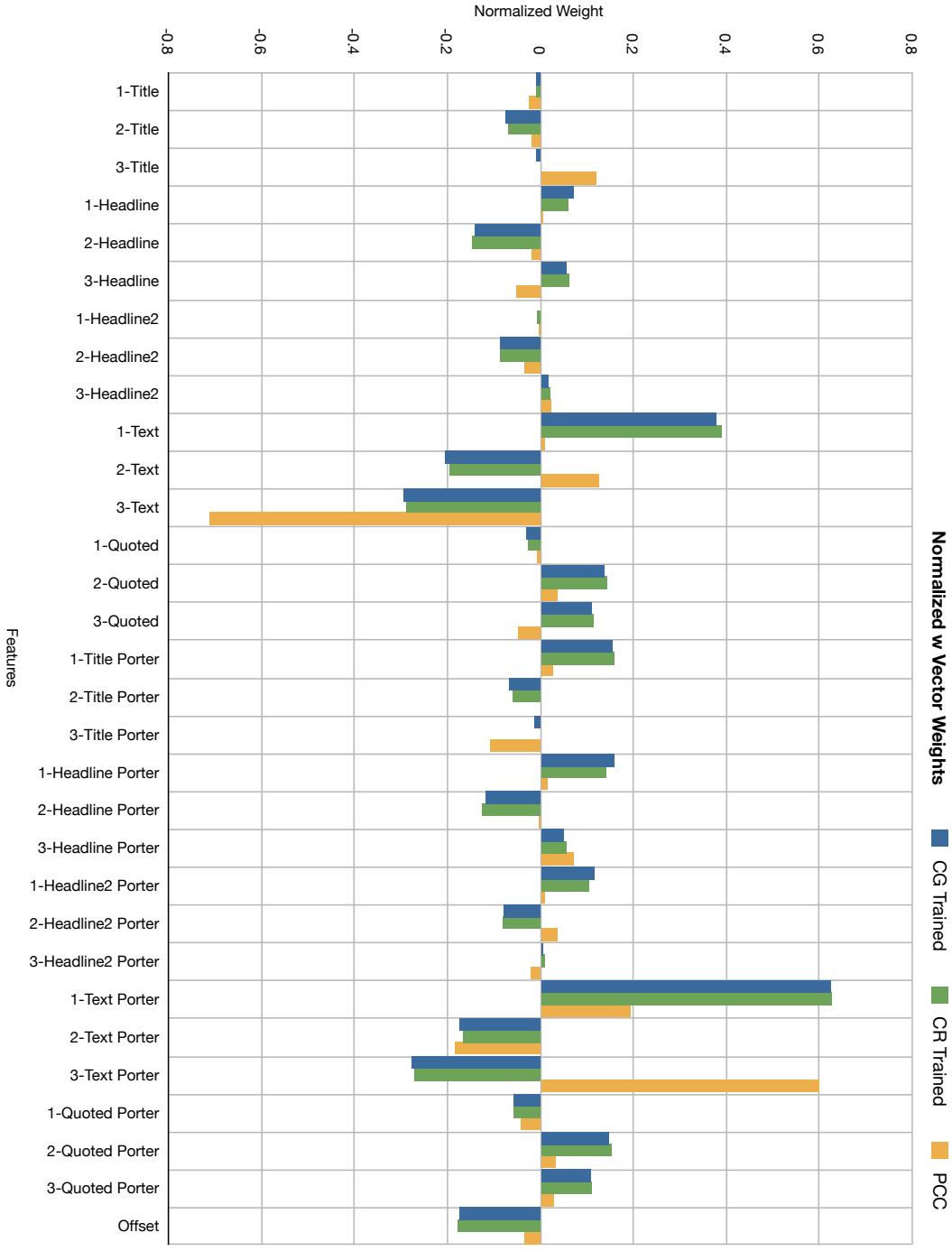

Figure 3.3: For the final models selected through cross validation trained through either $\mathcal{C}_G$, $\mathcal{C}_R$, or PCC, this presents a plot of the learned weights.

examine the weights learned by these supervised correlation clustering procedures on a dataset.

These supervised correlation clustering procedures all provide a learned weight vector $\mathbf{w}$ given a training sample $\mathcal{S}$, with the view that future sets of items $\mathbf{x}$ will have pairwise similarities between items $x_a, x_b \in \mathbf{x}$ take the form of $\langle \mathbf{w}, \psi_{a,b} \rangle$, where $\psi_{a,b}$ is the pairwise feature vector between x_a and x_b. These pairwise similarities are then used in the correlation clustering procedure to produce the output clustering $\mathbf{y} = h(\mathbf{x})$, as described in Section 3.1. So, we can see in the learned weight vector $\mathbf{w}$ which feature values correlate positively or negatively with co-cluster membership.

The News article dataset is a prime candidate for this sort of analysis, since it has a very simple and easy to understand feature set: all of its features are cosine similarities between different pieces of text that appear in news articles, so they are on roughly the same scale, which means that different elements within the same learned vector $\mathbf{w}$ are somewhat comparable. However, note that this sort of comparison is not statistically meaningful or valid: the features in this dataset are naturally highly dependent, and the weights learned by this discriminative procedure should be treated as being likewise dependent.

In Figure 3.3, we see a graphical representation of the learned weight vectors for the models whose performance figures have been figured throughout this section, for the greedy $\mathcal{C}_G$ trained model, the relaxed $\mathcal{C}_R$ trained model, and the PCC trained model. The weights are not those as they appear in $\mathbf{w}$, but rather $\frac{\mathbf{w}}{\|\mathbf{w}\|}$. The elements of the $\mathbf{w}$ vector are grouped by feature, with the number of tokens per gram noted first, which portion of the document the feature came from noted second, and whether or not this is the result of Porter stemming noted third.

For the sake of this discussion, let $\mathbf{w}_G$, $\mathbf{w}_R$, and $\mathbf{w}_P$ be the normalized weight vector $\frac{\mathbf{w}}{\|\mathbf{w}\|}$ that results from training with $\mathcal{C}_G$, $\mathcal{C}_R$, or PCC, respectively.

As we see, the vectors learned by $\mathcal{C}_G$ and $\mathcal{C}_R$ are remarkably similar. They rarely disagree in terms of whether the weight for the corresponding feature is positive or negative, and on the three occasions that they do the result is barely perceivable in Figure 3.3. This is in somewhat stark contrast to the PCC trained vector, which is even visually quite different. Futher, $\langle \mathbf{w}_G, \mathbf{w}_P \rangle \approx 0.205$ and $\langle \mathbf{w}_R, \mathbf{w}_P \rangle \approx 0.205$ in comparison to $\langle \mathbf{w}_G, \mathbf{w}_R \rangle \approx 0.999$.

There is one very perverse effect which is evident immediately upon viewing Figure 3.3: the most helpful features for determining item pair similarity is similarity of unigram article text both stemmed and unstemmed, but perversely the least helpful feature for item pair similarity is similarity of trigram article text, both stemmed and unstemmed. Upon further examination, the reason for this appears to be the noisy nature of the extraction: when we fail to extract text from a page from a given news web site, we typically fail for every news story from that web site and, further, we fail in exactly the same way by mistakenly grabbing the same exact text from each page of that website, even those dealing with different news stories entirely. (Sometimes this is a menu of links on the page, or some side text which all pages share, or an error message, or some such.) Since trigram similarity is a better indicator than unigram similarity of absolute similarity between bodies of text, the trigram similarity is working to counteract this "mistaken identity" case, as if to say, "yes, this pair has very strong unigram similarity, but because it has very high trigram similarity it is more probable that they share exactly the same text, which means that their similarity is completely coincidental."

Another interesting effect is the usefulness of quoted text. While unigram

quoted text is not considered helpful, higher order grams are one of the more useful features for quoted text. As unlikely as it is that two articles on the same news story will be written with exactly the same text, it is highly likely that the quotes in two articles on the same news story will contain exactly the same text.

3.6.8 Efficiency of Supervised Correlation Clustering

While we did not run a formal performance comparison on the time it took to learn, in a typical run, when run on the NP-coreference problem, learning a model for SVM-CC converged after about 1000 constraints were introduced into the working set, and the resulting quadratic program was typically reoptimized 50 times. In terms of the time spent between producing the most violated constraint versus solving the quadratic problem, the overhead of finding the most violated constraint is small relative to the time spend reoptimizing the QP; using greedy $\mathcal{C}_G$ clustering, only one percent of the time spent reoptimizing the QPs was spent clustering to find the most violated constraint. This was prior to the development of the 1-slack structural SVM described in Section 2.1.4, which dramatically reduces the complexity of the quadratic program, hence the time spent in the QP solver, and consequently the time spent training the algorithm. Of all the reported experiments, the longest SVM-CC ever took to converge was between 3 and 4 hours, with under one hour as a more typical time. Due to PCC's simplicity one might suspect superior performance; however, with slightly under half a million noun-phrase pairs in the training set, training PCC's classifier required half a week with half a million constraints.

3.7 Conclusions and Discussion

We formulated a supervised correlation clustering method SVM-CC based on an SVM framework for learning structured outputs. The algorithm accepts a series of "training clusters," a series of sets of items and clusterings over that set. The method learns a similarity measure between item pairs to cluster future sets of items in the same fashion as the training clusters.

The learning algorithm's correctness depends on an ability to iteratively find and introduce the most violated constraint. Since finding the most violated constraint is intractable for clustering, we use existing clustering methods to help find an approximation. We experimentally evaluate two approximations: one based on greedy clustering, and one based on a linear programming relaxation. Both produce comparable results. Further, we find that a simplified formulation that excludes the loss from $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ does not lead to a loss in accuracy. Overall, the results suggest that SVM-CC's ability to optimize to a custom loss function and exploit transitive dependencies in data does improve performance compared to a naïve classification approach.

CHAPTER 4

SUPERVISED K-MEANS AND SPECTRAL CLUSTERING

4.1 Introduction

Among the algorithms typically used for clustering, k-means and its variants are arguably the most widely used and effective. However, successful use of k-means and other clustering methods requires a carefully chosen similarity measure that must be constructed to fit the task at hand. In this chapter, we propose a supervised learning approach to finding a similarity measure so that k-means provides the desired clusterings for the task at hand. As described in Section 1.2, given training examples of item sets with their correct clusterings, the goal is to learn a similarity measure so that future sets of items are clustered in a similar fashion. In particular, we use the techniques of Section 2.1 to provide a structural support vector machine algorithm for this supervised k-means learning problem, capable of directly optimizing a parameterized similarity measure to maximize cluster accuracy. We show theoretically and empirically that the algorithm is efficient, and that it provides improved clustering accuracy compared to non-learning methods, as well as compared to more conventional approaches to this supervised clustering problem.

This chapter differs from Chapter 3 insofar as this chapter describes learning parameterizations for k-means clustering and its variants, rather than correlation clustering. The two methods have their respective advantages. An advantage to correlation clustering is its ability to dynamically pick the number of clusters, whereas k-means requires the number of clusters to be fixed a priori at k, making them unsuitable for many tasks if the number of partitions for an output cluster-

ing of a data set $\mathbf{x}$ is unknown. On the other hand, k-means clustering can be more efficient, since correlation clustering of any type requires a maximization over $O(|\mathbf{x}|^2)$ values, whereas the common k-means practice of making distance comparisons between points and clusters and points requires a maximization over $O(k|\mathbf{x}|)$ values for each iteration, where there are usually very few iterations. However, we stress that the two frameworks are complimentary, not competitive: which is the "right" clustering algorithm, k-means/spectral clustering or correlation clustering, depends on the problem.

In contrast to previous methods proposed for learning k-means or spectral clustering parameterizations summarized in Section 1.4, this method scales even under thousands of features and training examples, can utilize existing k-means clustering algorithms in the training procedure, and can optimize the learned parameterization to perform well on many different loss functions.

4.2 Parameterized k-Means

In this section we shall introduce the k-means clustering algorithm, and then describe increasingly complex parameterizations of k-means that allows us to adjust the clusterings k-means produces through supervised learning.

The k-means clustering algorithm is classically described as taking an input set $\mathbf{x}$ of m items, $x_1, x_2, \ldots, x_m$, where each item x_i has some corresponding vector $\psi_i \in \mathbb{R}^N$. A clustering algorithm computes some clustering $\mathbf{y}$ of $\mathbf{x}$ with k clusters so as to minimize intracluster Euclidean distance over these ψ_i, i.e.,

$$\operatorname*{argmin}_{\mathbf{y}} \sum_{y \in \mathbf{y}} \sum_{x_i \in y} \left\| \psi_i - \frac{\sum_{x_j \in y} \psi_j}{|y|} \right\|_2^2. \tag{4.1}$$

Algebraic manipulation reveals this minimization is equivalent to finding $\mathbf{y}$ to maximize

$$\operatorname*{argmax}_{\mathbf{y}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \langle \psi_i, \psi_j \rangle \tag{4.2}$$

in a form often called kernel k-means [36].

To avoid confusion, note that by k-means we refer to the problem of minimizing (4.1), and *emphatically not* to any one particular instantiation of search procedure that attempts to solve this problem, e.g., batch k-means, point-iterative k-means, or spectral clustering algorithms. In short, k-means is the problem, and we are comparing algorithms that solve this problem in the context of structural SVM learning.

How can we parameterize this (4.2) objective function to provide a family of similarity measures for learning? A simple but powerful parameterization is to provide some linear weighting $\mathbf{w} \in \mathbb{R}^N$ to distort the ψ_i dimensions:

$$\operatorname*{argmax}_{\mathbf{y}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \psi_i^T \operatorname{diag}(\mathbf{w}) \psi_j. \tag{4.3}$$

We can alternately phrase (4.3) as

$$\operatorname*{argmax}_{\mathbf{y}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \langle \mathbf{w}, \psi_i \circ \psi_j \rangle . \tag{4.4}$$

Here, $\circ$ is the componentwise vector product. By changing weights in $\mathbf{w}$, we affect what clustering $\mathbf{y}$ of $\mathbf{x}$ is optimal under this parameterized k-means objective (4.4).

4.2.1 Parameterization as Kernel Learning

Though formulation of (4.4) is simple, it is a somewhat limited parameterization insofar as it requires that points explicitly exist in a vector space. To begin to gen-

eralize this, suppose instead of $\psi_i \circ \psi_j$, that any pair x_i, x_j in $\mathbf{x}$ has a corresponding pairwise vector $\psi_{ij} \in \mathbb{R}^N$.

$$\underset{\mathbf{y}}{\operatorname{argmax}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \langle \mathbf{w}, \psi_{ij} \rangle \tag{4.5}$$

If we then define a matrix $K \in \mathbb{R}^{m \times m}$ with entries

$$K_{ij} = \langle \mathbf{w}, \psi_{ij} \rangle \tag{4.6}$$

we can view (4.5) as

$$\underset{\mathbf{y}}{\operatorname{argmax}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} K_{ij}. \tag{4.7}$$

For simplicity, we assume for any K the associated $\mathbf{x}$ and $\mathbf{w}$ are obvious in context.

Work in kernel k-means clustering often specifies that K is symmetric positive semi-definite (SPSD), i.e., $K \succeq 0$ [36]. Why? The items in the set $\mathbf{x}$ have representations in some (implicit) vector space if and only if $K \succeq 0$ [65]. This is relevant to our setting, since the proof of convergence for batch k-means clustering depends on the existence of this space, and may not converge without it [65].

How can we ensure $K \succeq 0$? Consider an alternate definition of K. For a given $\mathbf{x}$, let $K^{(\ell)} \in \mathbb{R}^{m \times m}$ be the matrix of the ℓ^{th} pairwise feature in pairwise ψ_{ij}, i.e., $K^{(\ell)}_{ij} = \langle \mathbf{e}_\ell, \psi_{ij} \rangle$. We may then define K as $K = \sum_{\ell=1}^{N} \mathbf{w}_\ell K^{(\ell)}$. Restricting $\mathbf{w} \geq 0$ and all $K^{(\ell)} \succeq 0$ will imply $K \succeq 0$, since non-negative linear combinations of symmetric positive semi-definite (SPSD) matrices are likewise SPSD. This style of parameterization has strong connections to the field of kernel learning [65].

Enforcing $\mathbf{w} \geq 0$ is the responsibility of the training procedure, but the constraint on the features in the pairwise ψ_{ij} is the responsibility of the practitioner providing these vectors. Fortunately, this is usually not difficult to satisfy. For example, the very common case with pairwise vectors $\psi_{ij} = \psi_i \circ \psi_j$ seen in (4.5)

satisfies the constraint. More generally, features in ψ_{ij} whose values comes from a kernel function evaluation over $x_i, x_j \in \mathbf{x}$ satisfy the constraint.

4.2.2 Parameterization as Similarity Learning

The restrictions to enforce $K \succeq 0$ pose practical disadvantages. First, for the user providing ψ_{ij} pairwise feature vectors, ensuring that every single feature is a kernel may be difficult in some settings. Second, enforcing positivity constraints on $\mathbf{w}$ is bothersome insofar as it may complicate the parameter learning procedure, and it is even unhelpful: it is plausible that some pairwise features are *negatively* correlated with common cluster membership. To take a canonical example, if one is clustering web pages, certain link relationships among pages are often strong indicators that pages are of different types [57]. With some effort, tricks may be employed to overcome some of these difficulties (for example, doubling features with positive and negative versions of the features to allow negative correlations, and diagonal offsets large enough to ensure $K \succeq 0$), but this is troublesome and often confusing.

To avoid these problems, the alternative to Section 4.2.1's restrictions is to simply lift them, i.e., accept any ψ_{ij} pairwise vectors and parameterization $\mathbf{w}$. The cost of this greater simplicity and flexibility is that the resulting K is often no longer SPSD. Though "kernel k-means" becomes a bit of a misnomer in this case, we retain its use, as an established term for this representation. This is not a major problem, but if a practitioner does not choose to enforce constraints to ensure $K \succeq 0$, this restricts us to clustering algorithms robust to $K \not\succeq 0$ in both our training procedures and inference. In this work we do not enforce $K \succeq 0$.

4.3 Supervised k-means with Structural SVMs

With k-means parameterization defined as above, how do we actually learn a parameterization? We provide a supervised approach based on structural support vector machines, taking as input a training set

$$\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}.$$

Each $\mathbf{x}_i \in \mathcal{X}$ is a set of items and $\mathbf{y}_i \in \mathcal{Y}$ a complete partitioning of that set. For example, $\mathcal{S}$ could have $\mathbf{x}_i$ as noun-phrases in a document and $\mathbf{y}_i$ as the partitioning into co-referent sets, or $\mathbf{x}_i$ as a pixel image with $\mathbf{y}_i$ as the segmentation of the image into coherent regions, etc. The output of the learning algorithm is a $\mathbf{w}$-parameterized hypothesis $h : \mathcal{X} \to \mathcal{Y}$, where the clustering algorithm in h uses the $\mathbf{w}$ parameterized similarity measure when clustering inputs $\mathbf{x}$. Intuitively, the goal is to learn some $\mathbf{w}$ so that each $h(\mathbf{x}_i)$ is close to $\mathbf{y}_i$ on the training set, and so that h predicts the desired clustering also for unseen sets of items $\mathbf{x}$.

To refresh your memory from Section 2.1, structural SVMs are a general method for learning hypotheses with complex structured output spaces [106]. From a training set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$, a structural SVM learns a hypothesis $h : \mathcal{X} \to \mathcal{Y}$ mapping inputs $\mathbf{x} \in \mathcal{X}$ to outputs $\mathbf{y} \in \mathcal{Y}$, trading off model complexity and empirical risk, with hypotheses taking the form

$$h(\mathbf{x}) = \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} f(\mathbf{x}, \mathbf{y}), \tag{4.8}$$

maximizing a discriminant function $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ with

$$f(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle . \tag{4.9}$$

The $\mathbf{\Psi}$ combined feature vector function relates inputs and outputs, and $\mathbf{w}$ is the model parameterization learned from $\mathcal{S}$. The quality of a hypothesis is evaluated

using a loss function $\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ describing the extent to which two outputs differ. The $\mathbf{\Psi}$ and Δ functions are task dependent.

In a similar vein to Chapter 3's phrasing parameter learning for correlation clustering in the language of structural SVMs, to use structural SVMs to learn parameterizations for k-means clustering, we must state our clustering procedure $h(\mathbf{x})$ in terms of $h(\mathbf{x}) = \text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$, provide a loss function $\Delta(\mathbf{y}, \hat{\mathbf{y}})$, and provide the separation oracle $\text{argmax}_{\mathbf{y} \in \mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y})$. These are explained in the following three sections.

4.3.1 Combined Feature Function $\mathbf{\Psi}$

To express $h(\mathbf{x})$ as $h(\mathbf{x}) = \text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$, we work from (4.7) and (4.6):

$$
\begin{aligned}
h(\mathbf{x}) &= \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} K_{ij} && (4.10) \\
&\equiv \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \langle \mathbf{w}, \psi_{ij} \rangle && (4.11) \\
&\equiv \operatorname*{argmax}_{\mathbf{y} \in \mathcal{Y}} \left\langle \mathbf{w}, \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \psi_{ij} \right\rangle . && (4.12)
\end{aligned}
$$

So, $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ is

$$
\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \sum_{y \in \mathbf{y}} \frac{1}{|y|} \sum_{i,j \in y} \psi_{ij} \tag{4.13}
$$

for the most general parameterization of k-means.

In this work, we also want to represent and learn from "relaxed" clusterings, such as those that appear in methods like spectral clustering. More specifically, we shall provide a matrix representation of clusterings. Consider this alternate representation of clusterings $\mathbf{y}$: for each partitioning $\mathbf{y}$ of m items into k clusters, let $\mathbf{Y} \in \mathbb{R}^{m \times k}$ be an equivalent alternate matrix representation of the clustering.

Each column in $\mathbf{Y}$ corresponds to some cluster $y \in \mathbf{y}$, where each element i in the column is $|y|^{-0.5}$ if $i \in y$, and is 0 otherwise. For example, the following two clustering representations are equivalent:

$$\mathbf{y} = \{\{1,3\},\{2,4,5\}\} \qquad \mathbf{Y} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{3}} \\ 0 & \frac{1}{\sqrt{3}} \end{bmatrix}.$$

More formally, any matrix $\mathbf{Y}$ corresponding to a discrete clustering $\mathbf{y}$ will obey three basic constraints. First is column orthonormality: for any columns $\mathbf{Y}_{:,i}$ or $\mathbf{Y}_{:,j}$ from $\mathbf{Y}$, $\|\mathbf{Y}_{:,i}\|_2 = 1$, $\mathbf{Y}_{:,i}^T \mathbf{Y}_{:,j} = 0$, i.e., $\mathbf{Y}^T\mathbf{Y} = I$. Second, each column's nonzero entries are equal: for any pair of column $\mathbf{Y}_{:,i}$'s entries $\mathbf{Y}_{j,i} \neq 0$ and $\mathbf{Y}_{\ell,i} \neq 0$, $\mathbf{Y}_{j,i} = \mathbf{Y}_{\ell,i}$. Third is that there are no negative entries: any entry $\mathbf{Y}_{j,i} \geq 0$.

With this new representation $\mathbf{Y}$, we may rephrase (4.7) as:

$$\operatorname*{argmax}_{\mathbf{Y}} \operatorname{trace}(\mathbf{Y}^T K \mathbf{Y}). \tag{4.14}$$

We phrase the objective in terms of (4.9) to get $\mathbf{\Psi}(\mathbf{x}, \mathbf{Y})$:

$$\begin{aligned} h(\mathbf{x}) &= \operatorname*{argmax}_{\mathbf{Y}} \operatorname{trace}(\mathbf{Y}^T K \mathbf{Y}) \\ &\equiv \operatorname*{argmax}_{\mathbf{Y}} \left\langle \mathbf{w}, \sum_{i=1}^{m} \sum_{j=1}^{m} \left(\mathbf{Y}_{i,:}^T \mathbf{Y}_{j,:}\right) \psi_{ij} \right\rangle. \end{aligned}$$

So, $\mathbf{\Psi}(\mathbf{x}, \mathbf{Y})$ is

$$\mathbf{\Psi}(\mathbf{x}, \mathbf{Y}) = \sum_{i=1}^{m} \sum_{j=1}^{i-1} \left(\mathbf{Y}_{i,:}^T \mathbf{Y}_{j,:}\right) \psi_{ij}. \tag{4.15}$$

Note that (4.15) generalizes (4.13) insofar as the two are equal for any $\mathbf{Y}$ corresponding to $\mathbf{y}$, and (4.15) is defined for any spectral output $\mathbf{Y}$.

As an aside, that $\mathbf{\Psi}(\mathbf{x}, \mathbf{Y})$ is quadratic in the entries of $\mathbf{Y}$ brings up a subtle but important distinction regarding the generality of structural SVMs versus alternative formulations of OP 4, like max-margin Markov nets (M^3N) [104], associative Markov nets, and their variants [103]. These alternatives require that "inference" (in this case, k-means clustering) be phrased as either a Markov random field or linear program, respectively. One could begin to express the quadratic nature of $\mathbf{Y}$ as pairwise cliques in an M^3N, or linearize clustering by optimizing $\mathbf{Z} = \mathbf{Y}\mathbf{Y}^T$ for associative networks. However, these methods would be incapable of feasibly capturing $\mathbf{Y}$ orthonormality, or the $rank(\mathbf{Z}) = k$ constraint on $\mathbf{Z}$. In contrast, the restriction of the structure and number of columns of $\mathbf{Y}$, the nonlinearity of $\mathbf{Y}$ in $\mathbf{\Psi}$, and the nonlinearity of the clustering procedure are all incidental and naturally expressed in structural SVMs since the structure of $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ is unrestricted.

4.3.2 Loss Function Δ

The loss function Δ for the dissimilarity between two clusterings we use in this work is

$$\Delta(\mathbf{Y}, \hat{\mathbf{Y}}) = 100 \cdot \left(1 - \frac{1}{k} \operatorname{trace}(\mathbf{Y}^T \hat{\mathbf{Y}} \hat{\mathbf{Y}}^T \mathbf{Y})\right) \tag{4.16}$$

$$= 100 \cdot \left(1 - \frac{1}{k} \|\mathbf{Y}^T \hat{\mathbf{Y}}\|_F^2\right). \tag{4.17}$$

For $\mathbf{Y}$ corresponding to a discrete partitioning $\mathbf{y}$, (4.16) is

$$\Delta(\mathbf{y}, \hat{\mathbf{y}}) = 100 \cdot \left(1 - \tfrac{1}{k} \sum_{y \in \mathbf{y}} \sum_{\hat{y} \in \mathbf{y}} \frac{|y \cap \hat{y}|^2}{|y| \cdot |\hat{y}|}\right). \tag{4.18}$$

This loss Δ has attractive qualities. It is symmetric and invariant to column rearrangements. Also, as seen in (4.18), Δ essentially counts agreement among pairs of items in clusters which is normalized by the size of the clusters in question. This

is favorable relative to alternate loss functions based on the Rand index [87] used in the correlation clustering work seen in Section 3.3.1. Where this normalization is absent as in pairwise loss Δ_P, loss becomes heavily biased against mistakes in larger clusters. Finally, though any judgment about the appropriateness of a loss function must necessarily be subjective, this Δ appears to give qualitatively sensible judgments about the similarity of two clusterings.

Though irrelevant for k-means since k-means assumes that it knows the number of clusters a priori, this Δ would not be able to meaningfully compare clusterings with different numbers of clusters: for example, if $\hat{\mathbf{y}}$ is the clustering with each item in its own cluster, i.e., $\hat{\mathbf{Y}} = \mathbf{I}$, then $\Delta(\mathbf{Y}, \hat{\mathbf{Y}}) = 0$ no matter what $\mathbf{Y}$ is. However, in our applications we suppose we know the number of clusters, so this does not apply.

4.3.3 Separation Oracle and Prediction

For the separation oracle $\text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + \Delta(\mathbf{y}_i, \mathbf{y})$, the form of Δ is well suited to constructing the separation oracle: one can employ a clustering algorithm as the separation oracle and cluster over the matrix $(K - \frac{1}{k}\mathbf{Y}\mathbf{Y}^T)$ in place of K in the (4.7) objective.

However, finding the actual clustering $\mathbf{y}$ that globally maximizes (4.7) either for prediction or computing the most violated constraint is an NP-hard problem. This has led to the adoption of many varied approximate algorithms to maximize this objective function. The survey in [36] characterizes many of the popular clustering algorithms that approximate the maximization of the discriminant function (4.7). We use three methods from that paper that are all robust to $K \not\succeq 0$. We denote

these differing methods as Iterative, Spectral, and Discrete. In prediction, one could use other clustering methods if one conformed to SPSD restrictions on K as defined in Section 4.2.1, including batch k-means, normalized cut algorithms, etc. In the separation oracle, however, we must use these robust methods: even with $K \succeq 0$, it is quite possible that $(K - \frac{1}{k}\mathbf{Y}\mathbf{Y}^T) \not\succeq 0$.

Iterative is point-incremental k-means [35]. We use point-incremental (i.e., recomputing cluster centers with each point reassignment) and not standard batch (i.e., recomputing cluster centers after a pass over all points) k-means since K easily becomes non-SPSD without positivity constraints on $\mathbf{w}$'s elements, breaking batch k-means' convergence guarantees.

The algorithm works by randomly assigning all m items to k clusters, and then iterating over all points, reassigning them to the cluster with the "closest" cluster center. Unlike typical batch k-means clustering which waits until a pass is completed before updating cluster centers, point-iterative k-means updates the centers upon each point reassignment. Compared to batch k-means, point-iterative k-means does not depend upon $K \succeq 0$ and tends to produce clusterings with lower intracluster distance [35].

Since this clustering algorithm is a form of local search, one may implement an approximate separation oracle for any loss function Δ, not just that described in Section 4.3.2, at the cost of k loss function evaluations for every point reassignment.

Spectral is a straightforward eigenanalysis of K to produce a "relaxed" clustering in the matrix representation $\mathbf{Y}$ described in Section 4.3.1. If we relax Section 4.3.1's constraints on $\mathbf{Y}$ *except* for having orthonormal columns, then this

optimization problem

$$\operatorname*{argmax}_{\mathbf{Y}} \operatorname{trace}(\mathbf{Y}^T K \mathbf{Y})$$

over this multi-vector Rayleigh quotient may be maximized by assigning $\mathbf{Y}$'s columns as the eigenvectors corresponding to the k-largest eigenvalues of K. This eigenvector matrix is a relaxed "clustering" in that we have relaxed the requirements for the special structure of $\mathbf{Y}$ listed in Section 4.3.1 that ensured it corresponded to some discrete clustering $\mathbf{y}$.

Discrete is a discretized spectral method via Bach and Jordan post-processing [5], and is a combination of the previous methods: once we have our eigenvector matrix $\bar{\mathbf{Y}}$, we cluster $\bar{K} = \bar{\mathbf{Y}}\bar{\mathbf{Y}}^T$ with point-incremental k-means to find a discrete $\mathbf{y}$.

There is one subtle but important point that arises from using approximations in the separation oracle: the known performance guarantees for Algorithm 1 are known to apply only to the case where the separation oracle $\operatorname{argmax}_{\mathbf{y}\in\mathcal{Y}} H(\mathbf{y})$ of maximization of cost function H in (2.21) is calculated exactly [106]. In Section 4.3.3 we constructed our separation oracle from a clustering algorithm, but because clustering algorithms are approximations, this may not find the globally optimal $\mathbf{y}$. Here we discuss what we still guarantee about our supervised k-means algorithms, in a discussion which mirrors the similar results in Section 3.4 for correlation clustering, and is treated in far greater detail in Chapter 5.

Consider the space of possible clusterings $\mathcal{Y}$ for training example $(\mathbf{x}_i, \mathbf{y}_i)$. During training, the ideal clusterer separation oracle would find the true maximizing clustering $\mathbf{y}^* = \operatorname{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle + \Delta(\mathbf{y}_i, \mathbf{y})$. (To reiterate, under this ideal case, Algorithm 1 is guaranteed to solve OP 4.) However, this ideal is unrealizable. So what happens when we use one of our approximations?

Let us first consider polynomial time termination. The polynomial time termination guarantee still holds, since the proof does not depend on the quality of the solution, but rather on the idea that any constraint violated by more than ϵ must increase the objective by some minimum amount [106].

Correctness and empirical risk are less easy to deal with. The separation oracles can be divided into two broad categories according to what they do solve, depending on whether they use the discrete clusterers Iterative/Discrete, or relaxed Spectral.

The methods Iterative and Discrete may return some suboptimal clustering, i.e., some clustering $\hat{\mathbf{y}}$ such that

$$\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \hat{\mathbf{y}}) \rangle + \Delta(\mathbf{y}_i, \hat{\mathbf{y}}) < \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}^*) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}^*). \tag{4.19}$$

In such a suboptimal case, constraints violated by more than ϵ in OP 4 may go undetected by Algorithm 1, leading to termination with a solution infeasible in OP 4. In other words, the problem becomes underconstrained. We term this type of suboptimal approximation undergenerating in the discussion of Section 5.3.1, which contains deeper theoretical discussion.

The method Spectral is a very different animal. Rather than searching $\mathcal{Y}$ for local maxima, it instead searches some relaxed $\overline{\mathcal{Y}}$ space which it can efficiently search for a global maximum. In this case, $\overline{\mathcal{Y}}$ is the space of all indicator matrices $\mathbf{Y}$ where the special structure of entries described in the Spectral method in Section 4.3.3 is abandoned, save for the requirement of orthonormal columns. More to the point, $\mathcal{Y} \subset \overline{\mathcal{Y}}$, and because the separation oracle searches over $\overline{\mathcal{Y}}$, at the end of Algorithm 1 we not only shall have all constraints in OP 4 respected, but additional constraints from outputs $\mathbf{Y} \in (\overline{\mathcal{Y}} - \mathcal{Y})$. The solution is feasible but probably suboptimal with respect to OP 4. The problem becomes overconstrained.

We will later term this type of relaxed approximation overgenerating in the much deeper discussion of Section 5.3.2.

Either underconstrained or overconstrained learning has its unique costs. With underconstrained learning, since constraints in OP 4 may be violated, slack no longer bounds empirical risk, thus eroding one of the basic principles of SVM learning. On the other hand, with overconstrained learning, Algorithm 1 solves a problem which accounts for outputs that would never arise from a discrete clustering algorithm, thus unnecessarily ruling out parameterizations $\mathbf{w}$ which may yield superior performance. It is unclear theoretically whether either way is better, so our experiments shall provide an empirical evaluation of both underconstrained and overconstrained learning for k-means style clustering.

4.4 Training Algorithm

For clarity, we present Algorithm 7, the training algorithm for learning the parameterization $\mathbf{w}$ of k-means or spectral clustering given the training sample $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$ with regularization parameter C and tolerance ϵ. This algorithm is Algorithm 1, instantiated with k-means clustering as described in the previous material of the chapter. In this description we use $\mathbf{Y}$ as a general term for both discrete or relaxed clusterings, as it generalizes both relaxed clusterings $\mathbf{Y}$ and discrete clusterings $\mathbf{y}$. The clustering algorithm, whether discrete k-means or spectral clustering, is that augmented with loss Δ as described in Section 4.3.3.

(Supervised k-Means/Spectral Clustering Algorithm)

Input: $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)$, C, ϵ
$S_i \leftarrow \emptyset$ for all $i = 1, \ldots, n$
repeat
  for $i = 1, \ldots, n$ **do**
    $\hat{\mathbf{Y}} \leftarrow$ k-means/spectral clustering of $\mathbf{x}_i$ augmented with loss Δ
    compute $\xi_i = \max\{0, \max_{\mathbf{Y} \in S_i} \Delta(\mathbf{y}_i, \mathbf{Y}) + \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{Y}) \rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle\}$
    if $\Delta(\mathbf{y}_i, \hat{\mathbf{Y}}) + \left\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \hat{\mathbf{Y}}) \right\rangle - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i) \rangle > \xi_i + \epsilon$ **then**
      $S_i \leftarrow S_i \cup \{\hat{\mathbf{Y}}\}$
      $\mathbf{w} \leftarrow$ optimize primal over $\bigcup_i S_i$
    end if
  end for
until no S_i has changed during iteration

Algorithm 7: Cutting plane algorithm for the supervised k-means/spectral clustering problem, based on the cutting plane algorithm for structural SVMs Algorithm 1.

4.5 Empirical Analysis

We implemented supervised k-means clustering with the SVMpython structural SVM package [39]. The module's code, instructions and examples of use, as well as the datasets that we used in our experiments, are accessible from:

`http://www.cs.cornell.edu/~tomf/projects/supervisedkmeans/`.

To empirically analyze our methods, we compare it to conventionally trained and untrained clusterers, and also provide comparisons of our methods using underconstrained and overconstrained learning on real and synthetic datasets. Parameterizations $\mathbf{w}$ and pairwise vectors ψ_{ij} are unconstrained as outlined in Section 4.2.2, i.e., not requiring $K \succeq 0$.

In all experiments, pairwise feature vectors ψ_{ij} are composed from "node" features vectors $\bar{\psi}_i, \bar{\psi}_j \in \mathbb{R}^{N_n}$ and an explicitly provided pairwise feature vector

Table 4.1: Dataset statistics, including number of example clusterings n, number of clusters k in each example clustering, average number of points m in the clusterings, node features N_n, and pairwise features N_p. (The SSVM learns $N = N_n + N_p$ weights in $\mathbf{w}$.)

Dataset	n	k	Avg. m	N_n	N_p
WebKB-L	4	6	1041	50397	100796
WebKB-N	4	6	1041	41131	0
News 8-1	7	10	150	0	30
News 8-2	7	10	150	0	30
News 8-4	7	10	150	0	30
Synth	5	5	100	0	750

$\bar{\psi}_{ij} \in \mathbb{R}^{N_p}$ such that

$$\psi_{ij} = \begin{bmatrix} \bar{\psi}_i \circ \bar{\psi}_j \\ \bar{\psi}_{ij} \end{bmatrix}.$$

Pairwise feature vectors ψ_{ij} are in $\mathbb{R}^N$ where $N = N_n + N_p$, and correspondingly we have $\mathbf{w} \in \mathbb{R}^N$. Some datasets evaluated have no node or explicit pairwise features, i.e., sometimes $N_n = 0$ or $N_p = 0$.

We shall provide a web page with our software for download, as well as the datasets shown in this chapter.

4.5.1 Datasets

We used three general "families" of datasets in our empirical analysis, from which we drew one or more specific evaluation datasets. The datasets are listed in Table 4.1.

News is a dataset related to the news article clustering dataset of Section 3.6 insofar as it uses different features, although we organize and use it in separate ways. The sets of items and partitioning was collected through trawling Google News for one day and extracting the text of news articles from the linked news sites. During any particular day, there are many different topics or stories. The set of articles in a particular story for a day form each of the example clusters. Each area (Google News has seven major areas: Business, Entertainment, Health, Nation, Sports, Technology, World) serves as an example clustering, with the stories forming the clusters, and the articles as the cluster elements. The data is expressed as a pairwise feature vector between articles, where each feature is the cosine similarity of TFIDF weighted token vectors, where these token vectors are unigrams, bigrams, and trigrams of text in the title, article text, and quoted sections of the article text, in both original and Porter stemmed versions of the features, for 30 features in all. We sampled from three days (August 1, 2, and 4 of 2004) to get three datasets (News 8-1, News 8-2, and News 8-4). Each of these news datasets contains seven example clusterings corresponding to the general area, and each clustering itself contains ten clusters, i.e., $k = 10$.

Its use in our experiment was to make a separate predictive experiment for each clustering as the test set. For example, when testing the predictive performance when the "Entertainment" clustering in the News 8-2 clustering was the test set, the six clusterings corresponding to the remaining areas in News 8-2 (Business, Health, Nation, Sports, Technology, World) comprised the training set, with no clusterings from 8-1 or 8-4 being present. So, there are 21 separate predictive experiments, each experiment corresponding to a differing clustering being chosen as the test clusterer.

In the same vein as the discussion of Section 1.5, note that this is another example of a learning task which would be impossible to phrase as a multiclass classification task, since we would not encounter the partitions (or "classess') used in prediction in our training set.

WebKB consists of web pages retrieved from the computer science departments of four universities, labeled as being a course web page, faculty page, student page, etc [25]. It is often used in classification and multiclass classification tasks that seek to exploit the link structure among the web documents. In our experiment, we effectively turned this into two closely related datasets.

One of these datasets contains only node features (WebKB-N) as TFIDF-scaled unigram word count vectors. There are no pairwise features.

The other dataset (WebKB-L) contains these word count features and additional features relating to the relationships among these documents, and also critically a pairwise feature vector with two regions, corresponding to documents where one document links to another, and another where both are linked from the same document (co-citation). If documents are linked or co-cited, the respective region in the pairwise feature vector will contain the componentwise product of the node features, plus a single 1 indicator feature. If they are not linked or co-cited, the corresponding region is zeroed.

Though this is naturally a classification rather than a clustering problem, as we know what classes will occur in our test data a priori, it nonetheless serves as an appropriate test bed for our supervised clustering algorithms as well.

Synth is a synthetic dataset meant to emphasize the importance of some features being harmful and others helpful, in the face of significant noise. It was

Table 4.2: Range of C values tested during the leave-one-out search for training hyperparameters. All powers of ten between and including these endpoints were considered. A separate C value was chosen for each different test set within that dataset through evaluating leave-one-out error on the resulting training set.

Dataset	Low C	High C	Dataset	Low C	High C
WebKB-L	$1{\cdot}10^{-1}$	$1{\cdot}10^{4}$	News	$1{\cdot}10^{0}$	$1{\cdot}10^{5}$
WebKB-N	$1{\cdot}10^{0}$	$1{\cdot}10^{5}$	Synth	$1{\cdot}10^{-2}$	$1{\cdot}10^{3}$

generated in this way: there are 5 clusters, each with 20 points. Between every pair of the 100 points is a pairwise feature vector. This pairwise feature vector is comprised of 15 "regions" (one for each possible cluster pair), each region with 50 features (so 750 pairwise features total). For a pair of points in clusters i and j, the feature "region" corresponding to i, j will have 5 of the 50 features active. Also, noise is introduced for each pairwise feature vector: instead of consistently indexing the region (i, j), it will 20% of the time replace i with a random cluster (so 16% of the time it will differ from i), and the same for j. So, only about 70.5% of pairwise vectors have the "correct" index. (Without noise, learned clusterers produced perfect clusterings. While useful as a sanity check, it makes for uninteresting comparisons.) Only one dataset was generated.

4.5.2 Experimental Setup

To evaluate performance, we trained k-means parameterizations on our dataset. For each dataset of n clustering examples, we ran n experiments, where each clustering was taken in turn as the single example "test set" with the $n-1$ remaining

clusterings as the training set. For each experiment, LOO cross validation was used on the $n-1$ size training set to choose the two training hyperparameters: C (values drawn from a sample of powers of 10 seen in Table 4.2), and which classifier to use as the final predictor (Iterative, Spectral, or Discrete).

The parameterizations were trained with Iterative and Spectral separation oracle supervised k-means trainers. In addition to these supervised k-means clustering methods, we have two baselines.

Pair is a model training method based on binary classifiers by taking all pairwise feature vectors, considering whether the associated pair is in the same cluster, and treating it as a binary classification problem trained for accuracy with an SVM. During classification, entries in the similarity matrix K are outputs of the learned binary classifier. This style of supervised clustering using binary classifiers has been successfully used in work on noun-phrase coreference resolution [78]. The resulting training method differs from supervised k-means clustering insofar as the clustering procedure and desired Δ are not considered in training, but it will still try to increase or decrease the similarity of pairs in or out of the same cluster, respectively. Hyperparameters (C and clusterer in prediction) were selected in an identical fashion to supervised k-means clustering.

None is a second baseline, which consists of Iterative classification with all equal weights, i.e., no training at all.

4.5.3 Clustering Accuracy

Table 4.3 and Table 4.4 detail the loss figures resulting from training the clusterer with the Iterative and Spectral separation oracle (columns Iterative and Spectral),

training the clusterer with the pairwise binary classifier (column Pair), and with no training (column None). While loss Δ values can reach 100, a more reasonable upper bound is $\frac{k-1}{k} \cdot 100$, the loss resulting from putting all points together in 1 cluster.

4.5.4 Supervised Clustering vs. Pairwise/Untrained

How do efforts to do any supervised k-means clustering compare against the more naïve pairwise binary training? On the WebKB-L, WebKB-N and News datasets, the performance gains from structural SVM training in Δ figures are quite dramatic, and both Iterative and Spectral trained supervised k-means clustering methods outperform these baselines on these datasets every time.

The relationship on Synth is somewhat different: while there are differences, the pairwise trained model even "wins" once (testing on cluster 3), and the extent to which either class of supervised k-means clustering models wins is not conclusively better, statistically speaking. Why does this happen? One important power of supervised clustering methods is their ability to exploit cluster structure: two items $i, j \in \mathbf{x}$ with low similarity K_{ij} can still be in the same cluster owing to the effect of other items in $\mathbf{x}$. In contrast, the baseline pairwise classifier treats all judgments on pairwise ψ_{ij} independently. However, since all ψ_{ij} are generated independently in the synthetic dataset and there is no long range dependency structure to exploit, pairwise classification for training $\mathbf{w}$ works fine.

The untrained model does quite poorly in Synth, but this is expected since the dataset was generated specifically to contain large numbers of pairwise features correlated negatively with co-cluster membership. Recall from Section 4.5.1 that

our Synth dataset had 15 "regions" of pairwise features (corresponding to each unordered pair of the 5 clusters), so only 5 of these 15 regions, or one third of all pairwise features, were positively correlated with co-cluster membership, whereas the remaining two-thirds are negatively correlated.

4.5.5 Discrete Iterative vs. Relaxed Spectral Clustering in Training

How does discrete Iterative compare against the relaxed Spectral when used as a separation oracle during training?

We use non-parametric tests like Fisher sign or Wilcoxon signed-rank tests. While the loss figures are not independent since they result from shared training sets, we accept these non-parametric tests as an imperfect measure that nevertheless gives some indication of difference.

Results of the comparison are seen in Table 4.5. These results reflect the feeling one might get glancing at Table 4.3 or Table 4.4: there is no clear winner in WebKB or News. The exception is the Synth synthetic data set, where the Iterative trained model appears to yield superior performance.

4.5.6 The Importance of Link Features in WebKB

The WebKB-L dataset differs from WebKB-N in that it contains pairwise features relevant to the hyperlink structure in the corpus, whereas WebKB-N are straightforward document vectors. Each of the 8 supervised k-means clustering WebKB-L

Table 4.3: Loss Δ on the test sets of the two WebKB datasets and the Synth dataset (lower is better). The left columns identify the dataset and the particular clustering used as the test dataset in the corresponding row.

Dataset	Test Clustering	Iterative	Spectral	Pair	None
WebKB-L	Cornell	45.3	53.3	79.7	74.7
	Texas	59.8	56.7	78.9	72.8
	Washington	53.1	46.6	60.6	76.2
	Wisconsin	47.3	60.2	81.1	77.5
WebKB-N	Cornell	63.0	61.4	74.8	78.6
	Texas	69.9	56.8	75.5	78.7
	Washington	68.8	58.2	74.9	78.3
	Wisconsin	72.6	66.2	77.0	78.6
Synth	1	43.3	55.6	48.1	74.7
	2	53.4	58.7	54.7	74.7
	3	56.0	56.7	55.2	74.7
	4	39.3	59.5	43.9	74.7
	5	40.3	63.4	49.1	74.7

trained models outperform their corresponding WebKB-N trained model. While 8 wins to 0 losses is statistically significant under a sign test, these loss Δ figures are not independent; nevertheless, the magnitude of the differences, always over 10 in the case of Iterative trained models, suggests a substantial gain. As the usefulness of exploiting hyperlink structure in WebKB is a feature of most publications featuring this dataset, it is important that our methods are able to handle definitions of these general pairwise features.

Table 4.4: Loss Δ on the test sets of the three News datasets (lower is better). The left columns identify the dataset and the particular clustering used as the test dataset in the corresponding row.

Dataset	Test Clustering	Iterative	Spectral	Pair	None
News 8-1	Business	23.7	20.6	45.2	49.5
	Entertainment	12.7	22.2	53.0	25.9
	Health	28.1	28.7	57.4	38.8
	Nation	3.8	3.8	40.2	14.6
	Sports	15.2	14.3	47.6	59.9
	Technology	35.9	30.4	51.7	37.3
	World	3.7	2.4	41.7	62.1
News 8-2	Business	3.6	4.6	34.1	63.8
	Entertainment	22.7	9.5	40.1	22.8
	Health	20.4	20.4	48.4	43.9
	Nation	24.6	23.7	47.4	60.6
	Sports	20.2	15.8	59.3	57.0
	Technology	16.1	13.8	48.3	41.3
	World	12.2	11.9	50.5	70.4
News 8-4	Business	19.7	14.9	42.7	33.5
	Entertainment	4.6	6.3	46.8	32.4
	Health	15.0	16.2	51.7	32.1
	Nation	19.4	20.3	41.2	30.0
	Sports	19.0	19.0	55.6	54.7
	Technology	5.8	11.6	46.4	37.6
	World	4.8	5.8	39.6	39.3

Table 4.5: Counts of the times within Table 4.3 and Table 4.4 the Iterative trained model won, tied, or lost versus the Spectral trained model respectively. For the purpose of this count, the results from the three news datasets are aggregated. The W, $n_{s/r}$, $P_{\text{1-tail}}$ columns are quantities relevant to the Wilcoxon test, where W is the sum of signed ranks, $n_{s/r}$ the number of non-tied trials, and $P_{\text{1-tail}}$ the level of significance.

Dataset	Win	Tie	Lose	W	$n_{s/r}$	$P_{\text{1-tail}}$
WebKB-L	2	0	2	4	4	>0.05
WebKB-N	0	0	4	4	4	>0.05
News	8	3	10	30	18	0.2611
Synth	5	0	0	15	5	0.05

4.5.7 Efficiency of Supervised k-Means/Spectral Clustering

Clustering performance aside, how does training time depend on characteristics of the dataset? To answer this question empirically, we took the basic Synth dataset described in Section 4.5.1. The basic dataset has 5 clustering examples, 5 clusters, 750 features, and 100 points. To test the algorithms in a controlled way, we varied each of these characteristics (examples, clusters, features, points), and trained over 20 training sets to test the time it took to train a model. Results are reported for both Iterative and Spectral clustering. The regularization parameter $C = 10^4$ was constant in all training methods. The structural SVM used in training was the 1-slack structural SVM described in Section 2.1.4.

As we increase the number of training example clusterings in our training set, Figure 4.1 reveals a relationship linear for Spectral and approximately linear for It-

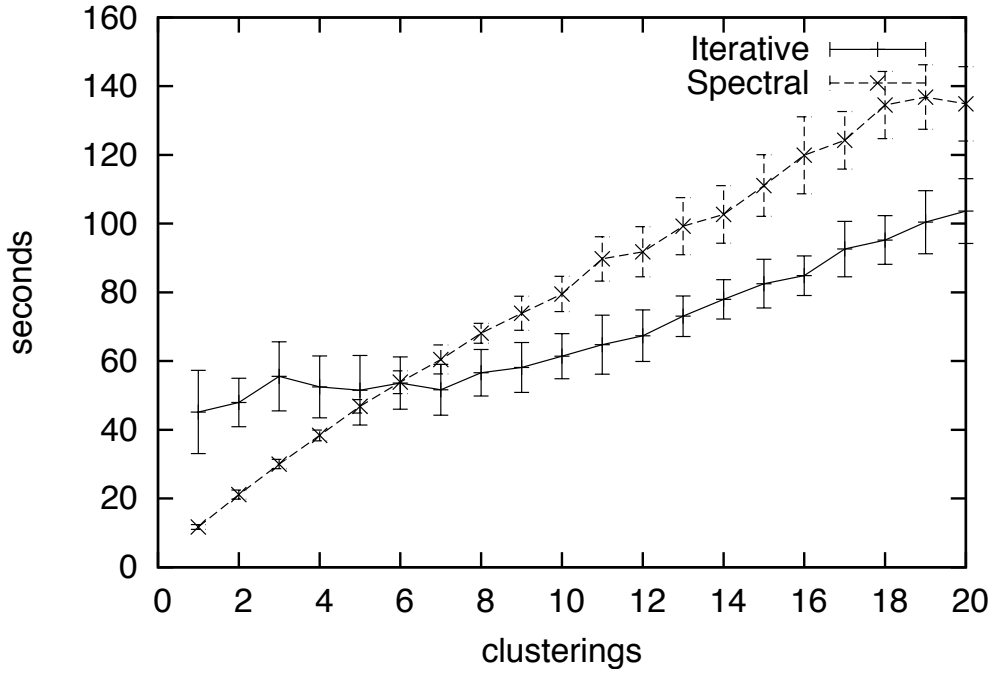

Figure 4.1: Results of a timing experiment on a synthetic dataset where we varied the number of example clusterings n in the training set.

erative. That training time is linear in the number of training examples is expected due to this being the 1-slack structural SVM of Section 2.1.4.

Figure 4.2 shows that increasing the number of clusters while holding other statistics constant leads to a steady decrease in training time for Spectral trained methods. This appears to be a symptom of the difficulty of learning this dataset: the number of points and dimensions is constant, but spread over an increasing number of clusters in each example. Consequently the best hypothesis that can be reasonably extracted from the provided data becomes weaker, and fewer iterations are required to converge. The Iterative method, on the other hand, often takes longer. Logs reveal this is due to one or two iterations where Iterative being used as the separation oracle took a very long time to converge, explaining the unstable nature of the curve.

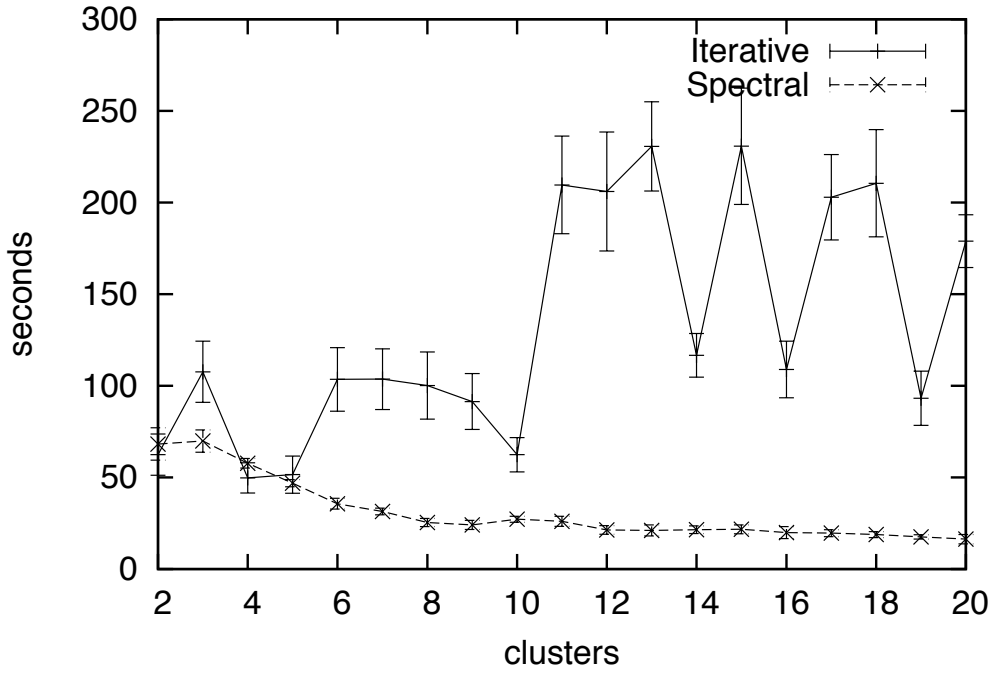

Figure 4.2: Results of a timing experiment on a synthetic dataset where we varied the number of example clusters k in each example.

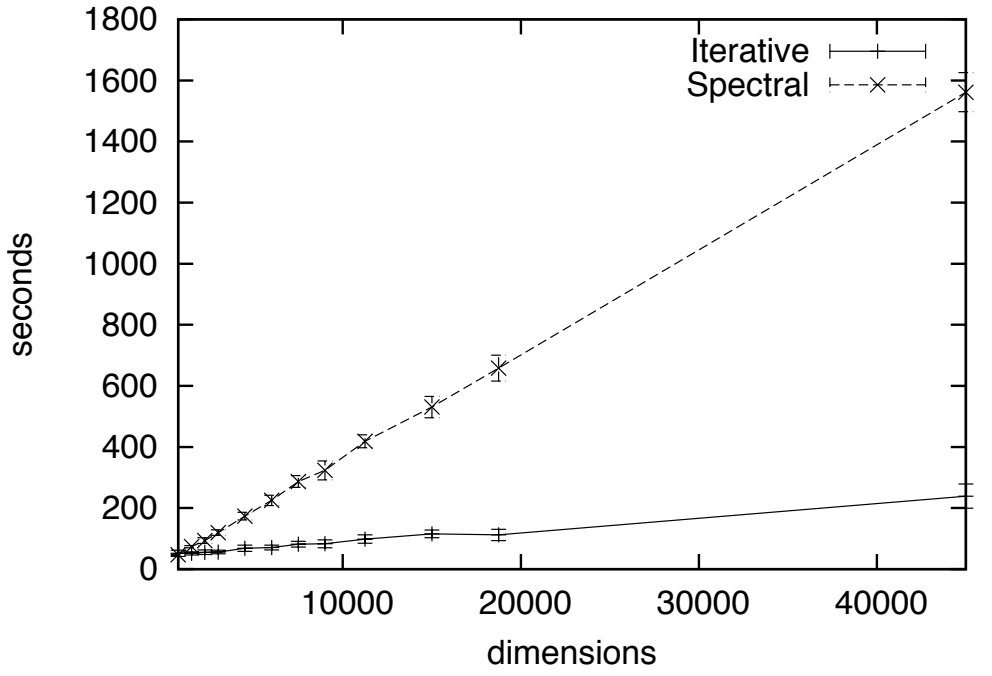

Figure 4.3: Results of a timing experiment on a synthetic dataset where we varied the number of features N in every pairwise feature vector.

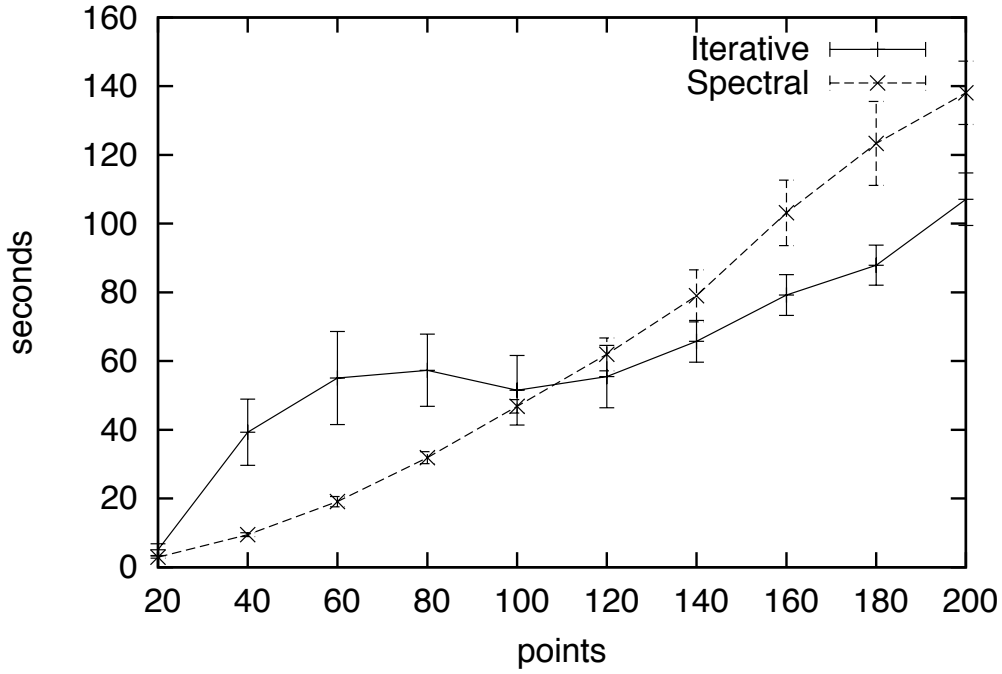

Figure 4.4: Results of a timing experiment on a synthetic dataset where we varied the number of points within each cluster m/k in the training set examples.

Figure 4.3 shows a linear relationship of number of features versus training time. This linear time relationship is unsurprising given that computing similarities and evaluating the $\mathbf{\Psi}$ function is linear in the number of features. Spectral is slower than Iterative both on account of the speed of the clustering algorithm, as well as requiring more iterations of Algorithm 1.

Let's now examine how training time varies with the number of points in each cluster. Figure 4.4 shows Spectral time complexity as a straightforward polynomially increasing curve (due to the LAPACK `DSYEVR` eigenpair procedure working on steadily larger matrices). The Iterative trained classifier also tends to increase with number of points, with a hump on lower numbers of points arising from Iterative clustering often requiring more time for the clusterer to converge on smaller datasets, a tendency reversed as more points presumably smooth the search space.

One theme seen throughout is that the timing behavior of relaxed spectral training is very predictable relative to the discrete k-means training. Considering the somewhat unpredictable nature of local search versus largely deterministic matrix computations, it is unsurprising to see the latter's relative stability carry over into model training time.

4.6 Conclusions and Discussion

We provided a means to parameterize the popular canonical k-means clustering algorithm based on learning a similarity measure between item pairs, and then provided a supervised k-means clustering method to learn these parameterizations using a structural SVM. The supervised k-means clustering method learns this similarity measure based on a training set of item sets and complete partitionings over those sets, choosing parameterizations optimized for good performance over the training set.

We then theoretically characterized the learning algorithm, drawing a distinction between the iterative local search k-means clustering method and the relaxed spectral relaxation, as leading to underconstrained and overconstrained supervised k-means clustering learners, respectively. Empirically, the supervised k-means clustering algorithms exhibited superior performance compared to naïve pairwise learning or unsupervised k-means. The underconstrained and overconstrained supervised k-means clustering learners compared to each other exhibited different performance, though neither was clearly consistently superior to the other. We also characterized the runtime behavior of both the supervised k-means clustering learners through an empirical analysis on datasets with varying numbers of ex-

amples, clusters, features, and items to cluster. We find training time is linear or better in the number of example clusterings, clusters per example, and number of features.

CHAPTER 5

APPROXIMATION ALGORITHMS AND STRUCTURAL SVMS

The supervised clustering algorithms in Chapter 3 and Chapter 4 relied upon a separation oracle that returned approximate solutions, i.e., one that did not necessarily return the most violated constraint, despite the theoretical guarantees seen in Section 2.1.3 depending upon finding the exact most violated constraint. While empirically the methods often learn model parameterization that perform well, a deep understanding of the use of approximations in structural support vector machines remains limited. Existing theoretical guarantees of solution quality and correctness must be revised and reviewed under the case of using approximations.

This problem is not unique to structural SVMs and supervised clustering. Discriminative training methods like conditional random fields [64], maximum-margin Markov networks [104], and structural SVMs [107] all share this problem. They all have theoretical guarantees of one form or another about training procedure convergence, and some mathematical characterization of solution quality, as seen in the case of structural SVMs in Section 2.1.3. Indeed, in cases where exact inference is tractable, these learning methods have substantially improving prediction performance on a variety of structured prediction problems, including part-of-speech tagging [3], natural language parsing [107], sequence alignment [114], and classification under multivariate loss functions [51].

However, these theoretical guarantees hold only under certain assumptions. In the context of structural SVMs, the basic assumption was that both the inference problem (i.e., computing a prediction) and the separation oracle required in the cutting-plane training algorith of Algorithm 1 can be solved exactly. As we saw in Chapter 3 and Chapter 4, clustering does not obey this assumption, as finding

the cluster that corresponds to the global maximum of the discriminant function of either correlation clustering or k-means is an NP-hard problem.

There are larger issues at stake beyond supervised clustering as well. In many important machine learning problems (e.g., multi-label classification, image segmentation, machine translation) the natural methods of modeling these problems lead to situations where exact inference and, the case of structural SVMs, the separation oracle are computationally intractable. Unfortunately, use of approximations in these settings abandons many of the existing theoretical guarantees of structural SVM training, and relatively little is known about discriminative training using approximations. While the work of Chapter 3 and Chapter 4, as well as the work of the existing literature of this field gives us some empirical confidence that these methods work at least sometimes, we do not really understand when they work and when they do not work.

To help expand our knowledge of approximate inference in structured learning, this chapter explores training structural SVMs on problems where exact inference is intractable. A pairwise fully connected Markov random field (MRF) serves as a representative class of intractable models. This class includes natural formulations of models for multi-label classification, image segmentation, and clustering. We identify two classes of approximation algorithms for the separation oracle in the structural SVM cutting-plane training algorithm, namely undergenerating and overgenerating algorithms, and we adapt loopy belief propagation (LBP), greedy search, and linear-programming and graph-cut relaxations to this problem. We provide a theoretical and empirical analysis of using these algorithms with structural SVMs.

We find substantial differences between different approximate algorithms in

training and inference. In particular, much of the existing theory can be extended to overgenerating though not undergenerating methods. In experimental results, intriguingly, our structural SVM formulations using the overgenerating linear-programming and graph-cut relaxations successfully learn models in which relaxed inference is "easy" (i.e., the relaxed solution is mostly integral), leading to robust and accurate models. We conclude that the relaxation formulations are preferable over the formulations involving LBP and greedy search.

5.1 Approximations in Structured Output Prediction

Several discriminative structural learners were proposed in recent years, including conditional random fields (CRFs) [64], Perceptron HMMs [23], max-margin Markov networks (M^3Ns) [104], and structural SVMs (SSVMs) [107]. As seen in Chapter 2, notational differences aside, these methods all learn (kernelized) linear discriminant functions, but differ in how they choose model parameters.

Recall that in Section 2.1.1 we introduced the margin-scaled structural SVM problem in the form of OP 4. However, because this problem in its raw form requires introducing a constraint for every wrong output, a step which is typically intractable, Algorithm 1 was introduced to solve OP 4 through use of a cutting plane algorithm. Algorithm 1 iteratively constructs a sufficient subset $\bigcup_i S_i$ of constraints and solves the QP only over this subset (line 10).

The algorithm employs a separation oracle to find the next constraint to include into the working set (line 7 of Algorithm 1). It finds the currently most violated constraint (or, a constraint that is violated by at least the desired precision ϵ), corresponding to the constraint which, if active under the current model parameterization $\mathbf{w}$, would require the greatest slack. If a polynomial time separation

oracle exists, OP 4 and Algorithm 1 have three theoretical guarantees [107] which are presented in detail in Section 2.1.3 and reviewed very briefly here:

Polynomial Time Termination: Algorithm 1 terminates in a polynomial number of iterations, and thus overall polynomial time.

Correctness: Algorithm 1 solves OP 4 accurate to a desired precision ϵ, since Algorithm 1 terminates only when all constraints in OP 4 are respected within ϵ (lines 9 and 14).

Empirical Risk Bound: Since each ξ_i upper bounds training loss $\Delta(\mathbf{y}_i, h(\mathbf{x}_i))$, $\frac{1}{n}\sum_{i=1}^{n}\xi_i$ upper bounds empirical risk.

Unfortunately, proofs of these properties rely on the separation oracle (line 7) being exactly solvable, and do not necessarily hold with approximations. We will later analyze which properties are retained.

5.2 Markov Random Fields in Structural SVMs

A special case of structural SVM that we will examine throughout this chapter is that applied to Markov random field (MRF), the same base formulation as seen in M^{3}N [104]. In this, $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ is constructed from an MRF log-potential function

$$f(\mathbf{x}, \mathbf{y}) = \sum_{k \in \text{cliques}(G)} \phi_k(y_{\{k\}}) \tag{5.1}$$

with graph structure $G = (V, E)$ and the loss function is restricted to be linearly decomposable in the cliques, i.e., $\Delta(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{k \in \text{cliques}(G)} \delta_k(y_{\{k\}}, \hat{y}_{\{k\}})$. Here, $\mathbf{y}$ is the value assignment to variables, δ_k are sub-component local loss functions, and ϕ_k are potential functions representing the fitness of variable assignment $y_{\{k\}}$ to clique

k. The network potential $f(\mathbf{x}, \mathbf{y})$ serves as a discriminant function representing the variable assignment $\mathbf{y}$ in the structural SVM, and $h(\mathbf{x}) = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} f(\mathbf{x}, \mathbf{y})$ serves as the maximum a posteriori (MAP) prediction.

OP 4 requires we express (5.1) in the form $f(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$. First express potentials as $\phi_k(y_{\{k\}}) = \left\langle \mathbf{w}, \psi(\mathbf{x}, y_{\{k\}}) \right\rangle$. The feature vector functions ψ_k relate $\mathbf{x}$ and label assignments $y_{\{k\}}$. Then, $f(\mathbf{x}, \mathbf{y}) = \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ where $\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \sum_{k\in\text{cliques}(G)} \psi_k(\mathbf{x}, y_{\{k\}})$.

In the following, we use a particular linearly decomposable loss function that simply counts the percentage proportion of different labels in $\mathbf{y}$ and $\hat{\mathbf{y}}$, i.e., $\Delta(\mathbf{y}, \hat{\mathbf{y}}) = \|100 \cdot \mathbf{y} - \hat{\mathbf{y}}\|_0 / |V|$. Further, in our applications, labels are binary (i.e., each $y_u \in \mathbb{B} = \{0, 1\}$), and we allow only $\phi_u(1)$ and $\phi_{uv}(1, 1)$ potentials to be non-zero. This latter restriction may seem onerous, but any pairwise binary MRF with non-zero $\phi_u(0), \phi_{uv}(0, 0), \phi_{uv}(0, 1), \phi_{uv}(1, 0)$ has an equivalent MRF where these potentials are zero.

To use Algorithm 1 for MRF training and prediction, one must solve two argmax problems:

Prediction: $\text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$

Separation Oracle: $\text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y})$

The prediction problem is equivalent to MAP inference. Also, we can state the separation oracle as MAP inference. Taking the MRF we would use to solve $\text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$, we include $\Delta(\mathbf{y}_i, \mathbf{y})$ in the argmax by incrementing the node potential $\phi_u(y)$ by $\frac{100}{|V|}$ for each wrong value y of u, since each wrong variable assignment increases loss by $\frac{100}{|V|}$. Thus, we may express the separation oracle as MAP inference.

5.3 Approximate Inference Theory

Unfortunately, MAP inference is $\#P$-complete for general MRFs. Fortunately, a variety of approximate inference methods exist. For prediction and the separation oracle, we explore two general classes of approximate inference methods, which we call undergenerating and overgenerating approximations.

5.3.1 Undergenerating Approximations

Undergenerating methods approximate $\text{argmax}_{\mathbf{y}\in\mathcal{Y}}$ by $\text{argmax}_{\mathbf{y}\in\underline{\mathcal{Y}}}$, where $\underline{\mathcal{Y}} \subseteq \mathcal{Y}$; in more conventional language, the maximizing $\mathbf{y}$ that they return may not be a global maxima in $\mathcal{Y}$. In Algorithm 1, because the separation oracle is searching a subset of the constraints, at the time of termination there may still remain constraints in OP 4 violated by more than ϵ. In this way, use of such a method in the separation oracle may result in a quadratic problem which is underconstrained with respect to the true optimal solution OP 4. In supervised correlation clustering we had the greedy approximation of Section 3.4.1, and in supervised k-means clustering the iterative point-incremental and discretized spectral clustering methods of Section 4.3.3.

In this work dealing with Markov random fields, we consider the following undergenerating methods in the context of MRFs:

Greedy iteratively changes the single variable value y_u that would increase network potential most.

LBP is loopy belief propagation [79].

Combine picks the assignment $\mathbf{y}$ with the highest network potential from both greedy and LBP.

We now theoretically characterize undergenerating learning and prediction. All theorems generalize to any learning problem, not just MRFs. Due to space constraints, provided proofs are proof skeletons.

Since undergenerating approximations can be arbitrarily poor, we must restrict our consideration to a subclass of undergenerating approximations to make meaningful theoretical statements. This analysis focuses on ρ-approximation algorithms, with $\rho \in (0, 1]$. What is a ρ-approximation? In our case, for predictive inference, if $\mathbf{y}^* = \text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ is the true optimum and $\mathbf{y}'$ the ρ-approximation output, then

$$\rho \cdot \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}^*) \rangle \leq \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}') \rangle \tag{5.2}$$

Similarly, for our separation oracle, for $\mathbf{y}^* = \text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y})$ as the true optimum, and if $\mathbf{y}'$ corresponds to the constraint found by our ρ-approximation, we know

$$\rho \left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}^*) \rangle + \Delta(\mathbf{y}_i, \mathbf{y}^*) \right] \leq \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}') \rangle + \Delta(\mathbf{y}_i, \mathbf{y}') \tag{5.3}$$

For simplicity, this analysis supposes $\mathcal{S}$ contains exactly one training example $(\mathbf{x}_0, \mathbf{y}_0)$. However, this is easily generalizable. To generalize, one may view n training examples as 1 example, where inference consists of n separate processes with combined outputs, etc. In a similar fashion, combined ρ-approximation outputs may be viewed as a single ρ-approximation output. Further, this practice of effectively combining multiple examples into one example reflects the actual implementation of the structural SVM [53].

Theorem 8. (Polynomial Time Termination) *If* $\bar{R} = \max_{i, \mathbf{y} \in \mathcal{Y}} \|\Psi(\mathbf{x}_i, \mathbf{y})\|$,

$\bar{\Delta} = \max_{i,\mathbf{y}\in\mathcal{Y}} \|\Delta(\mathbf{y}_i, \mathbf{y})\|$ *are finite, an undergenerating learner terminates after adding at most* $\epsilon^{-2}(C\bar{\Delta}^2\bar{R}^2 + n\bar{\Delta})$ *constraints.*

Proof. The original proof holds as it does not depend upon separation oracle quality (Algorithm 1, line 7). □

Lemma 1. *After line 6 in Algorithm 1, let* $\mathbf{w}$ *be the current model,* $\hat{\mathbf{y}}$ *the constraint found with the* ρ*-approximation separation oracle, and* $\hat{\xi} = H(\hat{\mathbf{y}})$ *the slack associated with* $\hat{\mathbf{y}}$*. Then,* $\mathbf{w}$ *and slack* $\hat{\xi} + \frac{1-\rho}{\rho}\left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \hat{\mathbf{y}})\rangle + \Delta(\mathbf{y}_0, \hat{\mathbf{y}})\right]$ *is feasible in OP 4.*

Proof. To outline the proof idea, if we knew the true most violated constraint $\mathbf{y}^*$, we would know the minimum ξ^* such that $\mathbf{w}, \xi^*$ was feasible in OP 4. The proof upper bounds ξ^*.

With a ρ-approximation algorithm as our separation oracle, instead of solving $\mathbf{y}^* = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} \Delta(\mathbf{y}_0, \mathbf{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y})\rangle$ exactly, we find some $\hat{y}$ such that

$$\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle \geq \rho\left[\Delta(\mathbf{y}_0, \mathbf{y}^*) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}^*)\rangle\right] \tag{5.4}$$

Since we did not solve $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ exactly, we have not necessarily found the most violated constraint. In fact, we have underestimated the slack required to make the current model $\mathbf{w}$ feasible under OP 4 by exactly this amount.

$$\left[\Delta(\mathbf{y}_0, \mathbf{y}^*) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}^*)\rangle\right] - \left[\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle\right] \tag{5.5}$$

The first term of (5.5) is unknown, but we have the benefit of the ρ-approximation bound to help us. We can be certain that we have not underestimated the required

slack by more than

$$\begin{aligned}
&[\Delta(\mathbf{y}_0, \mathbf{y}^*) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}^*)\rangle] - [\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle] \\
\leq\ & \frac{1}{\rho}\left[\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle\right] - [\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle] \\
=\ & \frac{1-\rho}{\rho}\left[\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle\right]
\end{aligned}$$

So, we know that the true slack ξ^* required for this example under $\mathbf{w}$ obeys

$$\xi^* \leq \hat{\xi} + \frac{1-\rho}{\rho}\left[\Delta(\mathbf{y}_0, \hat{y}) + \langle \mathbf{w}, \Psi(\mathbf{x}_0, \hat{y})\rangle\right] \tag{5.6}$$

Since the $\mathbf{w}$ is feasible under slack ξ^*, it must also be feasible under this upper bound. □

Theorem 9. *When iteration ceases with the result $\mathbf{w}, \xi$, if $\hat{\mathbf{y}}$ was the last found most violated constraint, we know that the optimum objective function value v^* for OP 4 lies in the interval*

$$\begin{aligned}
&\tfrac{1}{2}\|\mathbf{w}\|^2 + C\xi \leq v^* \leq \\
&\quad \tfrac{1}{2}\|\mathbf{w}\|^2 + C\left[\tfrac{1}{\rho}\left[\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \hat{\mathbf{y}})\rangle + \Delta(\mathbf{y}_0, \hat{\mathbf{y}})\right] - \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \mathbf{y}_0)\rangle\right]
\end{aligned}$$

Proof. This is simply Lemma 1 applied to the last iteration. □

So, even with ρ-approximate separation oracles, one may bound how far off a final solution is from solving OP 4. Sensibly, the better the approximation, i.e., as ρ approaches 1, the tighter the solution bound.

The next result concerns empirical risk. The SVM margin attempts to ensure that high-loss outputs have a low discriminant function value, and ρ-approximations produce outputs within a certain factor of optimum. As seen in Theorem 1, any $(\mathbf{w}, \xi)$ solution to OP 4 which is feasible (and not even necessarily optimal) will

have a ξ-based upper bound empirical risk, but only under the condition that $h(\mathbf{x}) = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y})\rangle$, i.e., $h(\mathbf{x})$ does not return an approximation to this argmax but rather the true maximizing argument. Recall that the proof of Theorem 1 depends upon the fact that if $\Delta(\mathbf{y}_i, h(\mathbf{x}_i)) > 0$, then it must be that $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, h(\mathbf{x}_i))\rangle > \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle$, leading to the constraint associated with $h(\mathbf{x}_i)$ requiring a greater slack. However, if $h(\mathbf{x}_i)$ does not return a maximizing argument, this proof falls apart. However, if we suppose h uses a ρ-approximate algorithm for inference, we can say something about the resulting empirical risk

Theorem 10. (ρ-Approximate Empirical Risk) *For $\mathbf{w}, \xi$ feasible in OP 4 from training with single example $(\mathbf{x}_0, \mathbf{y}_0)$, the empirical risk using ρ-approximate prediction has upper bound $(1-\rho)\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \mathbf{y}_0)\rangle + \xi$.*

Proof. The idea of the proof is to take the constraint associated with the output $\mathbf{y}' = h(\mathbf{x}_0)$ from OP 4 associated constraint, which we must be respecting if we have a feasible solution, then apply known bounds to the constraint's $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \mathbf{y}')\rangle$ term.

We have a single example $(\mathbf{x}_0, \mathbf{y}_0)$, with slack ξ. We know

$$\Delta\left(\mathbf{y}_0, \underset{\mathbf{y}}{\text{argmax}} \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y})\rangle\right) \leq \xi, \tag{5.7}$$

hence the claim that ξ upper bounds empirical risk. The thing is, ξ upper bounds empirical risk only when our prediction function h exactly solves that argmax. However, in general, based on the constraints in OP 4, we know that for any $\mathbf{y}'$ with the feasible solution $\mathbf{w}, \xi$:

$$\Delta(\mathbf{y}_0, \mathbf{y}') \leq \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}_0)\rangle - \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}')\rangle + \xi. \tag{5.8}$$

To illustrate the usefulness of this statement, let's first think of this in the "known separable" case, i.e., we have managed to find a feasible solution to OP 4

such that $\xi = 0$. In this case, it must be that for our training example $(\mathbf{x}_0, \mathbf{y}_0)$, the $\mathbf{y}_0$ is a maximizer, that is, $\mathbf{y}_0$ is a valid solution for $\text{argmax}_{\mathbf{y}} \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}) \rangle$, and in the case where there are multiple optimizers, any such $\hat{\mathbf{y}}$ must have $\Delta(\mathbf{y}_0, \hat{\mathbf{y}}) = 0$. In the case where we have a ρ-approximator, whatever such $\mathbf{y}'$ we find from this approximation must have $\langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}') \rangle \geq \rho \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}_0) \rangle$, and consequently $\Delta(\mathbf{y}_0, \mathbf{y}') \leq (1 - \rho) \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}_0) \rangle$. So, while ξ no longer necessarily bounds empirical risk when our predictor is a ρ-approximation, the existence of the margin-scaling-by-loss allows us to still say something useful about empirical risk.

The case where $\xi > 0$, the inseparable (or, more precisely, not provably separable) case is a little more difficult to imagine, but the bound of (5.8) still holds. However, this quantity is known only once we have made a prediction $\mathbf{y}'$, with no information available a priori. However, with some minimum fuss, we can produce a bound.

$$\begin{aligned} \Delta(\mathbf{y}_0, \mathbf{y}') &\leq \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}_0) \rangle - \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}') \rangle + \xi & (5.9) \\ &\leq (1 - \rho) \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}_0) \rangle + \xi & (5.10) \end{aligned}$$

This last relies upon

$$\begin{aligned} \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}') \rangle &\geq \langle \mathbf{w}, \rho \Psi(\mathbf{x}_0, \mathbf{y}^*) \rangle & (5.11) \\ &\geq \langle \mathbf{w}, \rho \Psi(\mathbf{x}_0, \mathbf{y}_0) \rangle & (5.12) \end{aligned}$$

where $\mathbf{y}^* = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} \langle \mathbf{w}, \Psi(\mathbf{x}_0, \mathbf{y}) \rangle$. In this way we see that the inseparable case is similar to the separable case. The theorem comes from (5.10). □

If also using undergenerating ρ-approximate training, one may employ Theorem 9 to get a feasible ξ.

5.3.2 Overgenerating Approximations

Overgenerating methods approximate $\text{argmax}_{\mathbf{y}\in\mathcal{Y}}$ by $\text{argmax}_{\mathbf{y}\in\overline{\mathcal{Y}}}$, where $\overline{\mathcal{Y}} \supseteq \mathcal{Y}$. We consider the following overgenerating methods:

LProg is an expression of the inference problem as a relaxed integer linear program [11]. We first add $y_{uv} \in \mathbb{B}$ values indicating if $y_u = y_v = 1$ to linearize the program:

$$\max_{\mathbf{y}} \sum_{u\in\{1..|V|\}} y_u \phi_u(1) + \sum_{u,v\in\{1..|V|\}} y_{uv}\phi_{uv}(1,1) \tag{5.13}$$

$$\text{s.t.} \quad \forall u, v. \quad y_u \geq y_{uv} \qquad y_v \geq y_{uv} \tag{5.14}$$

$$y_u + y_v \leq 1 + y_{uv} \qquad y_u, y_{uv} \in \mathbb{B} \tag{5.15}$$

We relax $\mathbb{B}$ to $[0, 1]$ to admit fractional solutions. Importantly, there is always some optimal solution where all $y_u, y_{uv} \in \{0, \frac{1}{2}, 1\}$ [46].

Cut is quadratic pseudo-Boolean optimization using a graph-cut [58]. This is a different relaxation where, instead of $\mathbf{y} \in \mathbb{B}^{|V|}$, we have $\mathbf{y} \in \{0, 1, \varnothing\}^{|V|}$.

The LProg and Cut approximations share two important properties [11, 46]: *Equivalence* says that maximizing solutions of the Cut and LProg formulations are transmutable. One proof defines this transmutation procedure, where $\varnothing$ (in cuts optimization) and $\frac{1}{2}$ (in LP optimization) variable assignments are interchangeable [11]. The important practical implication of equivalence is both approximations return what amounts to the same solution, modulo special cases where there are non-unique maximizing assignments. *Persistence* says unambiguous labels (i.e., not fractional or $\varnothing$) are optimal labels.

As a final detail, in the case of LProg, we have a loss function

$$\Delta(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{|V|} \sum_{u \in \{1..|V|\}} |y_u - \hat{y}_u| \tag{5.16}$$

and combined feature function

$$\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \sum_{u \in \{1..|V|\}} y_u \psi_u(1) + \sum_{u,v \in \{1..|V|\}} y_{uv} \psi_{uv}(1,1). \tag{5.17}$$

Cut's functions have similar formulations.

Theorem 11. (Polynomial Time Termination) *If* $\bar{R} = \max_{i,\mathbf{y} \in \overline{\mathcal{Y}}} \|\Psi(\mathbf{x}_i, \mathbf{y})\|$, $\bar{\Delta} = \max_{i,\mathbf{y} \in \overline{\mathcal{Y}}} \|\Delta(\mathbf{y}_i, \mathbf{y})\|$ *are finite (*$\overline{\mathcal{Y}}$ *replacing* $\mathcal{Y}$ *in the overgenerating case), an overgenerating learner terminates after adding at most* $\epsilon^{-2}(C\bar{\Delta}^2\bar{R}^2 + n\bar{\Delta})$ *constraints.*

Proof. The original proof holds as an overgenerating learner is a straightforward structural learning problem on a modified output range $\overline{\mathcal{Y}}$. □

Theorem 12. (Correctness) *An overgenerating Algorithm 1 terminates with* $\mathbf{w}, \xi$ *feasible in OP 4.*

Proof. The learner considers a superset of outputs $\overline{\mathcal{Y}} \supseteq \mathcal{Y}$, so constraints in OP 4 are respected within ϵ. □

With these "extra" constraints from overgenerating inference, Algorithm 1's solution may be suboptimal w.r.t. the original OP 4. Further, for undergenerating methods, correctness does not hold, as Algorithm 1 may not find violated constraints present in OP 4.

Theorem 13. (Empirical Risk Bound) *If prediction and the separation oracle use the same overgenerating algorithm, Algorithm 1 terminates with* $\frac{1}{n}\sum_i \xi_i$ *upper bounding empirical risk* $R^{\Delta}_{\mathcal{S}}(h)$.

Proof. Similar to the proof of Theorem 11.

5.3.3 Related Work

In prior work on discriminative training using approximate inference, structural SVMs have learned models for correlation clustering, utilizing both greedy and LP relaxed approximations [40]. For M^3Ns, Anguelov et al. [4] proposed to directly fold a linear relaxation into OP 4. This leads to a very large QP, and is inapplicable to other inference methods like LBP or cuts. Furthermore, we will see below that the linear-program relaxation is the slowest method. With CRFs, likelihood training requires computing the partition function in addition to MAP inference. Therefore, the partition function is approximated [26, 47, 62, 109], or the model is simplified to make the partition function tractable [101], or CRF max-likelihood training is replaced with Perceptron training [92].

The closest work on this subject is a theoretical analysis of MRF structural learning with LBP and LP-relaxation approximations using structural perceptron learning [59]. It defines the concepts *separable* (i.e., there exists $\mathbf{w}$ such that $\forall(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}, \mathbf{y} \in \mathcal{Y}, \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i)\rangle \geq \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y})\rangle$), *algorithmically separable* (i.e., there exists $\mathbf{w}$ so that empirical risk under the inference algorithm is 0), and *learnable* (i.e., the learner using the inference method finds a separating $\mathbf{w}$). The paper illustrates that, when using approximate inference, these concepts are not equivalent. Our work's major differences are our analysis handles non-zero training error, generalizes to any structural problem, uses structural SVMs, and we have an empirical analysis.

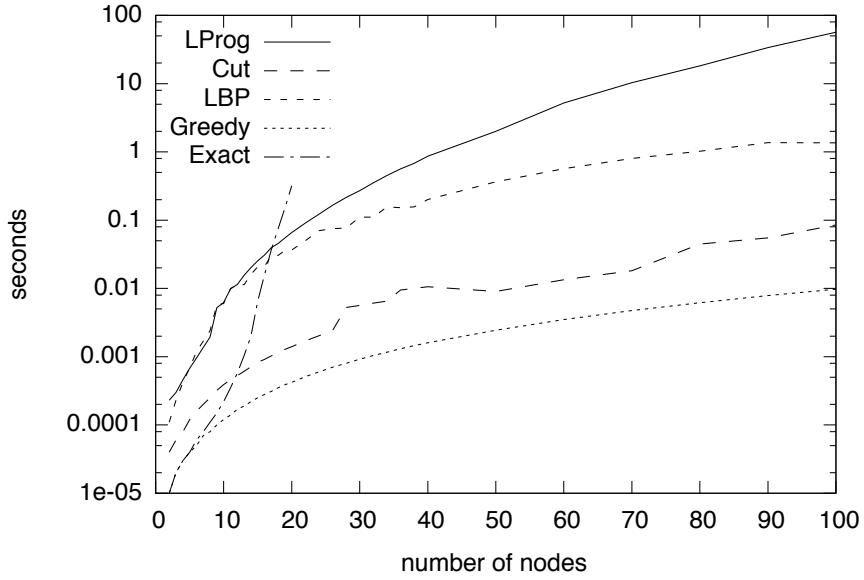

Figure 5.1: Runtime comparison. Average inference time for different methods on random problems of different sizes.

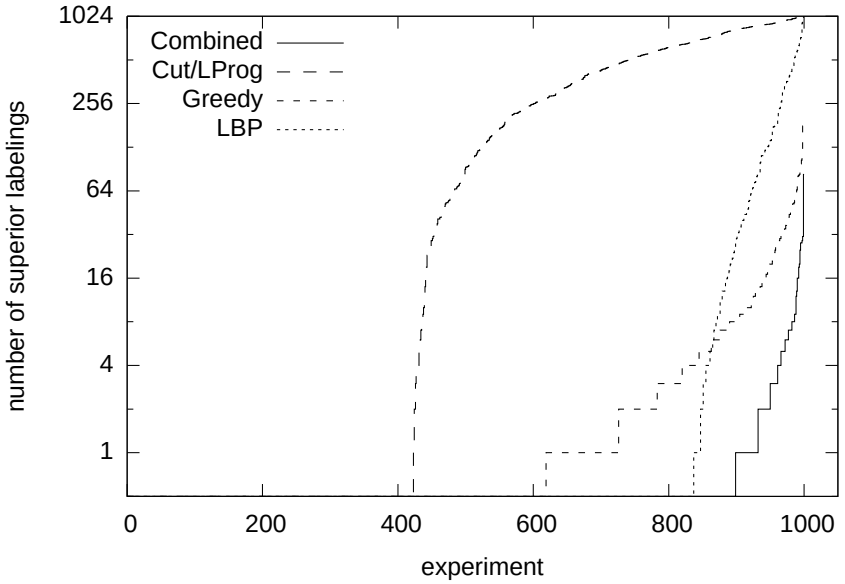

Figure 5.2: Quality comparison. Inference on 1000 random 18 label problems. Lower curves are better.

5.4 Experiments: Approximate Inference

Before we move into learning experiments, it helps to understand the runtime and quality performance characteristics of our MAP inference algorithms.

For runtime, Figure 5.1 illustrates each approximate inference method's average time to solve a single pairwise fully connected MRF with random potentials as the number of nodes increases.[1] Note that cuts are substantially faster than LBP, and several orders of magnitude faster than the linear relaxation while maintaining equivalence.

For evaluating solution quality, we generate 1000 random problems, ran the inference methods, and exhaustively count how many labelings with higher discriminant value exist. The resulting curve for 10-node MRFs is shown in Figure 5.2. For cut, $\varnothing$ labels are randomly assigned to 0 or 1. The lower the curve, the better the inference method. LBP finds "perfect" labelings more often than Greedy, but also tends to fall into horrible local maxima. Combined does much better than either alone; apparently the strengths of Greedy and LBP are complimentary.

Finally, note the apparent terrible performance of Cut, which is due to assigning many $\varnothing$ labels. At first glance, persistence is an attractive property since we *know* unambiguous labels are correct, but on the other hand, classifying only when it is certain leads it to leave many labels ambiguous.

5.5 Experiments: Approximate Learning

Our goal in the following experiments is to gain insight about how different approximate MRF inference methods perform in SSVM learning and classification. Our evaluation uses multi-label classification using pairwise fully connected MRFs

[1]Implementation details: The methods were C-language Python extension modules. LProg was implemented in GLPK (see `http://www.gnu.org/software/glpk/glpk.html`). Cut was implemented with Maxflow software [13]. Other methods are home-spun. Experiments were run on a 2.6 GHz P4 Linux box.

as an example application.

Multi-label classification bears similarity to multi-class classification, except classes are not mutually exclusive, e.g., a news article may be about both "Iraq" and "oil." Often, incorporating inter-label dependencies into the model can improve performance [16, 38].

How do we model this labeling procedure as an MRF? For each input $\mathbf{x}$, we construct an MRF with a vertex for each possible label, with values from $\mathbb{B} = \{0, 1\}$ (1 indicates $\mathbf{x}$ has the corresponding label), and an edge for each vertex pair (i.e., complete graph MRF).

What are our potential functions? In these problems, inputs $\mathbf{x} \in \mathbb{R}^m$ are feature vectors. Each of the ℓ possible labels u is associated with a weight vector $\mathbf{w}_u \in \mathbb{R}^m$. The resulting vertex potentials are $\phi_u(1) = \langle \mathbf{w}_u, \mathbf{x} \rangle$. Edge potentials $\phi_{uv}(1, 1)$ come from individual values in $\mathbf{w}$, one for each label pair. Thus, the overall parameter vector $\mathbf{w} \in \mathbb{R}^{\ell m + \binom{\ell}{2}}$ has ℓm weights for the ℓ different $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_\ell$ sub-component weight vectors, and $\binom{\ell}{2}$ parameters for edge potentials. In terms of ψ functions, $\psi_u(\mathbf{x}, 1)$ vectors contain an offset version of $\mathbf{x}$ to "select out" $\mathbf{w}_u$ from $\mathbf{w}$, and $\psi_{uv}(\mathbf{x}, 1, 1)$ vectors have a single entry set to 1 to "select" the appropriate element from the end of $\mathbf{w}$.

5.5.1 Datasets and Model Training Details

We use six multi-label datasets to evaluate performance. Table 5.1 contains statistics on these datasets. Four real datasets, **Scene** [12], **Yeast** [38], **Reuters** (the RCV1 subset 1 data set) [67], and **Mediamill** [99], came from the LIBSVM multi-label dataset collection [17]. **Synth1** is a synthetic dataset of 6 labels. Labels

Table 5.1: Basic statistics for the datasets, including number of labels, training and test set sizes, number of features, and parameter vector **w** size, and performance on baseline trained methods and a default model parameterization.

Dataset	Labels	Train	Test	Features	**w** Size	Baseline	Default
Scene	6	1211	1196	294	1779	11.43±.29	18.10
Yeast	14	1500	917	103	1533	20.91±.55	25.09
Reuters	10	2916	2914	47236	472405	4.96±.09	15.80
Mediamill	10	29415	12168	120	1245	18.60±.14	25.37
Synth1	6	471	5045	6000	36015	8.99±.08	16.34
Synth2	10	1000	10000	40	445	9.80±.09	10.00

follow a simple probabilistic pattern: label i is on half the time label $i-1$ is on and never otherwise, and label 1 is always on. Also, each label has 1000 related binary features (the learner does not know a priori which feature belong to each label): if i is on, a random 10 of its 1000 features are set to 1. All features otherwise implicitly have value 0. This hypothesis is learnable without edge potentials, but exploiting label dependency structure may result in better models, since edge potentials could capture the dependency that label i cannot be on if $i-1$ is on, etc. **Synth2** is a synthetic dataset of 10 labels. In this case, each example has exactly one label on, and all other labels are off; which of the 10 labels is on for a given example is chosen randomly. There are also 40 features. For an example, if label i is on, $4i$ randomly chosen features are set to 1. So, an example with 4 non-zero feature values has label 1 and no other labels, an example with 8 non-zero feature values has label 2 and no other labels, etc. Only models with edge potentials could possibly learn this concept, since the node potentials are a linear function, which could not capture just with node potentials alone the sort of relationships the node would have to observe (e.g., for the node corresponding to label 2 "on-ness," the

node would have to capture that with 8 features on, the node should be "on," that is, have value 1, but with 4 or 12 or more features on the node should be "off," that is, have value 0).

We used 10-fold cross validation to choose C from 14 possible values

$$C \in \{1{\cdot}10^{-2}, 3{\cdot}10^{-2}, 1{\cdot}10^{-1}, \ldots, 3{\cdot}10^{4}\}. \tag{5.18}$$

This C was then used when training a model on all training data. A separate C was chosen for each dataset and separation oracle, but not for each predictive inference method; this means that all performance figures for a given separation oracle and dataset (i.e., a row in one of the groups of Table 5.2 and Table 5.3) come from the same trained model.

5.5.2 Results and Analysis

Table 5.2 and Table 5.3 report loss on the test set followed by standard error. For each dataset, we present losses for each combination of separation oracle used in learning (the rows) and of predictive inference procedure used in classification (the columns). This lets us distinguish badly learned models from bad inference procedures as explanations for inferior performance.

For purpose of our base comparisons, we include three other inference methods to help shed insight into the workings of the structural SVM under approximate inference.

Baseline trains an MRF with no edges, making exact inference trivial at the cost of having no label dependencies. Performance measures for models trained for this loss appear in the "Baseline" column of Table 5.1.

Table 5.2: Multi-labeling loss on the first group of three of the six datasets. Results are grouped by dataset. Rows indicate separation oracle method. Columns indicate classification inference method.

	Greedy	LBP	Combine	Exact	LProg
	Scene Dataset				
Greedy	10.67±.28	10.74±.28	10.67±.28	10.67±.28	10.67±.28
LBP	10.45±.27	10.54±.27	10.45±.27	10.42±.27	10.49±.27
Combine	10.72±.28	11.78±.30	10.72±.28	10.77±.28	11.20±.29
Exact	10.08±.26	10.33±.27	10.08±.26	10.06±.26	10.20±.26
LProg	10.55±.27	10.49±.27	10.49±.27	10.49±.27	10.49±.27
	Yeast Dataset				
Greedy	21.62±.56	21.77±.56	21.58±.56	21.62±.56	24.42±.61
LBP	24.32±.61	24.32±.61	24.32±.61	24.32±.61	24.32±.61
Combine	22.33±.57	37.24±.77	22.32±.57	21.82±.56	42.72±.81
Exact	23.38±.59	21.99±.57	21.06±.55	20.23±.53	45.90±.82
LProg	20.47±.54	20.45±.54	20.47±.54	20.48±.54	20.49±.54
	Reuters Dataset				
Greedy	5.32±.09	13.38±.21	5.06±.09	5.42±.09	16.98±.26
LBP	15.80±.25	15.80±.25	15.80±.25	15.80±.25	15.80±.25
Combine	4.90±.09	4.57±.08	4.53±.08	4.49±.08	4.55±.08
Exact	6.36±.11	5.54±.10	5.67±.10	5.59±.10	5.62±.10
LProg	6.73±.12	6.41±.11	6.38±.11	6.38±.11	6.38±.11

Table 5.3: Multi-labeling loss on the second group of three of the six datasets. Results are grouped by dataset. Rows indicate separation oracle method. Columns indicate classification inference method.

	Greedy	LBP	Combine	Exact	LProg
	Mediamill Dataset				
Greedy	23.39±.16	25.66±.17	24.32±.17	24.92±.17	27.05±.18
LBP	22.83±.16	22.83±.16	22.83±.16	22.83±.16	22.83±.16
Combine	19.56±.14	20.12±.15	19.72±.14	19.82±.14	20.23±.15
Exact	19.07±.14	27.23±.18	19.08±.14	18.75±.14	36.83±.21
LProg	18.50±.14	18.26±.14	18.26±.14	18.21±.14	18.29±.14
	Synth1 Dataset				
Greedy	8.86±.08	8.86±.08	8.86±.08	8.86±.08	8.86±.08
LBP	13.94±.12	13.94±.12	13.94±.12	13.94±.12	13.94±.12
Combine	8.86±.08	8.86±.08	8.86±.08	8.86±.08	8.86±.08
Exact	6.89±.06	6.86±.06	6.86±.06	6.86±.06	6.86±.06
LProg	8.94±.08	8.94±.08	8.94±.08	8.94±.08	8.94±.08
	Synth2 Dataset				
Greedy	7.27±.07	27.92±.20	7.27±.07	7.28±.07	19.03±.15
LBP	10.00±.09	10.00±.09	10.00±.09	10.00±.09	10.00±.09
Combine	7.90±.07	26.39±.19	7.90±.07	7.90±.07	18.11±.15
Exact	7.04±.07	25.71±.19	7.04±.07	7.04±.07	17.80±.15
LProg	5.83±.05	6.63±.06	5.83±.05	5.83±.05	6.29±.06

Default always predicts the best-performing single labeling of the training set. In some sense, this is the worst one could do. This loss also appears in Table 5.1.

Exact exhaustively searches all labelings. For comparative purposes it is useful to know how we would do if we actually solved OP 4. Note that in order to enable comparisons on the Reuters and Mediamill datasets, we pruned these datasets so only the 10 most frequent labels were present.

Cut is omitted from Table 5.2 and Table 5.3. Its equivalence to LProg means the two are interchangeable and always produce the same results, excepting Cut's superior speed.

In all datasets, some edged model always exceeds the performance of the edgeless model. On Mediamill and Reuters, selecting only the 10 most frequent labels robs the dataset of many dependency relationships, which may explain the relatively lackluster performance.

The Sorry State of LBP, but Relax

Let's first examine the diagonal entries in Table 5.2 and Table 5.3. Models trained with LBP separation oracles yield generally poor performance. What causes this? LBP's tendency to fall into horrible local maxima (as seen in Section 5.4) misled Algorithm 1 to believe its most violated constraint was not violated, leading it to early termination, mirroring the result in [59]. The combined method remedies some of these problems; however, LProg still gives significantly better/worse performance on 3 vs. 1 datasets.

How does LProg training compare against exact training? Table 5.2 and Table 5.3 show that both methods give similar performance. Exact-trained models

Table 5.4: Percentage of "ambiguous" labels in relaxed inference. Columns represent different data sets. Rows represent different methods used as separation oracles in training.

	Scene	Yeast	Reuters	Mediamill	Synth1	Synth2
Greedy	0.43%	17.02%	31.28%	20.81%	0.00%	31.17%
LBP	0.31%	0.00%	0.00%	0.00%	0.00%	0.00%
Combine	2.90%	91.42%	0.44%	4.27%	0.00%	29.11%
Exact	0.95%	84.30%	0.67%	65.58%	0.00%	27.92%
LProg	0.00%	0.43%	0.32%	1.30%	0.00%	1.48%

significantly outperform relaxed-trained models on two datasets, but they also lose on two datasets.

Relaxation in Learning and Prediction

Observe that relaxation used *in prediction* performs well when applied to models trained with relaxation. However, on models trained with non-relaxed methods (i.e., models that do not constrain fractional solutions), relaxed inference often performs quite poorly. The most ludicrous examples appear in Yeast, Reuters, Mediamill, and Synth2. Table 5.4 suggests an explanation for this effect. The table lists the percentage of ambiguous labels from the relaxed classifier (fractional in LProg, $\varnothing$ in Cut). Ignoring degenerate LBP-trained models, the relaxed predictor *always* has the fewest ambiguous judgments. Apparently, SSVMs with relaxed separation oracles produce models that disfavor non-integer solutions. In retrospect, this is unsurprising: ambiguous labels always incur loss during training. Minimizing loss during training therefore not only reduces training error, but also encourages parameterizations that favor integral (i.e., exact) solutions. Under-generating and exact training do not control for this, leading to relaxed inference

yielding many ambiguous labelings.

On the other hand, observe that models trained with the relaxed separation oracle have relatively consistent performance, irrespective of the classification inference procedure; even LBP never shows the catastrophic failure it does with other training approximations and even exact training (e.g., Mediamill, Synth2). Why might this occur? Recall the persistence property from Section 5.3: unambiguous labels are optimal labels. In some respects, this property is attractive, but Section 5.4 revealed its dark side: relaxation predictors are very conservative, delivering unambiguous labels only when they are *certain*. By making things "obvious" for the relaxed predictors (which are the most conservative with respect to what they label), it appears they simultaneously make things obvious for all predictors, explaining the consistent performance of relaxed-trained models regardless of prediction method.

SSVM's ability to train models to "adapt" to the weakness of overgenerating predictors is an interesting complement with Searn structural learning [29], which trains models to adapt to the weaknesses of undergenerating search based predictors.

Known Approximations

How robust is SSVM training to an increasingly poor approximate separation oracle? To evaluate this, we built an artificial ρ-approximation separation oracle: for example $(\mathbf{x}_i, \mathbf{y}_i)$ we exhaustively find the optimal $\mathbf{y}^* = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} \langle w, \mathbf{\Psi}(\mathbf{x}_i, \mathbf{y}) \rangle + \Delta(\mathbf{y}_i, \mathbf{y})$, but we return the labeling $\hat{\mathbf{y}}$ such that $f(\mathbf{x}, \hat{\mathbf{y}}) \approx \rho f(\mathbf{x}, \mathbf{y}^*)$, finding the $\hat{\mathbf{y}}$ with discriminant function value closest to $\rho f(\mathbf{x}, \mathbf{y}^*)$ without exceeding it. In this way, we build an approximate undergenerating MRF inference method with

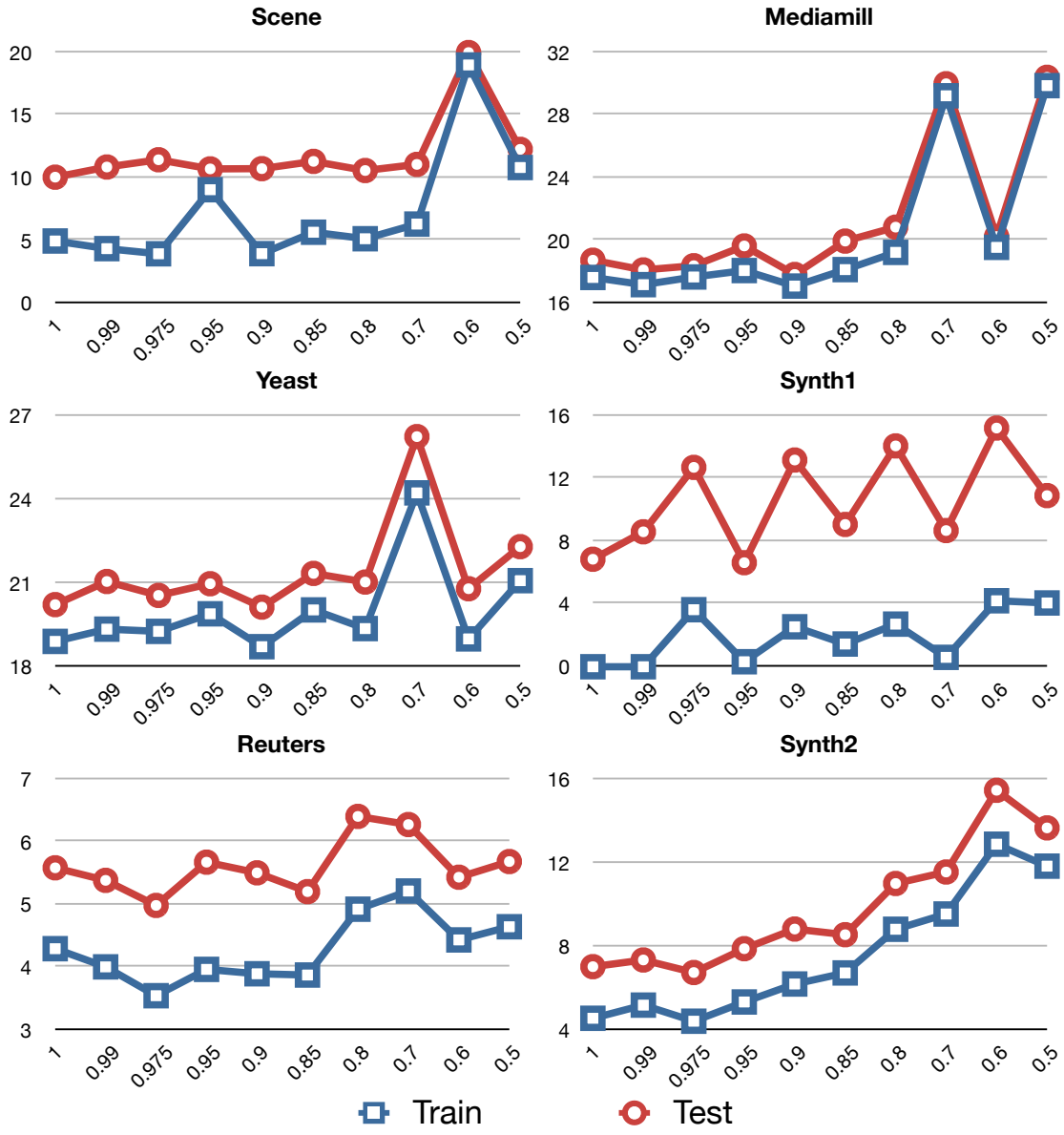

Figure 5.3: Known ρ-approximations chart, showing the information of Table 5.5 graphically.

Table 5.5: Known ρ-approximations table, showing performance change as we use increasingly inferior separation oracles.

ρ Approx.	Scene		Yeast		Reuters		Mediamill		Synth1		Synth2	
Factor	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test	Train	Test
1.000	4.97	10.06	18.91	20.23	4.30	5.59	17.65	18.75	0.00	6.86	4.57	7.04
0.990	4.36	10.87	19.35	21.06	4.01	5.39	17.19	18.13	0.00	8.61	5.20	7.36
0.975	3.95	11.45	19.27	20.56	3.55	4.99	17.68	18.40	3.64	12.72	4.43	6.76
0.950	9.06	10.72	19.90	20.98	3.97	5.68	18.09	19.66	0.32	6.64	5.35	7.90
0.900	3.96	10.74	18.72	20.14	3.90	5.51	17.10	17.84	2.55	13.19	6.21	8.84
0.850	5.67	11.32	20.04	21.35	3.88	5.21	18.15	19.97	1.45	9.08	6.74	8.57
0.800	5.15	10.59	19.37	21.04	4.93	6.41	19.25	20.86	2.72	14.09	8.83	11.02
0.700	6.32	11.08	24.24	26.26	5.22	6.28	29.24	30.01	0.60	8.69	9.56	11.57
0.600	19.01	20.00	19.00	20.80	4.44	5.44	19.57	20.26	4.21	15.23	12.90	15.48
0.500	10.83	12.28	21.09	22.31	4.65	5.69	29.89	30.42	4.07	10.92	11.85	13.68
0.000	71.80	71.00	45.78	45.36	58.48	58.65	33.00	34.75	36.62	36.84	49.38	50.01

known quality.

Table 5.5 and Figure 5.3 detail these results. The first column indicates the approximation factor used in training each model for each dataset. The remaining columns show train and test performance using exact inference.

What is promising is that test performance does not drop precipitously as we use increasingly worse approximations. For most problems, the performance remains reasonable even for $\rho = 0.9$.

5.6 Conclusion

This chapter theoretically and empirically analyzed two classes of methods for training structural SVMs on models where exact inference is intractable. Focusing on completely connected Markov random fields, we explored how greedy search, loopy belief propagation, a linear-programming relaxation, and graph-cuts can be used as approximate separation oracles in structural SVM training. In addition to a theoretical comparison of the resulting algorithms, we empirically compared performance on multi-label classification problems. Relaxation approximations distinguish themselves as preserving key theoretical properties of structural SVMs, as well as learning robust predictive models. Most significantly, structural SVMs appear to train models to avoid relaxed inference methods' tendency to yield fractional, ambiguous solutions.

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This chapter will present the conclusions of this work. Further, with this work done, there are still many unanswered questions, mysteries, and potential for future work with supervised clustering and structural SVMs, and so after the conclusions we briefly present some of the more interesting ideas that could be developed in future work.

6.1 Conclusions

As argued in Chapter 1, many tasks involve partitioning a given item set into related groups. For example, automated news aggregators group news articles which are about the same story. Noun-phrase coreference systems group a document's noun-phrases which refer to the same entity. In image segmentation, one identifies regions of the image corresponding to the same object. A common practice in these problems and others like them is to employ clustering techniques to find related groups in sets of items. Since manually tuning clustering algorithms to solve these problems is difficult, the common approach is to employ supervised machine learning techniques to learn how to partition other item sets of the same type, learning how to cluster item sets $\mathbf{x}$ based on example clusterings $\mathbf{y}$. *Supervised clustering* is the problem of tuning clustering algorithms using supervised learning so they perform well on a task of interest to the practitioner.

How can we learn a clustering function? The goal of nearly all popular clustering methods is to find the clustering maximizing some criteria $f(\mathbf{x}, \mathbf{y})$, where $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}$ is commonly called a discriminant function. This discriminant

function is typically some formula involving the pairwise similarity between pairs of items; for example, in correlation clustering $f(\mathbf{x}, \mathbf{y})$ it is the sum of all pairwise similarities between items $x_i, x_j \in \mathbf{x}$ in the same clustering in $\mathbf{y}$, and in k-means the sum of similarities of each item to its cluster's center. For this reason, nearly all supervised clustering methods learn the item pair similarity measure, thus affecting which clustering $\mathbf{y}$ will maximize $f(\mathbf{x}, \mathbf{y})$.

In this sense, one may view supervised clustering as a metric or similarity learning task. We argued, however, that general metric learning frameworks are insufficient, since they do not learn metrics optimized for clustering performance. All the different existing clustering methods would group items in different ways (whether it be k-means, spectral, correlation, single-link, complete-link, average-link, etc., clustering) even over the same pairwise similarity measure, so it is critical that the similarity measure be learned in such a fashion so that the cluster method in question performs well for the task at hand. Other methods might wind up finding parameterizations optimized to the wrong criteria as argued in Section 1.3 and Section 1.7.

In order to learn this parameterization for our clustering, we employed a structural SVM learning algorithm, which we described in Chapter 2 as a general method for learning parameterizations of functions with complex structured inputs and outputs. With a training set, the structural SVM learning method's goal is to find a parameterization such that the discriminant function is maximized for the correct output, versus all possible incorrect outputs. Violations of this are punished proportionately to each incorrect output's "loss" relative to the correct output, where loss is a sort of judgment function. Since different tasks may have different loss functions, structural SVMs have the ability to learn parameteriza-

tions optimized for specific tasks, an important distinction between the proposed supervised clustering method and those already existent in the literature.

We then derived supervised clustering methods for correlation clustering in Chapter 3 and k-means/spectral clustering in Chapter 4. In particular, we empirically demonstrated the method's usefulness in being able to optimize to a task specific loss function, its computational efficiency, and its ability to learn parameterizations of various clustering methods.

Since correlation and k-means style clusterings require the use of approximations to maximize their discriminant function, and structural SVMs incorporate the predictive method into the learning program, the learning method itself becomes approximate. We presented a detailed empirical and theoretical analysis of the use of approximations and structural SVMs in Chapter 5. In short, though some of the theoretical guarantees of the structural SVM learning algorithm no longer hold, we can make new statements for undergenerating approximations (based on some type of local maximization) and overgenerating approximations (based on some type of relaxation). In particular, when using ρ-approximate undergenerating approximations in structural SVMs, the extent to which the original theoretical guarantees are violated can be bounded. When using overgenerating approximations, the important theoretical guarantees hold at the cost of possible suboptimality of the structural SVM parameterization.

6.2 Agglomerative Clustering with Structural SVMs

While we have presented methods for learning parameterizations for correlation and k-means/spectral clustering, there are many other types of popular cluster-

ing algorithms that may be more appropriate for some tasks. One of the more popular of these methods are agglomerative methods including single link, complete link, and average link clustering. However, there are problems that prevent a straightforward method of learning parameterizations of these methods.

Consider the case of single link clustering. For a given set of items $\mathbf{x}$ with a model parameterization $\mathbf{w}$, suppose we infer a similarity matrix K where $K_{ij} = \langle \mathbf{w}, \psi_{ij} \rangle$. Classic single link clustering would, starting from a clustering where each item was in its own cluster, repeatedly find the two items $x_i, x_j \in \mathbf{x}$ not yet in the same cluster with maximum similarity, join them, and return the resulting dendrogram [55, 72]. The algorithm is similar to finding the maximum spanning tree in a completely connected graph with nodes corresponding to $\mathbf{x}$'s items and edge weights defined by K. Consider the following variant of the classic single link clustering algorithm, which differs insofar as this produces a simple partitioning (not a dendrogram), and it stops when there is no merge that does not involve a negative similarity in K. This algorithm is shown in Algorithm 8. In this proce-

(SINGLE LINK CLUSTERING)

Input: An input set of items $\mathbf{x}$ inferring similarity matrix K
$\mathbf{y} \leftarrow \{\{x_i\} : x_i \in \mathbf{x}\}$
let $\textsc{Merge}(\mathbf{y}, y, y') \equiv (\mathbf{y} \setminus \{y, y'\}) \cup \{y \cup y'\}$
let $\textsc{FindCluster}(\mathbf{y}, x) \equiv y \in \mathbf{y}$ such that $x \in y$
repeat
  $x_{\bar{i}}, x_{\bar{j}} \leftarrow \text{argmax}_{x_i, x_j \in \mathbf{x} : \textsc{FindCluster}(\mathbf{y}, x_i) \neq \textsc{FindCluster}(\mathbf{y}, x_j)} K_{ij}$
  if $K_{ij} \geq 0$ **then**
    $\mathbf{y} \leftarrow \textsc{Merge}(\mathbf{y}, \textsc{FindCluster}(\mathbf{y}, x_i), \textsc{FindCluster}(\mathbf{y}, x_j))$
  end if
until $\mathbf{y}$ has not changed during an iteration, or $|\mathbf{y}| = 1$
return $\mathbf{y}$

Algorithm 8: The variant single link clustering algorithm.

dure, the $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ combined feature function will be the sum of pairwise similarity

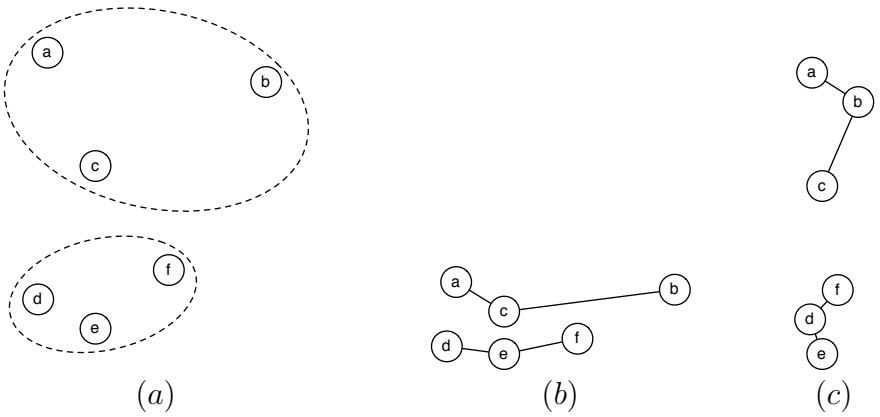

Figure 6.1: A set of six items with its partitioning, including the optimal single link partitioning in one dimensional distortion versus another distortion.

vectors ψ_{ij} corresponding to those x_i, x_j selected for merging in Algorithm 8 in the construction of $\mathbf{y}$ given set $\mathbf{x}$. The problem with this scheme is that those pairs included in $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ will depend upon $\mathbf{w}$.

Consider Figure 6.1, where Figure 6.1(a) shows a set of six items $\mathbf{x} = \{a, b, c, d, e, f\}$ with partitioning $\mathbf{y} = \{\{a, b, c\}, \{d, e, f\}\}$. Further, suppose that the pairwise feature vectors ψ_{ij} contain two features corresponding to the two dimensions in which the points of $\mathbf{x}$ are shown: one feature is the horizontal displacement of the two points raised to the -2 power, and the other is similar but for vertical displacement. The sum of the two would therefore be one over the square of the Euclidean distance. By changing the corresponding weights in $\mathbf{w}$, we effectively change the importance of the dimension in calculating the similarity or distance, e.g., effectively shrink or expand one dimension as one raises or lowers the corresponding weight in $\mathbf{w}$, respectively. For example, when one increases the weight in $\mathbf{w}$ corresponding to the horizontal feature, the horizontal *similarity* is raised, corresponding to the horizontal *distance* shrinking.

In Figure 6.1(b) we see a distorted version of the Euclidean space in which the points of $\mathbf{x}$ lie, corresponding to a $\mathbf{w}$ where the weight corresponding to the vertical dimension is high relative to the horizontal feature weight. Under such a $\mathbf{w}$, the combined feature function has a value $\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \psi_{ac} + \psi_{bc} + \psi_{de} + \psi_{ef}$. However, consider the distorted space shown in Figure 6.1(c), corresponding to a $\mathbf{w}$ where the horizontal feature weight is raised. Under that $\mathbf{w}$, the combined feature function has a value $\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \psi_{ab} + \psi_{bc} + \psi_{de} + \psi_{df}$. So, for learning single link clustering with structural SVMs, the evaluation of a function $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ requires that $\mathbf{y}$ must not only contain the partitioning of $\mathbf{x}$, but also which pairs of items $x_i, x_j \in \mathbf{x}$ were joined to lead to that partitioning.

The trouble is that many applications that might make use of an algorithm like Algorithm 8 might not care about which items were joined, but just whether the final partition was correct [78]. Without this link structure, it is impossible to evaluate $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ sensibly since the $\mathbf{\Psi}$ function does not have access to $\mathbf{w}$. An alternate formulation of structural SVMs may allow us to learn the latent link structure required by single link clustering and other agglomerative methods, which would fall prey to similar problems.

6.3 Non-smooth Loss and Margin-Scaled Structural SVMs

There is a subtle point that arises when using margin-scaled structural SVMs optimized over a relatively non-smooth loss function Δ, i.e., a Δ function which sometimes has very high $\Delta(\mathbf{y}, \mathbf{y}')$ for two outputs $\mathbf{y}$ and $\mathbf{y}'$ which have relatively close $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ and $\mathbf{\Psi}(\mathbf{x}, \mathbf{y}')$ combined feature functions.

Imagine we are learning a model for the noun-phrase coreference task, using the

MITRE loss Δ_M of Section 3.3.2, and the margin-scaling structural SVM of OP 4. Imagine a noun-phrase coreference learning task where there is a single training example $(\mathbf{x}_0, \mathbf{y}_0)$, where there is only one training example. The first two noun-phrases in $\mathbf{x}_0$, i.e., $x_1, x_2 \in \mathbf{x}_0$, are joined in $\mathbf{y}_0$, and all other noun-phrases are in their own individual cluster in $\mathbf{y}_0$, i.e., $\mathbf{y}_0 = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \ldots, \{x_{|\mathbf{x}_0|}\}\}$, with the first two coreferent and all others non-coreferent. Such an example is not actually that contrived; while no noun-phrase coreference labeling is this sparse, the cluster structure is still rather sparse.

Now consider the hypothetical "wrong output" $\mathbf{y}$ with all noun-phrases non-coreferent, $\mathbf{y}_0 = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \ldots, \{x_{|\mathbf{x}_0|}\}\}$. Then, $\boldsymbol{\Psi}(\mathbf{x}_0, \mathbf{y}_0) - \boldsymbol{\Psi}(\mathbf{x}_0, \mathbf{y}) = \frac{1}{|\mathbf{x}_0|^2}\psi_{12}$, where ψ_{12} is pairwse feature vector between $x_1, x_2 \in \mathbf{x}$, and $\Delta_M(\mathbf{y}_0, \mathbf{y}) = 100$. This leads to the constraint of (2.10) of OP 4 corresponding to $\mathbf{y}$ taking this form:

$$\left\langle \mathbf{w}, \frac{1}{|\mathbf{x}_0|^2}\psi_{12} \right\rangle \geq 100 - \xi_0. \tag{6.1}$$

In order to satisfy this constraint for $\xi_0 = 0$, that is, without using any slack, we would need a $\mathbf{w}$ with length at least

$$\|\mathbf{w}\|_2 \geq \frac{100 \cdot |\mathbf{x}_0|^2}{\|\psi_{12}\|_2}. \tag{6.2}$$

The structural SVM objective function (2.8), which is

$$\min_{\mathbf{w}, \boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\xi_0 \tag{6.3}$$

Now imagine the situation after the first iteration of Algorithm 1, when $\mathbf{y}$ is the first constraint returned by the separation oracle (which it will be, since $\Delta(\mathbf{y}_0, \mathbf{y}) = 100$ and no other output has so high a loss), and consequently the constraint (6.1) is the only one in the working set. In such a situation, the $\mathbf{w}$ which best enforces the margin on this single constraint while being of minimum length is some multiple

of ψ_{12}, which we term $\mathbf{w} = \beta\psi_{12}$ for some scalar multiple β. Further, since (6.1) is the only constraint, at the optimum solution to the (6.3) objective it is an equality, so the slack $\xi_0 = 100 - \beta\frac{\|\psi_{12}\|_2^2}{|\mathbf{x}_0|^2}$. The objective (6.3) then becomes

$$\min_{\beta} \frac{1}{2}\beta^2\|\psi_{12}\|_2^2 + 100C - C\beta\frac{\|\psi_{12}\|_2^2}{|\mathbf{x}_0|^2} \tag{6.4}$$

This reaches its minimum at $\beta = \frac{C}{|\mathbf{x}_0|^2}$, with required slack $\xi_0 = 100 - \frac{C\|\psi_{12}\|_2^2}{|\mathbf{x}_0|^4}$.

In effect, we have a constraint requiring that $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \mathbf{y}_0)\rangle$ and $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}_0, \mathbf{y})\rangle$, where the two $\mathbf{\Psi}$ are very close relative to the $\mathbf{\Psi}(\mathbf{x}_0, \mathbf{y}')$ of other outputs $\mathbf{y}'$. This constraint will effectively wind up "dominating" the others. If C is set too low, the ξ_0 will be set so high on account of this constraint that few other wrong outputs $\mathbf{y}'$ for $\mathbf{x}_0$ will have any chance of being introduced as constraints – effectively, we will learn from only one constraint. If C is set high enough to the point ξ_0 is lowered, so that other constraints do influence the problem, we run the risk of overfitting. The entire process is held hostage on account of this single wrong output $\mathbf{y}$.

This was an issue when learning coreference models optimized for MITRE loss Δ_M in the coreference experiments of Chapter 3: the C value had to be set very high in order to learn a meaningful model.

Problems of this sort are mitigated when one moves from the margin scaling model to the slack scaling model of OP 5. In slack scaling, the slack term is scaled by the loss function. All constraints require an equal margin, and only if the constraint is violated would the loss come into play, making it harder for the learning algorithm to "cheat" on that constraint by requiring proportionally more slack. It becomes harder for a single wrong output which just happens to have a very high Δ to hijack the process.

In practice, slack formulations are hardly used, owing to the separation oracle

(2.22) being harder to optimize. However, informal experiments with slack scaled learning with MITRE loss with a separation oracle based on local search yielded models with superior performance. Recent work has improved the ability to optimize slack scaled structural SVMs [94], techniques which could be extended to clustering to improve performance of optimizing to non-smooth loss functions like MITRE loss.

6.4 Nonlinear Parameterization for Clustering

The preceding discussion of parameterizations of correlation clustering and k-means clustering have not explicitly considered $\mathbf{w}$, and in actual application we have kept the parameterization as effectively a real vector $\mathbf{w} \in \mathbb{R}^N$. However, $\mathbf{w}$ may also be considered a non-linear parameterization vector by considering the duals of the structural SVM optimization problems shown in OP 4 and OP 5 of Section 2.1.1.

Optimization Problem 12. (Margin-Scaled Structural SVM Dual QP)

$$\max_{\boldsymbol{\alpha}} -\frac{1}{2} \sum_{\substack{i=1..n \\ \mathbf{y} \in \mathcal{Y}}} \sum_{\substack{j=1..n \\ \mathbf{y} \in \mathcal{Y}}} \alpha_{i,\mathbf{y}} \alpha_{j,\bar{\mathbf{y}}} \left\langle \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i) - \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}), \boldsymbol{\Psi}(\mathbf{x}_j, \mathbf{y}_j) - \boldsymbol{\Psi}(\mathbf{x}_j, \bar{\mathbf{y}}) \right\rangle + \sum_{\substack{i=1..n \\ \mathbf{y} \in \mathcal{Y}}} \alpha_{i,\bar{y}} \Delta(\mathbf{y}_i, \mathbf{y}) \tag{6.5}$$

$$s.t.\ \forall i, \mathbf{y} \quad : \quad \alpha_{i,\mathbf{y}} \geq 0 \tag{6.6}$$

Optimization Problem 13. (Slack-Scaled Structural SVM Dual QP)

$$\max_{\boldsymbol{\alpha}} -\frac{1}{2} \sum_{\substack{i=1..n \\ \mathbf{y} \in \mathcal{Y}}} \sum_{\substack{j=1..n \\ \mathbf{y} \in \mathcal{Y}}} \alpha_{i,\mathbf{y}} \alpha_{j,\bar{\mathbf{y}}} \left\langle \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i) - \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}), \boldsymbol{\Psi}(\mathbf{x}_j, \mathbf{y}_j) - \boldsymbol{\Psi}(\mathbf{x}_j, \bar{\mathbf{y}}) \right\rangle + \sum_{\substack{i=1..n \\ \mathbf{y} \in \mathcal{Y}}} \alpha_{i,\bar{y}} \tag{6.7}$$

$$s.t.\ \forall i, \mathbf{y} \quad : \quad \alpha_{i,\mathbf{y}} \geq 0 \tag{6.8}$$

$$\forall i \quad : \quad \sum_{\mathbf{y}} \frac{\alpha_{i,\mathbf{y}}}{\Delta(\mathbf{y}_i, \mathbf{y})} \leq \frac{C}{n} \tag{6.9}$$

In any structural SVM learning problem of the form presented in Section 2.1.1, we may view the parameterization $\mathbf{w}$ learned in the primals of OP 4 and OP 5 as a weighted sum of the $\boldsymbol{\Psi}$ combined feature vectors seen in training:

$$\mathbf{w} = \sum_{\substack{i=1..n \\ \mathbf{y} \in \mathcal{Y}}} \alpha_{i,\mathbf{y}} \left(\boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}_i) - \boldsymbol{\Psi}(\mathbf{x}_i, \mathbf{y}) \right). \tag{6.10}$$

For solutions obtained through Algorithm 1, this $\mathbf{y} \in \mathcal{Y}$ is actually limited to $\mathbf{y} \in S_i$, i.e., the working set of the i-th example, since there cannot be non-zero dual variables $\alpha_{i,\mathbf{y}}$ for outputs $\mathbf{y}$ outside of the working set S_i.

In the case of supervised clustering, whether it be the correlation clustering $\boldsymbol{\Psi}$ functions of (3.8) or (3.18), or the k-means/spectral $\boldsymbol{\Psi}$ functions of (4.13) or (4.15), these $\boldsymbol{\Psi}(\mathbf{x}, \mathbf{y})$ combined feature functions always take the form of some sort of weighted sum of the pairwise feature vectors between pairs of items in $\mathbf{x}$, i.e.,

$$\boldsymbol{\Psi}(\mathbf{x}, \mathbf{y}) = \sum_{x_i, x_j \in \mathbf{x}} \beta_{i,j,\mathbf{y}} \psi_{x_i, x_j} \tag{6.11}$$

where this $\beta_{i,j,\mathbf{y}} \in \mathbb{R}$ coefficient is some number depending upon $\mathbf{y}$ and whether this $\boldsymbol{\Psi}$ was calculated for learning correlation or k-means clustering. This β number is readily available from the appropriate $\boldsymbol{\Psi}(\mathbf{x}, \mathbf{y})$ equation. Let us be explicit. For $\boldsymbol{\Psi}(\mathbf{x}, \mathbf{y})$ in (3.8), $\beta_{i,j,\mathbf{y}} = 1/|\mathbf{x}|^2$ if x_i, x_j are in the same cluster in $\mathbf{y}$, and $\beta_{i,j,\mathbf{y}} = 0$

if x_i, x_j are in different clusters in $\mathbf{y}$. For $\mathbf{\Psi}(\mathbf{x}, \mathbf{e})$ in (3.18), $\beta_{i,j,\mathbf{y}} = e_{i,j}/|\mathbf{x}|^2$. For $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ in (4.13), $\beta_{i,j,\mathbf{y}} = 1/|c|$ where c is the cluster both x_i, x_j appear in in $\mathbf{y}$, or $\beta_{i,j,\mathbf{y}} = 0$ if x_i, x_j are in different clusters in $\mathbf{y}$. For $\mathbf{\Psi}(\mathbf{x}, \mathbf{Y})$ in (4.15), $\beta_{i,j,\mathbf{Y}} = \mathbf{Y}_{i,:}^T \mathbf{Y}_{j,:}$.

Combining (6.10) and (6.11), we can characterize $\mathbf{w}$ in a similar fashion.

$$\mathbf{w} = \sum_{\substack{i=1..n \\ x_j, x_\ell \in \mathbf{x}_i}} \gamma_{i,j,\ell} \psi_{x_j, x_\ell} \tag{6.12}$$

To be explicit, $\gamma_{i,j,\ell} = \sum_{\mathbf{y} \in \mathcal{Y}} \alpha_{i,\mathbf{y}} (\beta_{i,j,\mathbf{y}_i} - \beta_{i,j,\mathbf{y}})$. Again, for solutions obtained through Algorithm 1, this $\mathbf{y} \in \mathcal{Y}$ is limited to $\mathbf{y} \in S_i$.

For both correlation clustering and k-means, the pairwise similarity score K_{ij} in this work is always a product between the parameterization $\mathbf{w}$ and pairwise feature vector $\psi_{i,j}$, i.e., $K_{ij} = \langle \mathbf{w}, \psi_{x_i, x_j} \rangle$. This allows us to characterize the pairwise similarity score for an item pair $\mathbf{x}_j, \mathbf{x}_\ell \in \mathbf{x}$ more explicitly as

$$K_{j,\ell} = \sum_{\substack{i=1..n \\ x_{\hat{\jmath}}, x_{\hat{\ell}} \in \mathbf{x}_i}} \gamma_{i,\hat{\jmath},\hat{\ell}} \left\langle \psi_{x_j, x_\ell}, \psi_{x_{\hat{\jmath}}, x_{\hat{\ell}}} \right\rangle \tag{6.13}$$

Here, this inner product $\langle \cdot, \cdot \rangle$ can be replaced with some sort of kernel function evaluation $\kappa(\cdot, \cdot)$. An example use would be to use a degree-2 polynomial kernel. As seen in (4.3), the pairwise similarity $K_{i,j}$ can be understood as $\psi_i^T \operatorname{diag}(\mathbf{w}) \psi_j$. These ψ_i and ψ_j vectors do not necessarily, and in many applications do not, explicitly exist. If one instead wished to learn a full matrix parameterized inner product $\psi_i^T W \psi_j$ for a matrix $W \in \mathbb{R}^{m \times m}$, one could implicitly learn W by using a degree-2 polynomial kernel for κ. (However, it would probably be faster to just map the ψ_i feature vectors explicitly into a quadratic space with $\hat{\psi}_i$, so that $\psi_i^T W \psi_j \equiv \hat{\psi}_i^T \operatorname{diag}(\mathbf{w}) \hat{\psi}_j$, where $\mathbf{w}$ is the linearization of W.)

The difficulty with such a scheme is that all of these kernel evaluations would

be awfully inefficient. Clustering a new data set would require first calculating the similarity matrix K, and each entry would require as many kernel evaluations as there were pairs of items in each set of items $\mathbf{x}_i$ in all n training examples used to train the model, instead of the very simple linear inner product $K_{ij} = \langle \mathbf{w}, \psi_{ij} \rangle$. Fortunately, approximate kernel methods with strong theoretical guarantees and empirical results for structural SVMs have been developed which could be easily applied in this situation [113].

APPENDIX A

SVM^{PYTHON}: WRITING STRUCTURAL SVMS IN PURE PYTHON

This appendix describes my SVM^{python} framework [39]. SVM^{python} is a variation and extension of the SVM^{struct} software framework written by Thorsten Joachims. With SVM^{struct}, one may design a structured machine learning method as described in Section 2.1 by implementing a few functions in C, encapsulating task specific procedures. SVM^{python} allows the same functionality, but allows the developer to write the extension functions in pure Python. By doing so, SVM^{python} takes advantage of many of the features of Python, allowing for far more rapid implementation of ideas than would be possible under C.

To understand this document, one must have a basic understanding of the Python programming language. In particular, one should be able to understand Python code. Further, one should understand the material of Section 2.1, and in particular the cutting plane algorithm of Algorithm 1.

This is not a reference guide to SVM^{python}. The SVM^{python} distribution[1] contains its own documentation and reference. This document instead explains SVM^{python}. It first explains motivation for SVM^{python} by comparing and contrasting it with the SVM^{struct} package upon which it is based. It closes with a step by step example of building a binary classifier in SVM^{python} that makes use of nearly all of the important functionality of SVM^{python}. After reading this, one should understand whether to use SVM^{python} or SVM^{struct}, and be comfortable writing one's own structural SVM learning framework with SVM^{python}.

[1] Downloadable at `http://www.cs.cornell.edu/~tomf/svmpython2/` .

A.1 The Underlying SVM^{struct} Framework

The goal of SVM^{python} is to provide a framework as powerful as SVM^{struct}, except with allowing a developer to write in Python rather than C. Since they provide identical functionality, a developer has a choice of whether to use SVM^{python} or SVM^{struct}, and so a detailed understanding of both frameworks, and the relationship between them, is critical. We begin by explaining SVM^{struct}, calling attention to the fact that most of what is said about SVM^{struct} is true about SVM^{python}.

The SVM^{struct} framework is an implementation of the structural support vector machine, a cutting plane algorithm described in Section 2.1. The implementation exploits the fact that, independent of whatever problem is being addressed with the structural SVM, implementations of learning methods based on Algorithm 1 would share a tremendous amount of code between them. There are only a few task dependent pieces of Algorithm 1:

1. The combined feature function $\boldsymbol{\Psi}(\mathbf{x}, \mathbf{y})$.
2. The loss function $\Delta(\mathbf{y}, \hat{\mathbf{y}})$.
3. The separation oracle function, $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} H(\mathbf{y})$, with H defined in (2.21) and (2.22) for margin and slack loss scaling, respectively. The separation oracle finds the output $\hat{\mathbf{y}}$ associated with the most violated constraint for a given example $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{S}$ and current model parameterization $\mathbf{w}$.
4. The prediction function, $h_{\mathbf{w}}(\mathbf{x})$, with $h_{\mathbf{w}}(\mathbf{x}) \equiv \text{argmax}_{\mathbf{y} \in \mathcal{Y}} \langle \mathbf{w}, \boldsymbol{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$. Technically, this is not part of Algorithm 1 since that algorithm concerns only learning a model parameterization $\mathbf{w}$, but presumably, one would learn a model without intending to use it in prediction.

The SVM^{struct} framework performs all the non-task specific operations of Algorithm 1, and calls some hook functions to fill in the task specific material. These hook functions are implemented by the developer in C. The internal SVM^{struct} code is completely indifferent as to the nature of the inputs $\mathbf{x} \in \mathcal{X}$, the outputs $\mathbf{y} \in \mathcal{Y}$, the loss Δ, and the algorithms used for the separation oracle and prediction function.

This is worth emphasizing, since lack of understanding of the black box nature of the developer-provided hook functions is the source of most of the major misunderstandings about the implementation of the structural SVM: SVM^{struct} treats the developer hook functions as a complete black box, from which it extracts only a few standard data structures (notably the Ψ and Δ functions). It is totally agnostic to the form of the inputs $\mathbf{x}$, outputs $\mathbf{y}$, whatever algorithms are used in prediction and constraint inference. With this is a general purpose learning framework which can be used to implement a structural SVM learner for a wide variety of tasks.

Of course, there is far more involved in successfully leveraging SVM^{struct} than implementing four functions. The SVM^{struct} framework consists of two executables: a learner which takes a training set and outputs a model parameterization, and a predictor which takes a set of inputs $\mathbf{x}$ and a model parameterization, and outputs predictions $\mathbf{y}$ for each input $\mathbf{x}$. Practically, there are many other steps that must be accomplished:

1. The data structures holding an input $\mathbf{x}$ and output $\mathbf{y}$ must be defined in the structures `PATTERN` and `LABEL`, respectively.
2. In both training and prediction, it is necessary to read the training set or evaluation set $\mathcal{S} = \{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_n, \mathbf{y}_n)\}$.

3. If learning a linear parameterization $\mathbf{w}$, the most common case, one must at least set the number of linear parameters to learn.
4. The model and its learned parameters must be serialized once the model is learned, and subsequently deserialized when it comes time to make predictions with that learned model, as separate executables are built for training a model and predicting using that model.
5. Predictions $\mathbf{y} = h_{\mathbf{w}}(\mathbf{x})$ must be output to a file in the case of prediction.
6. The structures corresponding to $\mathbf{x}$, $\mathbf{y}$, and the model must be properly deallocated.

From the point of view of the developer, the task of implementing a structured learner with SVM^{struct} involves several steps:

1. Downloading the SVM^{struct} source code[2].
2. Modifying the `svm_struct_api_types.h` file, to add problem specific definitions of the structures `PATTERN` and `LABEL` (corresponding to $\mathbf{x}$ and $\mathbf{y}$ inputs and outputs), probably `STRUCTMODEL` and `STRUCT_LEARN_PARM` since learners may want to have user-specifiable options which will affect the learning process, and perhaps `STRUCT_TEST_STATS` if one wishes to retain and produce a more detailed performance report than average loss over the test set.
3. Modifying the `svm_struct_api.c` file, to fill in the myriad and mostly initially empty functions—28 in all, though some have default behavior perfectly acceptable for most applications—with the desired task specific functionality.

The SVM^{struct} learning and classification executables follow the paths shown in Figure A.1 and Figure A.2, respectively. Boxes indicating a particular process

[2]Available at `http://svmlight.joachims.org/svm_struct.html`.

in the algorithm. The first line holds the name of the function to implement in `svm_struct_api.c`, and subsequent lines describe what the user is intended to accomplish in implementing that function.

SVM^{struct}'s requirement that the developer implement these functions in C leads to some rather basic problems for developers.

1. Performing I/O of highly structured data in C is somewhat involved, especially in situations where the size of your inputs are unknown a priori.
2. The C language does not lend itself to quick prototyping. Research by its very nature almost always involves playing around with a number of different ideas, so barriers to change are undesirable. Through no particular fault of its own, simple changes in an instantiation of SVM^{struct} often require changes in many separate locations in the source code.
3. The minimal C standard library requires that users rely upon external non-standard libraries or their own implementation of even the most basic support algorithms (e.g., hash tables, union-find structures, string processing).

A.2 Introduction to SVM^{python}

The SVM^{python} software package allows one to write these developer interface functions in Python rather than C. What is Python? Python is a very high level interpreted programming language, with dynamic strong duck typing and garbage collection. As an interpreted language, Python is comparable to Perl or Ruby, though it has much simpler language structure than either. Python programs are often written to use a mixture of imperative, object oriented, and functional

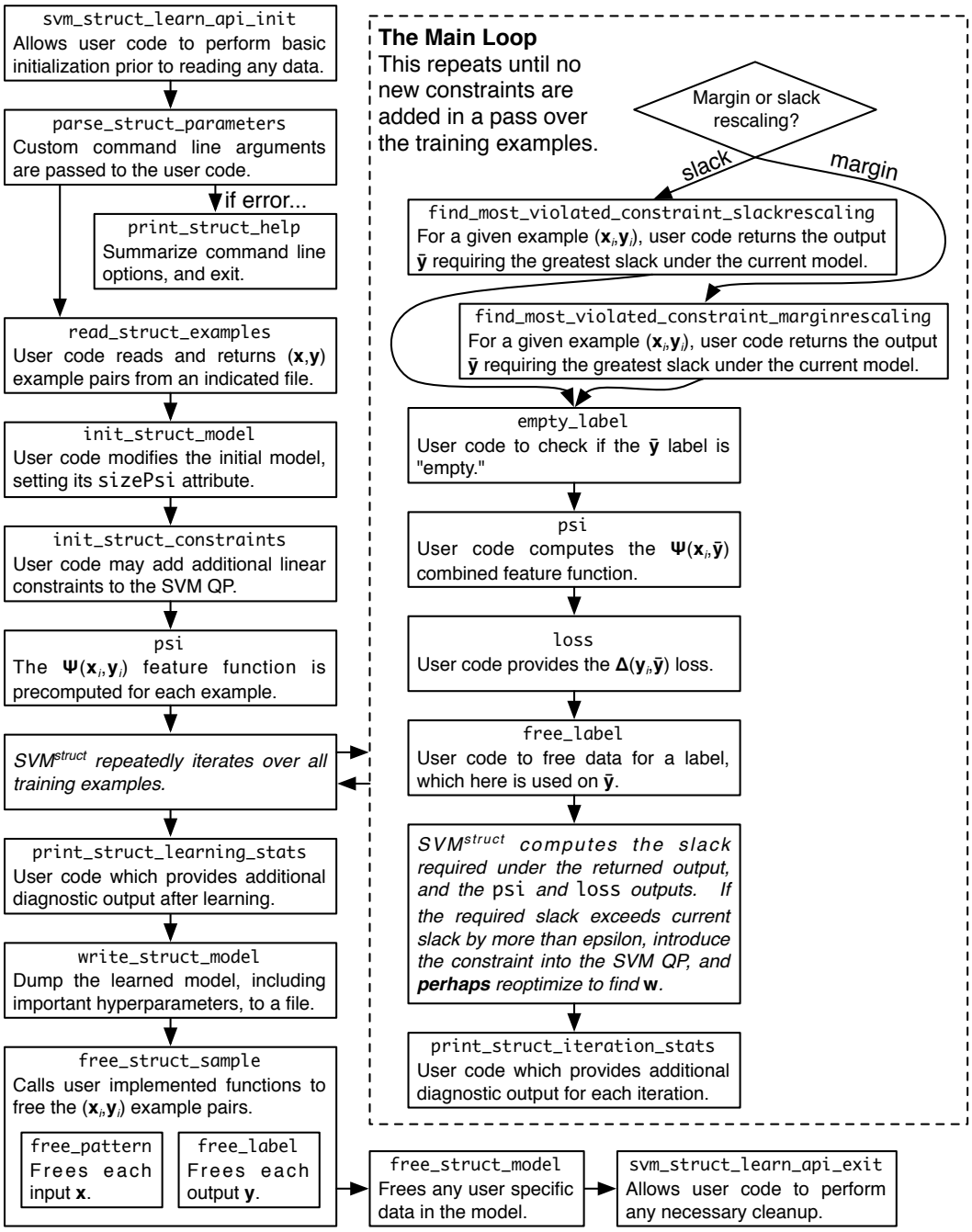

Figure A.1: Flowchart showing the flow of execution within the SVM^{struct} learner, with the flow of execution starting from the upper left. Steps associated with a particular call to a developer's extension function have the box lead with the function name in `svm_struct_api.c`.

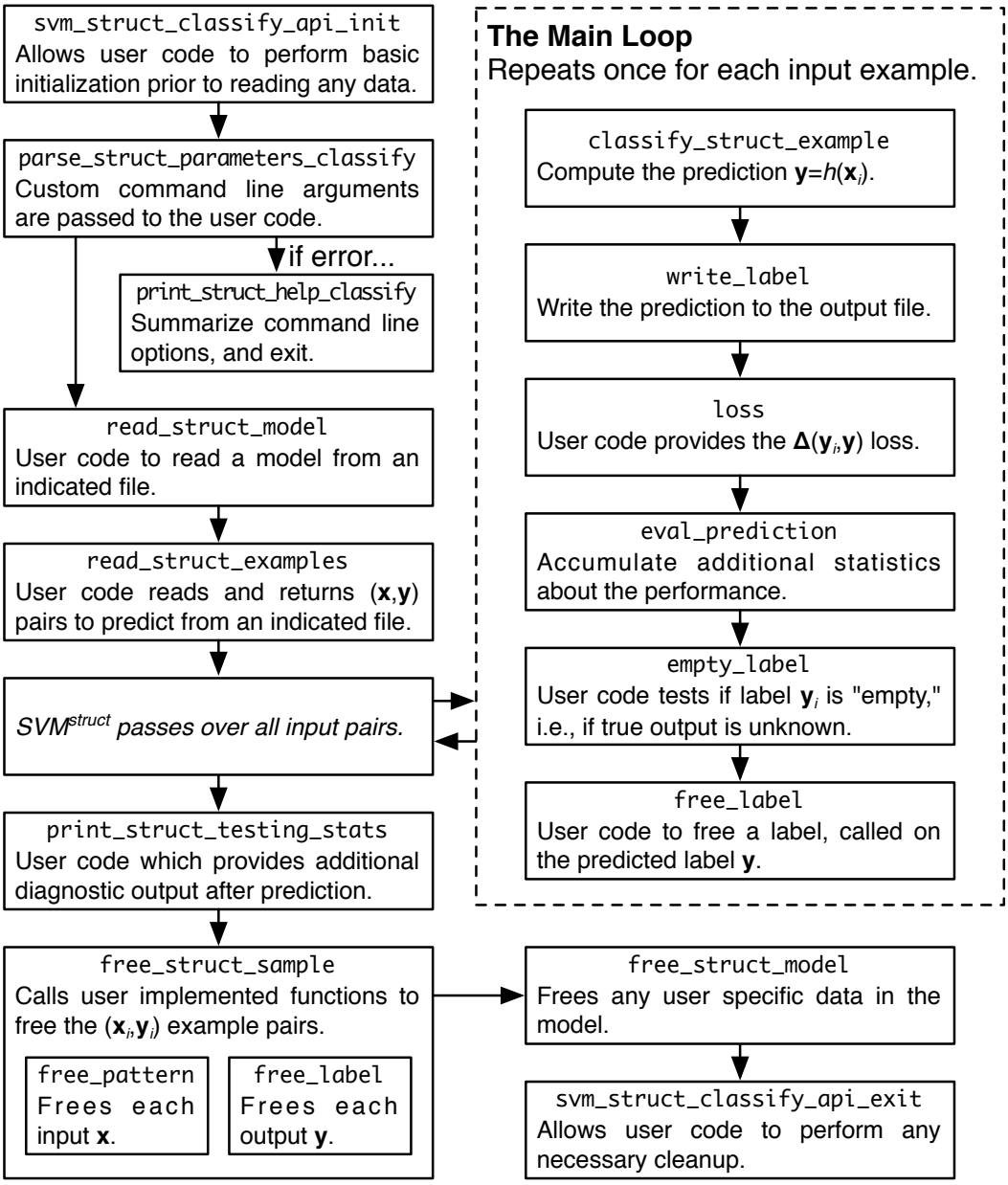

Figure A.2: Flowchart showing the flow of execution within the SVM^{struct} classifier, with the flow of execution starting from the upper left. Steps associated with a particular call to a developer's extension function have the box lead with the function name in `svm_struct_api.c`.

programming paradigms. Python is a general purpose language, well suited to a wide range of applications from simple scripts to large highly structured projects. Supporting this is an extremely large standard library offering a wide range of services. Converse to its rich library, Python's syntax and semantics are very spare and simple to the point of one being able to form a comprehensive understanding of them within a matter of minutes. In addition to these basic traits, there are also many subjective traits of Python which make it attractive: relative to other languages, code in Python is often highly compact, readable, very rapidly developed, and existing code is changed easily. It is difficult to justify these statements with any responsibly produced empirical evidence, but suffice to say, many Python programmers consider them true.

Perhaps most important to this document, however, is Python's relationship with C. Python provides an extensive C API library; the base functionality of the Python interpreter and its core classes are implemented with this API, but other code may make use of the API as well. This API allows C code to be written and compiled such that it is callable by Python code, a process called extending. Conversely, it is also possible for C code to call Python code for programs that want a programmable interface, a process called embedding.

What SVM^{python} is, is SVM^{struct} with embedded Python in the developer hook functions. In other words, the C code which SVM^{struct} intends to be a developer hook function instead calls functions from a Python module. The Python module is loaded at runtime. The underlying SVM^{struct} code is totally unaware and indifferent to the fact that the developer hook functions are instead calling functions defined in a Python module. It is important to note that any operations outside of the developer hooks, including most significantly optimization of the

SVM quadratic program, are totally unmodified and retain the speed of the C implementation.

To take one particular example, the C developer hook function `read_struct_examples` calls the Python module's `read_examples` function. The `read_struct_examples` C function has signature

```
SAMPLE read_struct_examples(char *file, STRUCT_LEARN_PARM *sparm)
```

where a `SAMPLE` item holds an array of $(\mathbf{x}, \mathbf{y})$ example pairs, and the number of examples. The Python function that this function calls within SVM^{python} has signature

```
def read_examples(file, sparm)
```

where the `file` argument is a Python string, and the `sparm` element is a special type provided by SVM^{python} called `Sparm`. This function is expected to return a sequence containing two-element tuples, consisting of an input $\mathbf{x}_i$ and output $\mathbf{y}_i$, so that the i-th element of the returned sequence holds the training example $(\mathbf{x}_i, \mathbf{y}_i)$. These inputs and outputs can be any Python objects whatsoever.

One special case is that if an output $\mathbf{y}_i$ is the Python `None` object, then that corresponds to the output being unknown. This is useful, for example, during classification, when sometimes we really do not know the "right" answer for our inputs. So, do not use `None` as your output unless this is the desired result.

By using SVM^{python}, one also inherits Python's weaknesses, most notably its slow execution speed. While enabling rapid development, being an interpreted language leads to a corresponding slowdown in runtime. In my own experience, competently written C tends to run at about a tenth of the time of competently

written Python. The typical approach within SVM^{python} development is to prototype a design in pure Python, and then move the most computationally intensive pieces to C code.

For the benefit of the developer's Python module, SVM^{python} provides a Python module named `svmapi`, within this are all of the relevant datatypes and functions from the C code. For example, the `Sparm` Python type seen above corresponds to the `STRUCT_LEARN_PARM` C type, `Sparse` corresponds to the `SVECTOR` C type, and so on. Further, many SVM^{struct} provided utility functions have analogs exposed in the `svmapi` module: the `classify_example` C function is exposed through the `Model.classify` method, the `create_svector` function has its functionality exposed through the constructor for `Sparse` instances, etc.

A.3 Default Behavior, and Model Persistence

Many of the Python analogues to their C hook functions must be implemented or the module will not function (for example, `psi` and `classify_example`). However, owing to some advantages of Python language, some of the functions, if not implemented, have default behavior which will be acceptable in a wide variety of circumstances.

To take one example, the C hook functions for memory management (`free_pattern`, `free_label`, `free_struct_model`, and `free_struct_sample`) have no analogue in Python, owing to reliance upon Python's garbage-collected memory management scheme. The function `write_label` for the output of labels $\mathbf{y}$ defaults to outputting the Python object's string representation to the file, which may be acceptable in some circumstances.

However, one of the most useful features of SVM^{python} and the use of default implementation of functions is its automatic serialization and deserialization of the learned model.

The model, represented in C through a `STRUCTMODEL` struct, and in Python through a `StructModel` class instance, is the main output of the learner. The learned model parameterization contains not only the learned $\mathbf{w}$, but also commonly additional hyperparameters affecting how both learning and classification are performed. For example, across many applications, it is common to have command line options to change how features are induced from the input data, or to allow choice among different styles of normalization, or to provide other more task specific hyperparameters. It is also common for hyperparameters to be determined from the training data, e.g., the number of features of various types, or in the supervised k-means clusterer the value of k.

In implementation, in SVM^{struct} under C, such an addition would require several modifications concerning its declaration and usage, naturally. However, because this is an object that is shared between two processes, we must also properly read and write all hyperparameters to and from the model file within the `write_struct_model` and `read_struct_model` files, respectively, which we call serialization and deserialization.

In C, a hyperparameter in a model has a lifecycle consisting of five or six phases: declaration, setting, usage, serialization, deserialization, deallocation. More specifically, these phases are: declaration of the hyperparameter in `STRUCTMODEL`, setting the hyperparameter either by parsing the relevant command line option or analyzing the training data, usage of the hyperparameter value in code, serialization of the hyperparameter to the model file, the deserialization from the model file,

and possibly the deallocation if the hyperparameter if it is stored in dynamically allocated memory outside of the `STRUCTMODEL` block.

Unfortunately, all of these phases occur in different places throughout the code: declaration is in the `STRUCTMODEL` structure, setting is in the `init_struct_model` hook function, usage is obviously wherever the hyperparameter is meant to control behavior, serialization is in `write_struct_model`, deserialization is in `read_struct_model`, and the possible deallocation is in `free_struct_model`.

Within SVM^{python}, by relying upon Python objects we obviate not only the problems of declaration and memory management, but also the problems of serialization and deserialization, through the use of "pickling."

Pickling is Python's primary serialization procedure. Serialization is the process of converting an object to data which can be later recovered, through deserialization, to an accurate clone of the original object. It is typically used in situations where two or more processes need access to the same object but, for whatever reason, it is infeasible for these processes to share the same address space and directly address the same object. This typically happens in network transmission or in situations when the processes are run during different times.

Through the use of Python's `pickle` or `cPickle` modules, an object is transformed into a bytes array and optionally written to a file or port, or transformed into a string. Pickling and unpickling describe the serialization and deserialization of an object, whereas picklable and unpicklable describes those objects which can and cannot be pickled, respectively. Without going into details, the majority of Python objects are picklable, as are those picklable objects containing picklable objects, and so on. Further, many of the types declared in `svmapi` have been

implemented to be picklable, including `StructModel`.

A.4 Flow of Control in SVM^{python}

Similar to Figure A.1 and Figure A.2, we present flowcharts Figure A.3 and Figure A.4 showing the control flow within the SVM^{python} learning and classification procedures, respectively.

A.5 Using SVM^{python} to Make a Binary Classifier

In this example, we will use SVM^{python} to build an actual binary classifier. We shall start with a very simple minimal binary classifier, and then work our way up, exploring all the concepts necessary to write a module for SVM^{python} as we proceed. Despite the simplicity of the example, this will illustrate in fairly complete depth all the steps necessary to make a structured learner in SVM^{python}.

A.5.1 An Initial Bare Bones Binary Classifier

For the sake of simplicity this initial binary classifier will work with linear kernels only, though we note that the framework can be used with other frameworks as well. We begin by making a Python module, which we shall name `binary1`. This is just a plain source file `binary1.py`.

The first order of business in either training or classification is the reading of data. In this case, with binary classification, we read SVM^{light} style inputs, where

lines, each corresponding to an example, are of the form

$$\langle label \rangle \quad \langle index \rangle{:}\langle value \rangle \quad \langle index \rangle{:}\langle value \rangle \quad \langle index \rangle{:}\langle value \rangle \ldots$$

where $\langle label \rangle$ is either 1 or -1, and the remainder of the line specifies a sparse vector, with each positive $\langle index \rangle$ as the integer element number of the vector and $\langle value \rangle$ as the real value, with successive index values being strictly increasing.

```
import svmapi

def read_examples(filename, sparm):
    # This reads example files of the type read by SVM^light.
    examples = []
    for line in file(filename): # Each line corresponds to an example.
        if line.find('#')>=0: line=line[:line.find('#')] # Ignore comments.
        tokens = line.split()
        if not tokens: continue                          # Skip empty lines.
        target = int(tokens[0])                          # Get the label y.
        assert target==-1 or target==1                   # Ensure labels.
        tokens=[tuple(t.split(':')) for t in tokens[1:]] # Get the features.
        features=[(int(k),float(v)) for k,v in tokens]   # Get index,value pairs.
        examples.append((features, target))              # Append example pair.
    print len(examples), 'examples read'
    return examples
```

We have an `import svmapi` for the benefit of future functions which will make use of the functionality provided by the `svmapi` module. Our function `read_examples` should return a sequence of two-element tuples, with the first item of the i-th tuple as the input $\mathbf{x}_i$ (in this case, a bag-of-words document feature vector), and the second element as the label $\mathbf{y}_i$ (in this case, either the integer -1 or 1).

Suppose we try to run the SVM^{python} learner on this module.

```
./svm_python_learn --m binary1 -c 1e3 example1/train.dat model.1
```

We indicate we want to run the SVM^{python} learning executable using the `binary1` module, regularization parameter $C = 1 \cdot 10^3$, the training set at path `example1/train.dat`[3], writing the resulting learned model parameterization to the file `model.`

[3]Note that this comes from the `example1` set available for download from `http://svmlight.joachims.org/` .

1. However, we run into a problem in that we get the error function `Could not find function init_model!` Obviously, just reading data is not enough to learn. We must initialize the model, and perform other tasks.

```
def init_model(sample, sm, sparm):
    sm.size_psi = max(max(k for k,v in x) for x,y in sample)+1
```

In `init_model`, we must at least initialize the maximum size of the $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ vector by setting the `size_psi` attribute. The `psi` function (which corresponds to the combined feature function $\mathbf{\Psi}$) returns vectors, and SVM^{struct} needs to know ahead of time what the maximum size of these vectors will be. In the case of binary classification, we have $\mathbf{\Psi}(\mathbf{x}, \mathbf{y}) = \mathbf{y} \cdot \mathbf{x}$, so the size is the maximum index value for any $\mathbf{x}$ we could return. These vectors are indexed from 0, so we want the maximum "index" value for our vectors plus 1. As for the value itself:

```
def psi(x, y, sm, sparm):
    return svmapi.Sparse([(k, y*v) for k,v in x]) # Psi(x,y) = y * x
```

The `psi` function is the programmatic implementation of the $\mathbf{\Psi}(\mathbf{x}, \mathbf{y})$ combined feature function. The return value is expected to be either a `Sparse` object, or a `Document` object (which functions as essentially a collection of `Sparse` objects, more useful when using kernels on complex objects). A `Sparse` instance is instantiated with the first argument as a sequence of two-element tuples of index-value pairs, where the index is a non-negative `int`, and the value is a `float`. We also provide the loss function Δ:

```
def loss(y, ybar, sparm):
    return 1 if y != ybar else 0  # 1 if labels differ, 0 if the same.
```

The loss function $\Delta(\mathbf{y}, \bar{\mathbf{y}})$ has value 1 if $\mathbf{y} \neq \bar{\mathbf{y}}$, and 0 if $\mathbf{y} = \bar{\mathbf{y}}$. This loss function provided here is equivalent to the default loss function which SVM^{python} will use if

no loss function is provided, so we could just as easily not use it, but we provide it for clarity.

Then we have code for the prediction $h(\mathbf{x}) = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} \langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$ and the separation oracle $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y}\in\mathcal{Y}} H(\mathbf{y})$. If you work forward from the cost functions H of (2.21) and (2.22) respectively for this particular $\mathbf{\Psi}$ and Δ, you will find that `find_most_violated_constraint_margin` and `find_most_violated_constraint_slack` return the proper $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$.

```
def score(x, sm):
    return sum(sm.w[k]*v for k,v in x)   # Effectively, <w, x>.

def classify_example(x, sm, sparm):
    return 1 if score(x,sm)>=0 else -1   # Considered positive for <w,x> >= 0.

def find_most_violated_constraint_margin(x, y, sm, sparm):
    return 1 if 2*score(x,sm)>=y else -1 # Return most violated output.

def find_most_violated_constraint_slack(x, y, sm, sparm):
    return find_most_violated_constraint_margin(x,y,sm,sparm)
```

In fact, the $\text{argmax}_{\mathbf{y}} H(\mathbf{y})$ is identical in both cases. Though this almost never occurs in problem domains more complicated than binary classification, there is still default behavior built into SVM^{python}: if neither of the above separation oracle functions is defined, it will instead default to the more general `find_most_violated_constraint`, which we can define in place of having two separate identical functions:

```
def find_most_violated_constraint(x, y, sm, sparm):
    if 2*score(x, sm) >= y: return 1   # Returns the output associated with
    else: return -1                    # the most violated constraint.
```

Note the use of the `score` function. This function is not one of the developer hook functions; it is just a helpful function in this case. It is worth noting that developers are free to declare whatever other functions, classes, modules, or other

objects within this hook module that they like, so long as they do not use one of the names reserved for the developer hook functions.

This completes the linear binary classification module. Figure A.5 provides a comprehensive view of the entire `binary1` module. We can then learn and classify.

```
./svm_python_learn --m binary1 -c 1e3 -w 2 example1/train.dat model.1
./svm_python_classify --m binary1 example1/test.dat model.1 predictions
```

The first command learns a model parameterization from the training data `example1/train.dat` and writes it to `model.1`, using regularization parameter $C = 1 \cdot 10^3$, and the `-w 2` option tells the structural SVM to use the 1-slack variant described in Section 2.1.4. The second command uses this model parameterization to classify `example1/test.dat`, writing the predicted values to a file named `predictions`.

A.5.2 Writing the Output Hook Functions

SVM^{python} and SVM^{struct} allow many hook functions through which the developer may produce custom output at appropriate times. By default, these functions do nothing, with the exception of `write_label`, which simply prints the output to a file. However, in some cases, it may be useful to produce other output.

Let us start with `print_learning_stats`. One nice thing would be if the training executable output the average loss on the training set. By default, SVM^{python} does not. Fortunately for us, `print_learning_stats` hook function is called once learning finishes. In this case, we sum the losses of predictions over all training examples, and then output the resulting average.

```
def print_learning_stats(sample, sm, cset, alpha, sparm):
    total = sum(loss(y, classify_example(x,sm,sparm), sparm) for x,y in sample)
    print 'Average loss on train set is', float(total) / float(len(sample))
```

On the earlier training command, now we have an additional line of output: `Average loss on train set is 0.0075`, which is the same average loss one gets if one runs `svm_python_classify` over the training set with the learned model.

The classification executable, on the other hand, does output average loss, but in certain situations it would be helpful to have more information beyond the average loss. For example, in this case of binary classification we might also wish to have precision and recall over the test set. In implementing such a change, we make use of the hook functions `eval_prediction` and `print_testing_stats`. Unlike in training, during classification the predictions are already being produced. After every prediction, the `eval_prediction` function is called. The intent of this function is to accumulate statistics, which are then output in `print_testing_stats` once iteration over all inputs has finished. Let us see the implementation, which will then be explained.

```
def eval_prediction(exnum, (x, y), ypred, sm, sparm, teststats):
    if exnum==0: teststats = 0, 0, 0
    falseneg, falsepos, truepos = teststats
    if   y== 1 and ypred== 1: truepos  += 1
    elif y== 1 and ypred==-1: falseneg += 1
    elif y==-1 and ypred== 1: falsepos += 1
    return falseneg, falsepos, truepos

def print_testing_stats(sample, sm, sparm, teststats):
    falseneg, falsepos, truepos = teststats
    # Compute recall.
    try: rec=float(truepos)/float(truepos+falseneg)
    except ZeroDivisionError: rec=1.0
    print 'Recall is %g' % rec
    # Compute precision.
    try: prec=float(truepos)/float(truepos+falsepos)
    except ZeroDivisionError: prec=1.0
    print 'Precision is %g' % prec
```

The idea is that `eval_prediction`, getting the true input/output pair $(\mathbf{x}, \mathbf{y})$ and predicted output $\mathbf{y}_{pred} = h(\mathbf{x})$, accumulates in `teststats` statistics. Initially this argument is `None`, but in the first example (when `exnum==0`), we let it be the

number of false negatives, false positives, and true positives, which we increment as appropriate depending upon `y` and $\mathbf{y}_{pred}$. Whatever the return value from `eval_prediction` is passed as the next call's `teststats`, with the final return value passed to `print_testing_stats` as its `teststats` value. In this case, the `teststats` is a tuple of the three integers.

It would be quite possible and in some respects simpler to simply repeat the predictions within `print_testing_stats`. However, through this arrangement, the additional computational cost of duplicating the prediction computation is avoided.

There is an additional hook function `print_iteration_stats`, which is called at the end of each iteration over all training examples. This function accepts many arguments that detail some of the more technical aspects of how iteration is proceeding. We do not detail the use of this function as its usefulness is somewhat esoteric.

Finally, there is the `write_label` hook function, which accepts two arguments: an open file, and a label which came from the `classify_example` prediction function. If left unimplemented, the default behavior for this function is to simply print the label to the file. We have seen this default behavior in action before: the `predictions` file produced by the `svm_python_classify` executable in the earlier sample commands will contain predictions, `1` and `-1`, one per line, corresponding to the outputs. For the sake of argument, suppose that we choose to make output more parsimonious: instead of having `1` and `-1` one per line, suppose we have a continuous stream of `+` and `-` with no linebreaks. We can accomplish this quite simply by implementing the `write_label` hook function like so:

```
def write_label(fileptr, y):
    fileptr.write('+' if y==1 else '-')
```

With this change, the predictions file will contain as many characters as there are documents in the classification set, and only `+` and `-` characters.

A.5.3 Custom Constraints

Algorithm 1, and correspondingly its implementation within SVM^{struct} and SVM^{python} starts with an empty constraint set. For various reasons, sometimes one wants to introduce special constraints. In this case we will show how to introduce constraints so that, when learning under a linear model, all learned weights are positive.

Unfortunately, this example requires a few minor changes to other seemingly unrelated pieces of the code, specifically the introduction of a bias term and an output of the feature.

We must first introduce a bias term, because in our input data, all feature values are positive, and if we have all weights positive, it would be impossible to render anything other than a positive judgement. However, unlike, say, SVM^{light} or other SVM frameworks for binary classification, the SVM^{struct} and SVM^{python} frameworks do not define a bias term; the concept would not have a consistent meaning across all machine learning applications, and it is unclear how the value of the term would be derived. However, since an "offset" is often useful, a common approach is to instead add a constant "offset" feature to data. By doing so, we effectively add a bias term, although one subject to regularization through the $\|\mathbf{w}\|$ term of the SVM QP. We have our $\mathbf{x}$ examples as a list of index-value pairs representing a vector; these are used in the computation of $\mathbf{\Psi}$, in prediction, and in the separation oracle. Within `read_examples`, after `features` is declared we add in a new index-value pair like so:

```
features = [(0, 1.0)] + features
```

The index value 0 is valid; remember that these index values are reused as indices in the `Sparse` declaration in the `psi` function, which requires only non-negative integers. However, the 0 index value never appears in SVM^{light} style data. Thus, we may safely use it as a bias term. In reality, from an efficiency standpoint, it would be preferable to not explicitly insert this feature; as it is shared, we could modify the `psi`, `classify_example`, and separation oracle functions to act as if the constant feature is there. However, in this didactic document, we go with simplicity.

For informative purposes, we also introduce the following at the end of the `print_learning_stats` function. This is so we can clearly see the effect of our coming change.

```
print 'Range of sm.w: %g to %g, with bias feature %g' % (
    min(sm.w[1:]), max(sm.w[1:]), sm.w[0])
```

If we run the learner as is with command

```
./svm_python_learn --m binary2 -c 1e5 example1/train.dat model.1
```

one of the last lines of output is

```
Range of sm.w: -1.01823 to 1.47326, with bias feature 0.0587552
```

so we have some negative weights. Now let us try adding the constraints to enforce non-negative feature values. We do this through implementation of the `init_constraints` function.

Before we begin on how to do that, it is important to understand `Document` objects, and why they exist. The `Document` class (or the `DOC` struct in C) is SVM^{python}'s, SVM^{struct}'s, and SVM^{light}'s way of storing vectors. We need a `Document` class in addition to the existing `Sparse` vector class, since vectors in the context of kernel machines are more complicated than normal intuitive understanding of real vectors would suggest.

For example, suppose we have vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathcal{V}$ where $\mathcal{V}$ is some vector space. Suppose further we wish to exploit a kernel $\kappa : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$ which maps into an implicit vector space $\mathcal{V}'$, with $\phi : \mathcal{V} \rightarrow \mathcal{V}'$ as the implicit mapping from $\mathcal{V}$ to $\mathcal{V}'$, so that $\kappa(\mathbf{r}, \mathbf{s}) \equiv \langle \phi(\mathbf{r}), \phi(\mathbf{s}) \rangle$. If we want to compute the product $\langle \phi(\mathbf{a}) + \phi(\mathbf{b}), \phi(\mathbf{c}) + \phi(\mathbf{d}) \rangle$, this product is equivalent to $\kappa(\mathbf{a}, \mathbf{c}) + \kappa(\mathbf{a}, \mathbf{d}) + \kappa(\mathbf{b}, \mathbf{c}) + \kappa(\mathbf{b}, \mathbf{d})$, and emphatically not equivalent to $\kappa(\mathbf{a} + \mathbf{b}, \mathbf{c} + \mathbf{d})$.

Representation of vectors summed in the implicit feature space *but not real space* is endemic through kernelized SVMs, most notably in the model parameterization $\mathbf{w}$, but in structural SVMs also in the $\mathbf{\Psi}$ combined feature function as seen in (6.11) under Section 6.4.

To support this sort of "implicit sum," we have `Document` objects, collections of `Sparse` objects. Kernelized products between `Document` instances are sums of the kernelized products between combinations of the `Sparse` instances contained within the two `Document` instances. All `Sparse` instances have parameters to control how their kernel evaluation proceeds: only those with matching `kernel_id` attributes have their product computed (so `Sparse` instances with mismatching IDs are effectively orthogonal), and any kernel evaluations of a `Sparse` instance is multiplied by its `factor` attribute.

Constraints are exposed to SVM^{python} developer hook functions as sequences of two-element tuples. The first element is a `Document` instance, and the second element a `float`. If we call these $\mathbf{d}$ and ℓ, respectively. This leads to a constraint:

$$\langle \mathbf{w}, \mathbf{d} \rangle + \xi_j \geq \ell \tag{A.1}$$

The ξ_j is a slack variable. Owing to the nature of the optimization procedure, it is impossible for a constraint to not have an associated slack. Which slack variable is used (e.g., the value of j) is controlled by a `Document` instance's `slackid` attribute. This must be a positive integer. We select a slack ID that is not used by any training example (namely, the number of examples plus one), and share it among all of the positivity constraints.

```
def init_constraints(sample, sm, sparm):
    cons = []
    for k in xrange(1,sm.size_psi):
        s = svmapi.Sparse([(k, 100.0)])
        d = svmapi.Document([s], slackid=len(sample)+1)
        cons.append((d, 1.0))
    return cons
```

The `init_constraints` function allows the user to define custom constraints by returning a list of `Document`, `float` tuples. So, for every feature, we add a constraint, with a vector that (in the linear case) selects out that feature, multiplies it times 100, and enforces that this be greater than 1 punishable by the slack associated with the constraint, so the feature value must exceed $\frac{1}{100}$. Unfortunately, due to the requirement that some slack exists, an arrangement like this is the best we can do while ensuring positivity. We could change the 100 to a higher value, but at some point we make the underlying matrix problem ill-conditioned, so this must be done with care. Owing to the presence of the slack variable, it is somewhat tricky to get a proper mix to ensure positivity.

If we re-run the learner under the above settings, we now get the final output

```
Range of sm.w: 0.0096154 to 3.30798, with bias feature -1.30616
```

A.5.4 Kernels

In this section we will extend our previously linear classifier into one that can handle non-linear implicit feature mappings through kernels.

For the sake of this section, we do not use the additional code written in Section A.5.3, but rather start from the code as it existed prior to that section.

In SVM^{python} and SVM^{struct}, one may set the SVM kernel through the `-t` command line option. While the use of kernels affects internal computations, user hook code is also not immune. Classification and the separation oracle involve an argmax over an inner product $\langle \mathbf{w}, \mathbf{\Psi}(\mathbf{x}, \mathbf{y}) \rangle$, but with the use of kernels, this inner product is potentially a kernel evaluation. The code as it stands makes assumptions that it is work in the linear case only. Fortunately, with a few calls to some supporting objects and support functions, we can get the learner and classifier "kernelizable" in short order.

Our first step is to change `read_examples` so that $\mathbf{x}$ examples are stored as `Sparse` instances, instead of Python lists of two-element tuples. (Actually, from a space and time efficiency standpoint, this would have been a good implementation to have in any event.) We remove the line starting with `examples.append` and in its place insert two lines

```
svec = svmapi.Sparse(features)
examples.append((svec, target))             # Append example pair.
```

so that the $\mathbf{x}$ input is now `svec` instead of `features`. As a side note, with this

change, the module's functionality remains intact even with the same code: `Sparse` objects iterate over their index-value pairs just as if they were a list of index-value tuples.

The next two steps are to change the `psi` and `score` functions.

```
def psi(x, y, sm, sparm):
    return svmapi.Sparse(x, factor=y)

def score(x, sm):
    return sm.svm_model.classify(x)
```

Recall that previously, in `psi`, we directly modified the vector values so that they were negated if $\mathbf{y} = -1$. Our intention was really that the inner product $\langle \mathbf{w}, \mathbf{x} \rangle$ be negated if $\mathbf{y} = -1$, and it is more appropriate, in the general kernel case, that this negation happen after the kernel evaluation, not before. This is the primary reason for the `factor` attribute on the `Sparse` instance, a multiplicative factor by which kernel evaluations on this `Sparse` vector are multiplied. In this new version of `psi`, we construct a new `Sparse` vector from the existing $\mathbf{x}$ `Sparse` vector, and set the appropriate factor.

The `score` function, as before, computes the inner product $\langle \mathbf{w}, \mathbf{x} \rangle$, but this time with the aid of the method `classify` on the `Model` instance contained within the `StructModel` instance. This function computes the kernelized inner product between the model parameterization and our vector $\mathbf{x}$.

We now have a kernelized binary classifier, and by relying upon existing library routines we can do so without much effort at all.

A.6 Summary

This chapter described SVM^{python}, an extension to SVM^{struct} that allows one to derive a structural SVM learning algorithm in Python. We first discussed SVM^{struct} which, owing to the use of C, poses some challenges for some developers, especially researchers that may be more interested in prototyping new ideas quickly than in raw speed. In SVM^{python}, by allowing developers to develop structural SVM algorithms in pure Python, we retain Python's advantages in rapid prototyping and superior comprehensibility. Further, by relying upon Python, many of the mundane tasks of implementing a structural learner in C are obviated, most notably those related to memory management, rich comparisons of data, and especially model parameterization, serialization, and deserialization. We closed the chapter with an in-depth example implementation of a very simple structural SVM that does binary classification. Through this example, we provided a tour of most of the important functionality of SVM^{python} from a developer's perspective.

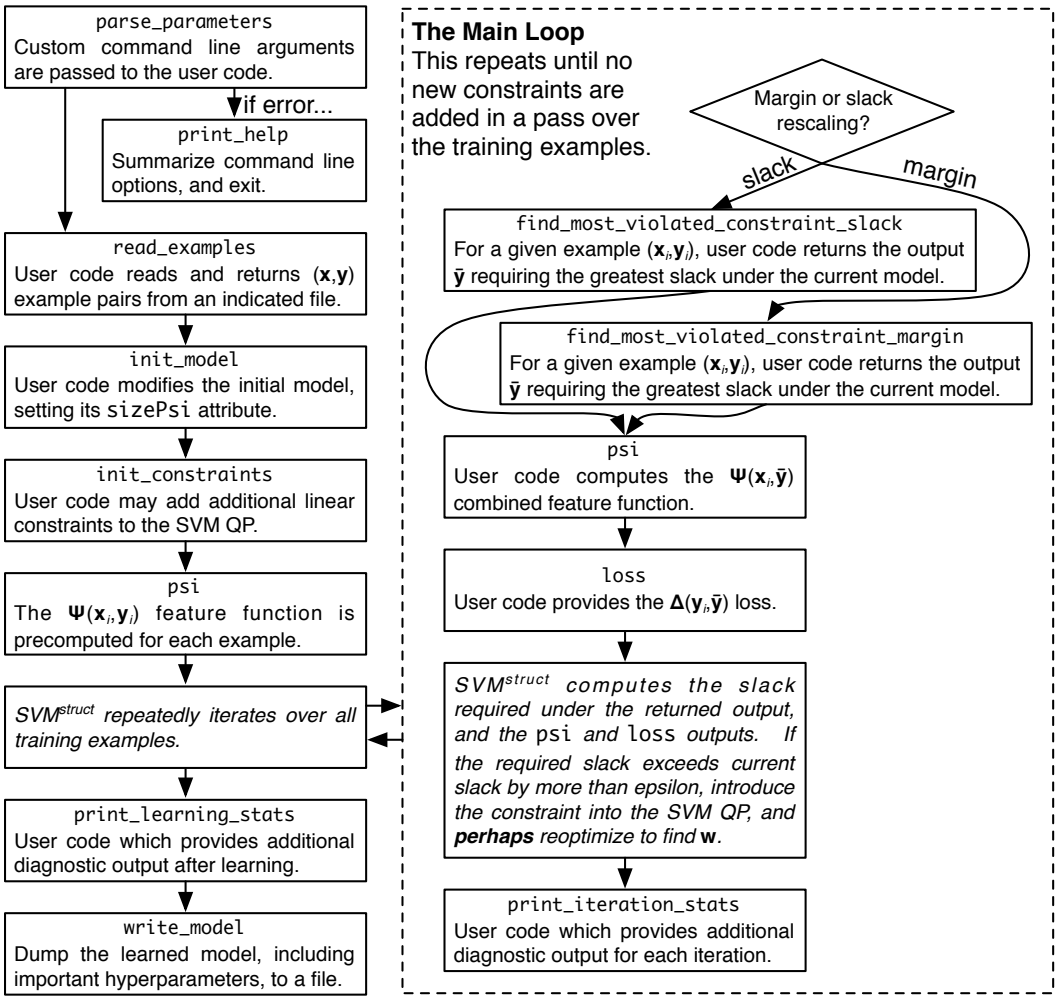

Figure A.3: Flowchart showing the flow of execution within the SVM^{python} learner, with the flow of execution starting from the upper left. Steps associated with a particular call to a developer's module function have the box lead with the function name.

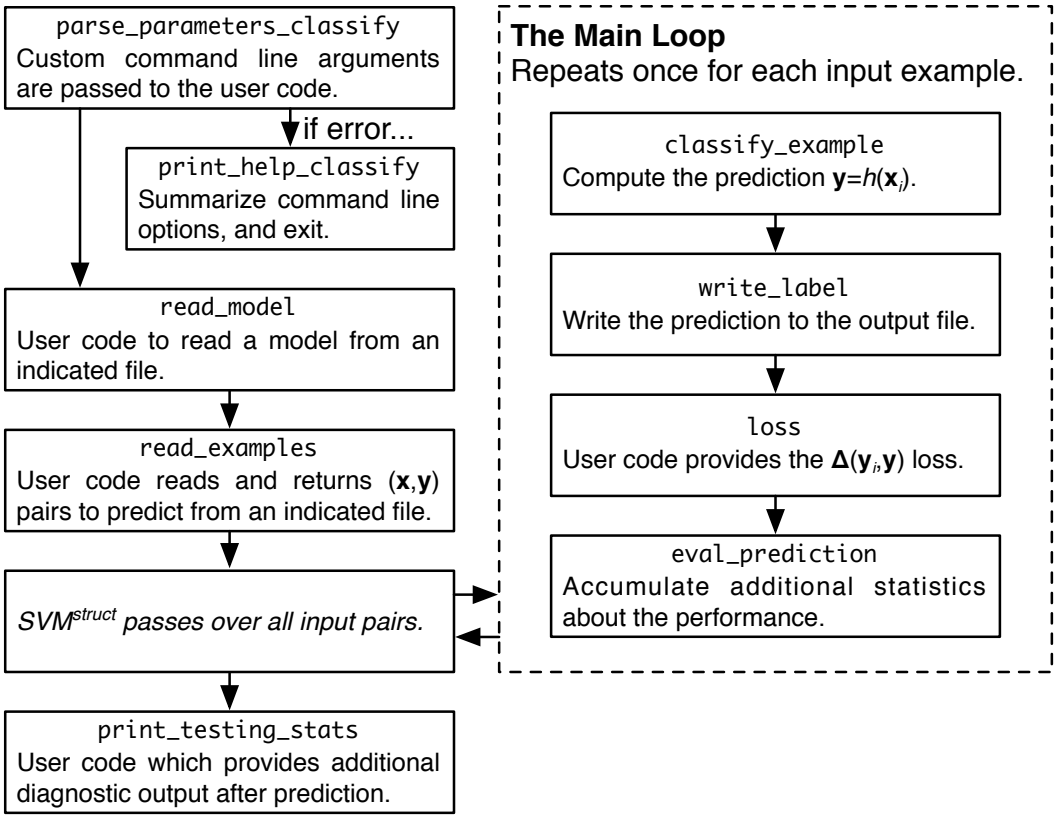

Figure A.4: Flowchart showing the flow of execution within the SVM^{python} classifier, with the flow of execution starting from the upper left. Steps associated with a particular call to a developer's module function have the box lead with the function name.

```
import svmapi

def read_examples(filename, sparm):
    # This reads example files of the type read by SVM^light.
    examples = []
    for line in file(filename): # Each line corresponds to an example.
        if line.find('#')>=0: line=line[:line.find('#')] # Ignore comments.
        tokens = line.split()
        if not tokens: continue                          # Skip empty lines.
        target = int(tokens[0])                          # Get the label y.
        assert target==-1 or target==1                   # Ensure labels.
        tokens=[tuple(t.split(':')) for t in tokens[1:]] # Get the features.
        features=[(int(k),float(v)) for k,v in tokens]   # Get index,value pairs.
        examples.append((features, target))              # Append example pair.
    print len(examples), 'examples read'
    return examples

def init_model(sample, sm, sparm):
    sm.size_psi = max(max(k for k,v in x) for x,y in sample)+1

def psi(x, y, sm, sparm):
    return svmapi.Sparse([(k, y*v) for k,v in x]) # Psi(x,y) = y * x

def loss(y, ybar, sparm):
    return 1 if y != ybar else 0         # 1 if labels differ, 0 if the same.

def score(x, sm):
    return sum(sm.w[k]*v for k,v in x)   # Effectively, <w, x>.

def classify_example(x, sm, sparm):
    return 1 if score(x,sm)>=0 else -1   # Considered positive for <w,x> >= 0.

def find_most_violated_constraint(x, y, sm, sparm):
    return 1 if 2*score(x,sm)>=y else -1 # Return most violated output.
```

Figure A.5: The code for `binary1.py`, an extension module that implements linear binary classification for SVM^{python}.

APPENDIX B

PYGLPK : THE PYTHON GNU LINEAR PROGRAMMING KIT

PyGLPK is a Python extension module that provides an interface to the GLPK, the GNU Linear Programming Kit [71]. The underlying GLPK is a library containing a set of C routines through which one may solve linear programming and mixed integer programming problems.

The GLPK solves linear programs of the form where one is minimizing (or maximizing) variables $x_0, x_1, x_2, \ldots, x_{n-1}$ and $y_0, y_1, y_2, \ldots, y_{m-1}$ over an objective function

$$z = c_0x_0 + c_1x_1 + c_2x_2 + \cdots + c_{n-1}x_{n-1} + c' \tag{B.1}$$

subject to constraints

$$\begin{array}{rcccccccc}
y_0 & = & a_{00}x_0 & + & a_{01}x_1 & + & a_{02}x_2 & +\cdots+ & a_{0,n-1}x_{n-1} \\
y_1 & = & a_{10}x_0 & + & a_{11}x_1 & + & a_{12}x_2 & +\cdots+ & a_{1,n-1}x_{n-1} \\
y_2 & = & a_{20}x_0 & + & a_{21}x_1 & + & a_{22}x_2 & +\cdots+ & a_{2,n-1}x_{n-1} \\
\vdots & \vdots & \vdots & & & & & & \\
y_{m-1} & = & a_{m-1,0}x_0 & + & a_{m-1,1}x_1 & + & a_{m-1,2}x_2 & +\cdots+ & a_{m-1,n-1}x_{n-1}
\end{array} \tag{B.2}$$

and variable bounds

$$\begin{array}{c}
\forall i \in 0..n-1 : \ell_{x_i} \leq x_i \leq u_{x_i} \\
\forall j \in 0..m-1 : \ell_{y_j} \leq y_j \leq u_{y_j}
\end{array} \tag{B.3}$$

where the x_i and y_j variables (the column and row variables, respectively) are all in $\mathbb{R}$, optionally with some $x_i \in \mathbb{Z}$ within a mixed integer program. The c_i and a_{ij} values are not variables, but constant objective function terms and constraint matrix terms.

The PyGLPK is a Python C extension module, which provides a module `glpk` in Python for utilizing the functionality of the GLPK, allowing one to solve problems.

In this chapter we will review the PyGLPK, and go over concrete examples of its use. This chapter does not present a full list and description of every method and attribute of every object in the PyGLPK. Those wishing such a comprehensive document will find it in the PyGLPK source distribution. The GLPK presents a very large library of API functions, and the PyGLPK exposes nearly all of its functionality. A suitable explanation of every possible use of every possible option would be a book unto itself. Rather, we present a functional introduction to PyGLPK, demonstrating most of its important functionality. This chapter assumes one is familiar with both linear programming and the Python programming language.

This chapter describes PyGLPK 0.3, which encapsulates the functionality of GLPK 4.18 through 4.31. At the time of this writing GLPK is a quickly evolving library, and future versions of PyGLPK may change as the underlying GLPK changes.

B.1 Principles of the PyGLPK

The GLPK was chosen for the underlying linear programming library because it is freely available, reasonably efficient, well documented, allows mixed integer programming, provides a programmatic interface through a dynamic library, and is relatively bug free both in terms of crashes and memory leaks. Surprisingly few linear programming libraries meet all of these criteria. Being bug free is particularly important, as some applications of linear programming may call for many thousands of separate linear programs to be solved, as in Section 5.5, even a small memory leak can be debilitating.

Python interfaces to GLPK already exist, so why write a new Python wrapper?

While some Python interfaces to GLPK exist, they all either expose an extremely limited subset to the API [93], or they are just total transliterations of the GLPK C API into Python [80, 81]. The goal of PyGLPK was to expose nearly all documented behavior of the GLPK while maintaining a Pythonic interface.

Pythonic, a vague but useful adjective, means making good use of or supporting Python idioms. By an idiom we mean a characteristic way of accomplishing a certain type of task. For even the simplest programming tasks, there is often an established "right way" to accomplish that task, suggested by either the language or the culture that grew up around that language. For example, consider an object that contains a collection of items, and the task of iterating over and retrieving those items. In C one might construct `next` and `get` functions that, given the object and an index, return either that subobject or the index to the next object. In C++ one might use operator overloading to provide iterators whose use resembles pointer arithmetic. In Java one might exploit the +Iterable+ and +Iterator+ interfaces.

Let us give a concrete example of this through a crash introduction to PyGLPK. The class central to `glpk` is `LPX`, which encapsulates a linear program, including program definition and current solutions. An `LPX` instance (let us call it `lp`) contains a member `rows`, e.g., `lp.rows`, which somehow contains the linear program's rows. We seem to know nothing about how to interact with this row's object. However, if we further state that `rows` acts like a sequence, a hypothetical semi-experienced Python programmer knows how to do many things with no further information. He knows how to get the number of rows (`len(lp.rows)`); he knows how to get the first row (`lp.rows[0]`); he knows how to iterate over all rows (`for row in lp.rows: # do something`); he knows how to get a list of

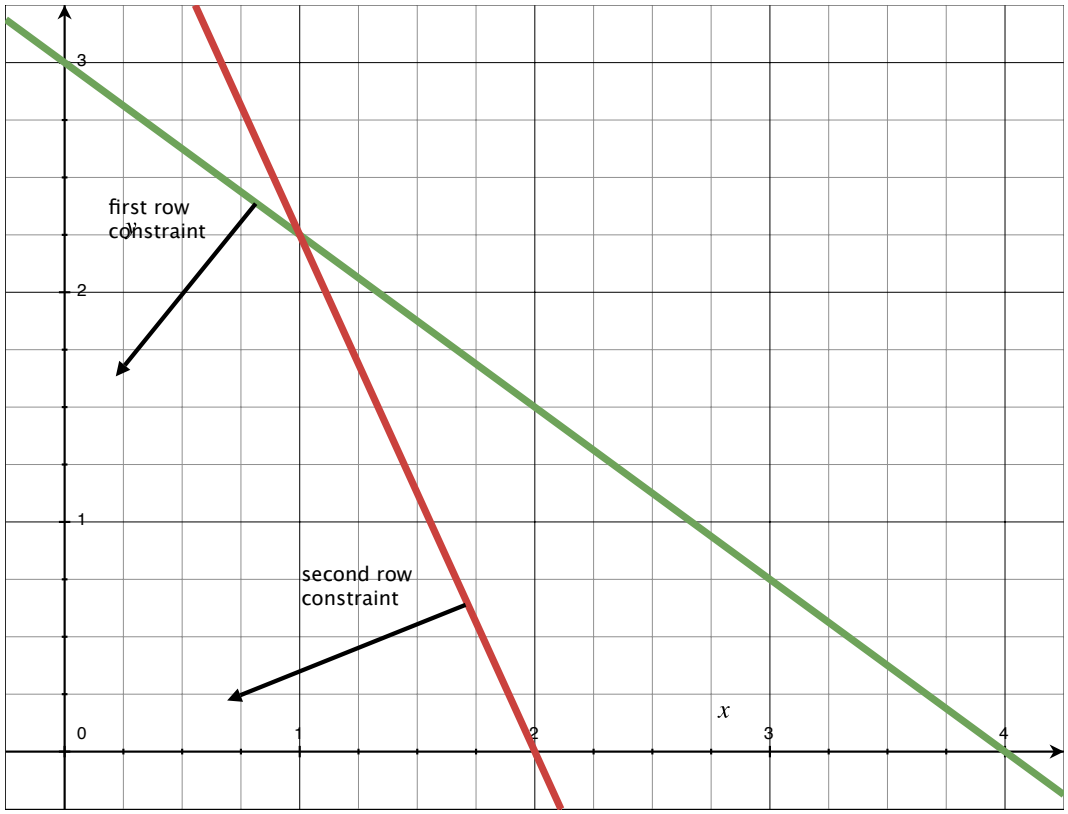

Figure B.1: Graphical representation of the two linear constraints of the problem of Section B.2.

the last three rows (`lp.rows[-3:]`); he knows how to delete the last three rows (`del lp.rows[-3:]`). Through respecting established Python idioms with regard to sequences, these portions of PyGLPK could be termed Pythonic.

B.2 Simple Two Dimensional Example

In this section we will illustrate the usage of PyGLPK by presenting a simple linear program, and explaining the PyGLPK code that implements and solves the linear program line by line. Suppose we consider the following maximization problem over two variables x_0, x_1:

Optimization Problem 14. (Simple Linear Program of Section B.2)

$$
\begin{array}{lll}
\textit{maximize} & & x_0 + x_1 \\
\textit{subject to} & y_0 = & 3x_0 + 4x_1 \\
& y_1 = & 9x_0 + 4x_1 \\
& y_0 \leq & 12 \\
& y_1 \leq & 18
\end{array}
$$

The constraints of this problem are represented graphically in Figure B.1. From the picture and the linear constraint equations, it is clear that the optimal solution lies at $x_0 = 1$, $x_1 = 2.25$. However, we will solve this problem with PyGLPK.

```
1  import glpk                                          # Import the PyGLPK module.
2
3  lp = glpk.LPX()                                      # Construct the linear program.
4  lp.obj.maximize = True                               # Set as maximization.
5  lp.cols.add(2)                                       # Add const. matrix columns.
6  for col in lp.cols:                                  # Iterate over columns.
7      col.name = 'x%d'%col.index                       # Name the columns x0, x1.
8      col.bounds = None, None                          # No bounds on the variables.
9      lp.obj[col.index] = 1.0                          # Each objective coef. is 1.
10 lp.rows.add(2)                                       # Add const. matrix rows.
11 for row in lp.rows: row.name = 'y%d'%row.index      # Name the rows y0, y1.
12 lp.matrix = [3, 4,                                  # Set the constraint matrix
13              9, 4]                                  # matrix in one assignment.
14 lp.rows[0].bounds = None, 12                        # Constrain y0 <= 12 .
15 lp.rows[1].bounds = None, 18                        # Constrain y0 <= 18 .
16 lp.simplex()                                        # Run the simplex algorithm.
17 for col in lp.cols:                                 # For each column print out the
18     print '%s = %g' % (col.name, col.value)         # variable's name and value.
```

Let us go through this program line by line. In line 1 we see the import statement of the PyGLPK module, which is named `glpk`. In line 3 we construct the linear program instance by calling the `LPX` constructor. Line 4 illustrates the use of the `obj` member of the linear program, an object of type `Objective`, an instance of which holds information relating to the objective function value. In this case, we are setting its `maximize` attribute to `True`, which indicates that we are optimizing

to maximize the objective (whereas, if the attribute were set `False`, we would be minimizing the objective).

The problem has two columns, and two rows. The rows and columns are stored within two attributes of the linear program, `rows` and `cols` respectively, both of type `BarCollection`. We add two members with this object's `add` method in line 5. By iterating over the `cols` member as we do in line 6, we can go over all of the columns within the program. The variable `col` is a member of class `Bar`; both columns and rows are instances of this class. These instances hold information not only about a column and row within the constraint matrix, but also information about the variable associated with that row or column.

In line 7 we set the name of each column's variable to `"x0"` and `"x1"`, to match the names given in OP 14. This is accomplished by assigning a string to each `Bar` instance's `name` attribute. Note that actually naming the columns is completely optional. It is done here for illustrative purposes.

In line 8, we set the bounds for the column's variable. In both cases, we do not want any bounds for the variable, neither lower nor upper bounds. Setting bounds is done through a `Bar` instance's `bounds` attribute, which accepts two values. The first value is a lower bound, the second is an upper bound. This establishes the acceptable range for a variable. The value `None` indicates that there should be no corresponding bound. So, by setting the `bound` attribute to `None, None`, we indicate that the variable should be completely unbounded.

In line 9, we set the objective function's coefficients. Since there is a term in the objective function for every column variable, we set the column variable's coefficient in the objective function, while we are at it. The coefficient is set

by indexing into the `obj` attribute of the linear program, indexing either being numeric, or by name of the column. In this case, indexing is numeric, by using the column's `index` attribute to get its corresponding column number in the linear program's constraint matrix.

In line 10, we next add the two rows, in a similar fashion to how we added the two columns.

In line 11, we iterate over the rows and set the row names to `"y0"` and `"y1"`, in a similar fashion as we set the column names in lines 6 and 7.

In line 12 and 13, we set the linear program's constraint matrix. In this particular case, we set the constraint matrix all at once, by assigning to a linear program's `matrix` attribute. In a real application, it would be far more plausible to set the constraint matrix row by row, or column by column, or at least set the constraint matrix in sparse notation. We shall see examples of these in future examples, but for now, we keep this simple example where a matrix is specified completely and explicitly in one command.

In line 14 and 15, we have the bounds set on the rows variables. In OP 14, we want to have $y_0 \leq 12$ and $y_1 \leq 18$. Correspondingly, we set the bounds on the row indexed by 0, which corresponds to y_0, to `None, 12` (meaning no lower bound, but an upper bound of 12), and the bounds on the row indexed by 1, which corresponds to y_1 ,to `None, 18` (no lower bound, and an upper bound of 18).

We have now fully specified our problem. It is now time to run actual optimization. In line 16, we run the simplex algorithm through the `simplex` method on the `LPX` instance.

In the last two lines, 17 and 18, we print the variable values for x_0 and x_1, by iterating over the columns in the `cols` instance, and for each column, printing out its name and value. As expected, the final output to this problem is

```
x0 = 1
x1 = 2.25
```

We have thus built and solved a very simple linear program.

B.3 Satisfiability Solver Example

In addition to linear programming, PyGLPK through the GLPK also allows one to solve mixed integer programs (MIP). Mixed integer programs are specified in the same fashion as a regular linear program, except that certain column variables can be constrained to also be integral. While optimizing a mixed integer program is known to be an NP-hard problem, they are useful in many situations. One of the most commonly used subclasses of a mixed integer program is a Boolean integer problem, where the column variables must be integral and be only 0 and 1.

In this section we show a simple example of how to use PyGLPK to find solutions for the Boolean satisfiability problem for a given conjunctive normal form expression. The implementation will thus essentially prove through reduction the NP-hardness of the problem.

First, what is a Boolean satisfiability problem? In the Boolean satisfiability problem, given a Boolean expression, we want to determine if there is an assignment of $True$ and $False$ values to the variables of the expression such that the expression evaluates to $True$. Suppose one has a conjunctive normal form expression, that

is, a conjunction (and-ing, $\wedge$) of several disjunctions (or-ing, $\vee$) of logical literals, e.g.:

$$(\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \tag{B.4}$$

We want to find truth values to all four x_i variables so that the CNF expression is true. This problem has been viewed from many different ways, but we'll see how to encode and (we hope) solve it within a mixed-linear program. We will build a function `solve_sat` to satisfy a given CNF.

First, we need to define how we encode our input CNF expressions that we want to satisfy:

- Each logical literal is represented as either a positive or negative integer, where `i` and `-i` correspond to the logical literals x_i and $\neg x_i$, respectively.
- Each clause in the expression, i.e., disjunction of literals, is represented as a tuple of such encoding of literals, e.g., `(-1, 2, -3)` represents the disjunction $(\neg x_1 \vee x_2 \vee \neg x_3)$.
- The entire conjunctive expression is a list of such tuples, e.g., the expression above would have encoding:
  `[(-1, -3, -4), (2, 3, -4), (1, -2, 4), (1, 3, 4), (-1, 2, -3)]`
- The function will return either `None` if it could not find a satisfying assignment, or a list of Booleans `assignment` representing the satisfying assignment, where the truth of each logical variable x_i is held in `assignment[i-1]`.

This is our strategy for how to solve this with a mixed integer program:

1. For each logical variable x_i, have a column variable representing both its positive and negative literals x_i and $\neg x_i$. These column variables should be either 0 or 1 depending on whether the corresponding literal is false or true, respectively.

   With some extra effort, at the cost of a slight amount of extra complexity, one could halve the number of variables, with one column variable for each logical expression variable, instead of two for each. However, for didactic purposes we accept a slightly less efficient encoding of the problem.

2. Because we want literal consistency, we specify that the sum of all literal pair column variables must be 1. This forbids literals for a given logical variable from being set both false or both true.

3. For each clause, we define a constraint specifying that the sum of all its literal column variables must be at least 1. This forces each clause to be true.

4. First we run the simplex solver (implying a relaxed problem where the column variables can range from 0 to 1). Then we run the integer solver (the column variables can be either 0 or 1).

5. If the integer solver finds an optimal solution, we return a list of `bool` values, `True` and `False`, corresponding to x_1, x_2, etc., in the input CNF. If a positive literal has a corresponding column variable with value 1, then we assign its logical variable to true. Correspondingly, if there is no satisfying assignment found, we return `None`.

Here is the implementation of that function:

```
1  def solve_sat(expression):
2      if len(expression)==0: return [] # Trivial case.  Otherwise count vars.
3      numvars = max([max([abs(v) for v in clause]) for clause in expression])
4      lp = glpk.LPX()                  # Construct an empty linear program.
5      glpk.env.term_on = False         # Stop the annoying output.
```

```
6       lp.cols.add(2*numvars)          # As many columns as there are literals.
7       for col in lp.cols:             # Literal must be between false and true.
8           col.kind = bool
9       def lit2col(lit):               # Function to compute column index.
10          return [2*(-lit)-1,2*lit-2][lit>0]
11      for i in xrange(1, numvars+1):  # Ensure "oppositeness" of literals.
12          lp.rows.add(1)
13          lp.rows[-1].matrix = [(lit2col(i), 1.0), (lit2col(-i), 1.0)]
14          lp.rows[-1].bounds = 1.0    # Must sum to exactly 1.
15      for clause in expression:       # Ensure "trueness" of each clause.
16          lp.rows.add(1)
17          lp.rows[-1].matrix = [(lit2col(lit), 1.0) for lit in clause]
18          lp.rows[-1].bounds = 1, None # At least one literal must be true.
19      retval = lp.simplex()           # Try to solve the relaxed problem.
20      assert retval == None           # Should not fail in this fashion.
21      if lp.status!='opt': return None # If no relaxed solution, no exact sol.
22      retval = lp.integer()           # Try to solve this integer problem.
23      assert retval == None           # Should not fail in this fashion.
24      if lp.status != 'opt': return None
25      return [col.value > 0.99 for col in lp.cols[::2]]
```

B.3.1 Line by Line Explanation

We shall now go over this code line by line.

Lines 2 through 3 are pretty straightforward non-PyGLPK Python code. The first line takes care of the boundary case where we have an empty expression. In the second line, from the expression, we find the maximum indexed logical variable we have, and use that as our count of the number of logical variables.

In line 4, with the LPX constructor, we construct an empty linear program.

The line 5 is a simple preference assignment to quiet the input. Within the `glpk` module, there is a singleton member `env`, of type `Environment`. By assigning to various attributes contained within `env`, you can affect behavior of the underlying GLPK library. In this case, we are assigning `False` to the `term_on` (terminal output on) parameter, to suppress all output.

In line 6, we add as many column variables in the linear program as there

are possible literals over all our logical variables. Each logical variable x_i has two possible literals: itself (x_i), and its negation ($\neg x_i$).

Initially we have no columns at all. So, from the `lp.cols` we call the `add` method, telling it to add as many columns as there are twice the number of logical variables.

In line 7 and 8, we set the `kind` of each column object as a `bool`. The `kind` attribute may be any of `float` (which is default), `int`, or `bool`. A mixed integer program is an `LPX` instance which has 1 or more columns set as `int` or `bool`. Setting the kind as `bool` is actually equivalent to setting the kind as `int` with bounds from 0 to 1.

Remember, these `lp.cols` objects act like sequences (albeit with restrictions on their content). In order to access their elements (in this case, columns), we can either iterate over the columns as we do here, or index into them directly as `lp.cols[colnum]`.

Line 9 and 10 define a helper function `lit2col` to smooth implementation by providing a clean translation from the literal number in our `expression` function argument, to the column number. Recall that we have a column for each possible literal. This function maps literal code 1 to column index 0, -1 to column index 1, 2 to 2, -2 to 3, 3 to 4, -3 to 5, 4 to 6, and so forth.

Lines 11 through 14 define our consistency constraints to make sure two opposite literals are not both true or not both false. For each logical variable, we add one new row (what will be a consistency constraint). Notice that we are now using the `lp.rows` object. Recall that this is similar to the `lp.cols` object, and in reality they are the same type, except it represents the rows of the problem,

instead of the columns.

In line 13 we get the last row, which is the one we just added (note the use of the -1 index to address the last row), and assign to its `matrix` attribute. The `matrix` attribute for any row or column corresponds to the entries of the row or column vector in our constraint matrix. In this case, we are setting the two locations of this constraint matrix row corresponding to the two column variables for x_i and $\neg x_i$ to 1.0, so that a variable and its negation must be 1, and neither 0 (both false) or 1 (both true).

Finally, in line 14, we set the `bounds` attribute for this row's variable to `1.0`. Note that this differs from the previous bound definition: here we use only one number. This indicates we want an equality constraint. More generally, setting the `bounds` attribute to just a single value a is equivalent `bounds` to a, a. For instance, it would have been equivalent to assign `1.0, 1.0` to `bounds`.

We also have constraints added for each term within the larger clause, in lines 15 through 18. These are our clause satisfiability constraints, to make sure that at least one literal in each clause is true. For each clause we, again, add a single row, as in line 16. We access this last added row, and assign to its `matrix` attribute. In this case, we are specifying that the row's constraint coefficients should be 1.0 for each column variable corresponding to each literal within this clause. Finally, we set the `bounds` attribute for this row, establishing the lower bound `1` and upper bound `None`. An assignment of `None` indicates unboundedness in this direction.

In line 19, we finally employ the simplex solver to attempt to solve a relaxed version of basic the problem. The problem is relaxed in the sense that the variables can be non-integers. The simplex solver does not respect the `kind` set to the column

variable, treating the column variables as all continuous real valued variables. However, it does respect the $[0, 1]$ bounds on the column variables implied by the `bool` kind on the bounds. We run the simplex solver first because the integer optimization method requires an existing optimal basic solution, e.g., a simplex solution.

Line 20 is a quick assertion check to ensure there are no problems with the simplex solver. The `simplex` method and other solver codes typically return `None`, unless the method was unable to start the search due to a fault in the problem definition (which returns the string `'fault'`), or because the simplex search terminated prematurely (due to one of several possible conditions).

In a real application, one would probably be interested in writing code in a fault tolerant matter to see what went wrong in an attempt to solve the problem, and then attempt to solve it. However, for this toy example, we just noisily fail with an exception. Note that "not terminating prematurely" does not indicate that an optimal solution was found. For instance, the solver could have determined that a solution did not exist, perhaps due to unboundedness or infeasibility. It merely means that the search did not terminate abnormally, or incorrectly. In order to check whether we found an optimal solution – as opposed to, say, having determined that the problem is infeasible – we check the `status` attribute, as we do in line 21. If it does not hold `'opt'`, that is, that an optimal solution was found, then we return `None` to indicate that we could not find a satisfying assignment. Note that we return `None` in this case because, if there is no solution to the relaxed unconstrained problem, there is certainly no solution to the constrained version. There could be a solution to the relaxed problem and no solution to the integer problem, but not vice versa.

After calling the simplex solver, we hold an optimal basic solution to the relaxed problem. We solve the problem as an integer problem in line 22. This is very similar to our invocation of the `simplex` solver, except this time we are using the `integer` solver. Again, in line 23, we fail noisily if we encounter something unexpected, and in line 24 quietly return `None` if we could not find a satisfying integer assignment.

The function we are writing is supposed to return a satisfying truth assignment to all our variables if such an assignment is possible. Since we have gotten this far without returning `None` or throwing an exception, we know we have a satisfying assignment. In particular, a variable is true if its positive literal has a corresponding column variable of 1.

Remember that literal x_1 corresponds to column 0, x_2 to column 2, x_3 to column 4, x_i to column $2(i-1)$, and so forth. We go over each of the even columns (using the slice `::2` to indicate every column from beginning to end, counting by 2s), test whether the value of this columns variable is `1`, and return the resulting list as our satisfying assignment in line 25.

B.3.2 Example Run

How does this work? Recall our CNF formula.

$$(\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \tag{B.5}$$

This has the encoding

```
[(-1, -3, -4), (2, 3, -4), (1, -2, 4), (1, 3, 4), (-1, 2, -3)]
```

Suppose we run this in our Python interpreter.

```
exp = [(-1, -3, -4), (2, 3, -4), (1, -2, 4), (1, 3, 4), (-1, 2, -3)]
print solve_sat(exp)
```

This prints out:

```
[True, True, False, False]
```

So, $x_1 = True$, $x_2 = True$, $x_3 = False$, and $x_4 = False$. Is this a satisfying assignment? The first and second clauses are true because $\neg x_4$. The third and fourth clauses are true because x_1. The fifth (last) clause is true because x_2. Since all the clauses are true, the conjunction is true as well, so the expression is satisfied with this variable assignment.

Now, suppose we input the expression $x_1 \wedge \neg x_1$, which is plainly unsatisfiable.

```
exp = [(-1,), (1,)]
print solve_sat(exp)
```

This prints out:

```
None
```

This value indicates that the expression is unsatisfiable, which is what we want.

B.4 Conclusions

In this chapter we have provided a brief functional introduction to PyGLPK, a Python encapsulation of the existing GNU Linear Programming Kit. The PyGLPK presents an interface to GLPK which encapsulates nearly all of the documented functionality of the GLPK in a Pythonic interface. The explanation of PyGLPK

focused on examples of using PyGLPK to solve specific problems: a small two dimensional example, and an implementation of a SAT solver. We illustrated how to set up a problem, add constraints, optimize, and retrieve solution values in both linear and integer problems.

BIBLIOGRAPHY

[1] Charu C. Aggarwal, Stephen C. Gates, and Philip S. Yu. On the merits of building categorization systems by supervised clustering. In *ACM SIGKDD-1999*, pages 352–356. ACM Press, 1999.

[2] Yasemin Altun, David McAllester, and Mikhail Belkin. Maximum margin semi-supervised learning for structured variables. In Yair Weiss, Bernhard Schölkopf, and John Platt, editors, *Advances in Neural Information Processing Systems (NIPS) 18*, pages 33–40, Cambridge, MA, 2006. MIT Press.

[3] Yasemin Altun, Ioannis Tsochantaridis, and Thomas Hofmann. Hidden Markov support vector machines. In *ICML '03: Proceedings of the 20th international conference on Machine learning*, pages 3–10, 2003.

[4] Dragomir Anguelov, Ben Taskar, Vassil Chatalbashev, Daphne Koller, Dinkar Gupta, Geremy Heitz, and Andrew Ng. Discriminative learning of Markov random fields for segmentation of 3D scan data. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 2*, volume 2, pages 169–176, Washington, DC, USA, 2005. IEEE Computer Society.

[5] Francis R. Bach and Michael I. Jordan. Learning spectral clustering. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems (NIPS) 16*. MIT Press, 2003.

[6] Francis R. Bach and Michael I. Jordan. Learning spectral clustering, with application to speech separation. *Journal of Machine Learning Research*, 7:1963–2001, 2006.

[7] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2002.

[8] Sugato Basu, Mikhail Bilenko, and Raymond J. Mooney. A probabilistic framework for semi-supervised clustering. In *ACM SIGKDD-2004*, pages 59–68, August 2004.

[9] J. C. Bezdek. Pattern recognition with fuzzy objective function algoritms. 1981.

[10] Mikhail Bilenko, Sugato Basu, and Raymond J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *ICML '04: Proceedings of the 21st international conference on Machine learning*, New York, NY, USA, 2004. ACM Press.

[11] Endre Boros and Peter L. Hammer. Pseudo-boolean optimization. *Discrete Appl. Math.*, 123(1-3):155–225, 2002.

[12] Matthew R. Boutell, Jiebo Luo, Xipeng Shen, and Christopher M. Brown. Learning multi-label scene classification. *Pattern Recognition*, 37(9):1757–1771, 2004.

[13] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *PAMI*, 26(9):1124–1137, 2004.

[14] Ulf Brefeld and Tobias Scheffer. Semi-supervised learning for structured output variables. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 145–152, New York, NY, USA, 2006. ACM Press.

[15] Claire Cardie and Kiri Wagstaff. Noun phrase coreference as clustering. In *Joint Conference on Empirical Methods in NLP and Very Large Corpora*, 1999.

[16] Nicolò Cesa-Bianchi, Claudio Gentile, and Luca Zaniboni. Hierarchical classification: combining Bayes with SVM. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, 2006.

[17] Chih-Chung Chang and Chih-Jen Lin. LIBSVM : A library for support vector machines, 2001. Software at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[18] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, editors. *Semi-Supervised Learning (Adaptive Computation and Machine Learning)*. The MIT Press, September 2006.

[19] Olivier Chapelle and Alexander Zien. Semi-supervised classification by low density separation. In *Tenth International Workshop on Artificial Intelligence and Statistics*, pages 57–64, 01 2005.

[20] Ira Cohen, Nicu Sebe, Fabio G. Cozman, and Thomas S. Huang. Semi-supervised learning for facial expression recognition. In *MIR '03: Proceedings*

of the 5th ACM SIGMM international workshop on Multimedia information retrieval, pages 17–22, New York, NY, USA, 2003. ACM Press.

[21] William Cohen and Jacob Richman. Learning to match and cluster entity names. In *ACM SIGIR'01 Workshop on Mathematical/Formal Methods in IR*, 2001.

[22] David Cohn, Rich Caruana, and Andrew McCallum. Semi-supervised clustering with user feedback, April 2001.

[23] Michael Collins. Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *ACL-EMNLP*, pages 1–8, Morristown, NJ, USA, 2002. Association for Computational Linguistics.

[24] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, January 1967.

[25] Mark Craven, Dan DiPasquo, Dayne Freitag, Andrew McCallum, Tom Mitchell, Kamal Nigam, and Seán Slattery. Learning to extract symbolic knowledge from the world wide web. In *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 509–516, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[26] Aron Culotta, Michael Wick, and Andrew McCallum. First-order probabilistic models for coreference resolution. In *NAACL-HLT*, pages 81–88, 2007.

[27] Hal Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, Los Angeles, CA, August 2006.

[28] Hal Daumé III, John Langford, and Daniel Marcu. Search-based structured prediction. 2006.

[29] Hal Daumé III, John Langford, and Daniel Marcu. Searn in practice. Tech Report, 2006.

[30] Hal Daumé III and Daniel Marcu. A Bayesian model for supervised clustering with the Dirichlet process prior. *Journal of Machine Learning Research*, 6:1551–1577, 2005.

[31] Jason V. Davis, Brian Kulis, Prateek Jain, Suvrit Sra, and Inderjit S. Dhillon.

Information-theoretic metric learning. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 209–216, New York, NY, USA, 2007. ACM.

[32] Tijl De Bie, M. Momma, and Nello Cristianini. Efficiently learning the metric using side-information. In *ALT2003*, volume 2842, pages 175–189. Springer, 2003.

[33] Erik Demaine and Nicole Immorlica. Correlation clustering with partial information. In *RANDOM-APPROX 2003*, pages 1–13, August 2003.

[34] A.P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[35] Inderjit S. Dhillon, Yuqiang Guan, and J. Kogan. Iterative clustering of high dimensional text data augmented by local search. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, page 131, Washington, DC, USA, 2002. IEEE Computer Society.

[36] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. A unified view of kernel k-means, spectral clustering and graph cuts. Technical Report TR-04-25, University of Texas Dept. of Computer Science, 2005.

[37] J. C. Dunn. A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 1973.

[38] André Elisseeff and Jason Weston. A kernel method for multi-labelled classification. In *Advances in Neural Information Processing Systems (NIPS) 14*, pages 681–687, Cambridge, MA, 2002. MIT Press.

[39] Thomas Finley. SVM^{python}, 2007. `http://www.cs.cornell.edu/~tomf/svmpython2/`.

[40] Thomas Finley and Thorsten Joachims. Supervised clustering with support vector machines. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 217–224, New York, NY, USA, 2005. ACM Press.

[41] Thomas Finley and Thorsten Joachims. Supervised k-means clustering. Number 1813-11621, 2008. `http://hdl.handle.net/1813/11621`.

[42] Chris Fraley and Adrian E. Raftery. Model-based clustering, discriminant analysis, and density estimation. *Journal of the American Statistical Association*, pages 611–631, June 2002.

[43] Bogdan Gabrys and Lina Petrakieva. Combining labelled and unlabelled data in the design of pattern classification systems. *International Journal of Approximate Reasoning*, 35(3):251–273, 2004.

[44] Yuhong Guo, Dana Wilkinson, and Dale Schuurmans. Maximum margin bayesian networks, 2005.

[45] Peter Haider, Ulf Brefeld, and Tobias Scheffer. Supervised clustering of streaming data for email batch detection. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 345–352, New York, NY, USA, 2007. ACM.

[46] Peter L. Hammer, Pierre Hansen, and Bruno Simeone. Roof-duality, complementation, and persistency in quadratic 0–1 optimization. *Math. Program.*, 28:121–155, 1984.

[47] Xuming He, Richard S. Zemel, and Miguel A. Carreira-Perpinan. Multiscale conditional random fields for image labeling. *cvpr*, 02:695–702, 2004.

[48] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, 2002.

[49] Feng Jiao, Shaojun Wang, Chihoon Lee, Russ Greiner, and Dale Schuurmans. Semi-supervised conditional random fields for improved sequence segmentation and labeling. In *Proceedings of Coling/ACL 2006*, Sydney, Australia, July 17-21 2006.

[50] Thorsten Joachims. Learning to align sequences: A maximum-margin approach. Technical report, August 2003.

[51] Thorsten Joachims. A support vector method for multivariate performance measures. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 377–384, New York, NY, USA, 2005. ACM Press.

[52] Thorsten Joachims. Training linear SVMs in linear time. In *ACM SIGKDD-2006*, pages 217–226, New York, NY, USA, 2006. ACM.

[53] Thorsten Joachims, Thomas Finley, and Chun-Nam John Yu. Cutting-plane training of structural SVMs. *Machine Learning Journal*, 2008, to appear.

[54] Thorsten Joachims and John Hopcroft. Error bounds for correlation clustering. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 385–392, New York, NY, USA, 2005. ACM Press.

[55] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, (2):241–254, 1967.

[56] Toshihiro Kamishima and Fumio Motoyoshi. Learning from cluster examples. *Machine Learning*, 53(3):199–233, December 2003.

[57] Jon M. Kleinberg. Hubs, authorities, and communities. *ACM Computing Surveys*, 31(4), December 1999.

[58] Vladimir Kolmogorov and Carsten Rother. Minimizing non-submodular functions with graph cuts – a review. *PAMI*, 26(2):147–159, 2004.

[59] Alex Kulesza and Fernando Pereira. Structured learning with approximate inference. In J.C. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems (NIPS) 20*, pages 785–792, Cambridge, MA, 2008. MIT Press.

[60] Brian Kulis, Sugato Basu, Inderjit Dhillon, and Raymond Mooney. Semi-supervised graph clustering: a kernel approach. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 457–464, New York, NY, USA, 2005. ACM.

[61] Sanjiv Kumar and Martial Hebert. Discriminative random fields: A discriminative framework for contextual interaction in classification. In *Proceedings of the 2003 IEEE International Conference on Computer Vision (ICCV '03)*, volume 2, pages 1150–1157, 2003.

[62] Sanjiv Kumar and Martial Hebert. Discriminative fields for modeling spatial dependencies in natural images. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems (NIPS) 16*, Cambridge, MA, 2004. MIT Press.

[63] Simon Lacoste-Julien, Ben Taskar, Dan Klein, and Michael I. Jordan. Word alignment via quadratic assignment. In *Proceedings of the Human Language*

Technology Conference of the NAACL, Main Conference, pages 112–119, New York City, USA, June 2006. Association for Computational Linguistics.

[64] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML '01: Proceedings of the 18th international conference on Machine learning*, 2001.

[65] Gert R. G. Lanckriet, Nello Christianini, Peter L. Bartlett, Laurent El Ghaoui, and Michael I. Jordan. Learning the kernel matrix with semi-definite programming. In *ICML '02: Proceedings of the 19th international conference on Machine learning*, pages 323–330, 2002.

[66] Gert R. G. Lanckriet, Nello Cristianini, Peter Bartlett, Laurent El Ghaoui, and Michael I. Jordan. Learning the kernel matrix with semidefinite programming. *Journal of Machine Learning Research*, 5:27–72, 2004.

[67] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, 2004.

[68] Jai Li, Surajit Ray, and Bruce G. Lindsay. A nonparametric statistical approach to clustering via mode identification. *Journal of Machine Learning Research*, 8(8):1687–1723, 2007.

[69] Percy Liang, Alexandre Bouchard-Côté, Dan Klein, and Benjamin Taskar. An end-to-end discriminative approach to machine translation. In *ACL*, 2006.

[70] Percy Liang, Ben Taskar, and Dan Klein. Alignment by agreement. In *Proceedings of the Human Language Technology Conference of the NAACL, Main Conference*, pages 104–111, New York City, USA, June 2006. Association for Computational Linguistics.

[71] Andrew Makhorin. GNU linear programming kit, version 4.31, 2008. `http://www.gnu.org/software/glpk/glpk.html`.

[72] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing*. MIT Press, Cambridge, MA, 1999.

[73] D. Martin, C. Fowlkes, D. Tal, and J. Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and

measuring ecological statistics. In *Proc. 8th Int'l Conf. Computer Vision*, volume 2, pages 416–423, July 2001.

[74] Andrew McCallum, Dayne Freitag, and Fernando Pereira. Maximum entropy markov models for information extraction and segmentation. In *ICML '00: Proceedings of the 17th international conference on Machine learning*, pages 591–598. Morgan Kaufmann, 2000.

[75] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *ACM SIGKDD-2000*, pages 169–178, 2000.

[76] Andrew McCallum and Ben Wellner. Toward conditional models of identity uncertainty with application to proper noun coreference. In *IJCAI Workshop on Information Integration on the Web*, 2003.

[77] MUC-6. In *Proceedings of the Sixth Message Understanding Conference (MUC-6)*. Morgan Kaufmann, 1995.

[78] Vincent Ng and Claire Cardie. Improving machine learning approaches to coreference resolution. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 104–111, 2002.

[79] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[80] Joao Pedro Pedroso. Python-glpk, 2007. `http://www.dcc.fc.up.pt/~jpp/code/python-glpk/`.

[81] Minh-Tri Pham. Ctypes-glpk: A python wrapper for glpk using ctypes, 2007. `http://ctypes-glpk.googlecode.com/`.

[82] V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *Advances in Neural Information Processing Systems (NIPS) 13*, pages 995–1001. MIT Press, 2001.

[83] V. Punyakanok, D. Roth, and W. Yih. The necessity of syntactic parsing for semantic role labeling. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1117–1123, 2005.

[84] V. Punyakanok, D. Roth, W. Yih, and D. Zimak. Learning and inference

over constrained output. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1124–1129, 2005.

[85] Yuan Qi, Martin Szummer, and Thomas P. Minka. Bayesian conditional random fields. In *AI & Statistics*, January 2005.

[86] Ashish Raj, Gurmeet Singh, and Ramin Zabih. MRF's for MRI's: Bayesian reconstruction of MR images via graph cuts. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1061–1068, Washington, DC, USA, 2006. IEEE Computer Society.

[87] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(366):846–850, 1971.

[88] Philippe Rigollet. Generalization error bounds in semi-supervised classification under the cluster assumption. *Journal of Machine Learning Research*, 8:1369–1392, 2007.

[89] Gian-Carlo Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[90] D. Roth. Reasoning with classifiers. In *Proc. of the European Conference on Machine Learning (ECML)*, pages 506–510, 2001.

[91] D. Roth and W. Yih. Probabilistic reasoning for entity and relation recognition. In *Proc. the International Conference on Computational Linguistics (COLING)*, pages 835–841, 2002.

[92] D. Roth and W. Yih. Integer linear programming inference for conditional random fields. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, 2005.

[93] Jean-Sébastien Roy. PuLP: A linear programming modeler in python, 2005. `http://www.jeannot.org/~js/code/index.en.html#PuLP`.

[94] Sunita Sarawagi and Rahul Gupta. Accurate max-margin training for structured output spaces. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 888–895, New York, NY, USA, 2008. ACM.

[95] Matthew Schultz and Thorsten Joachims. Learning a distance metric from

relative comparisons. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 16*. MIT Press, Cambridge, MA, 2004.

[96] Matthias Seeger. Learning with labeled and unlabeled data. Technical report, Institute for ANC, Edinburgh, UK, 2000.

[97] Shai Shalev-Shwartz and Nathan Srebro. Svm optimization: inverse dependence on training set size. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 928–935, 2008.

[98] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 731, Washington, DC, USA, 1997. IEEE Computer Society.

[99] Cees G. M. Snoek, Marcel Worring, Jan C. van Gemert, Jan-Mark Geusebroek, and Arnold W. M. Smeulders. The challenge problem for automated detection of 101 semantic concepts in multimedia. In *ACM-MULTIMEDIA*, 2006.

[100] Wee Meng Soon, Hwee Tou Ng, and Daniel Chung Yong Lim. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544, 2001.

[101] Charles Sutton and Andrew McCallum. Fast, piecewise training for discriminative finite-state and parsing models. Technical Report IR-403, Center for Intelligent Information Retrieval, 2005.

[102] Chaitanya Swamy. Correlation clustering: maximizing agreements via semidefinite programming. In *ACM-SIAM SODA*, pages 526–527. Society for Industrial and Applied Mathematics, 2004.

[103] Ben Taskar, Vassil Chatalbashev, and Daphne Koller. Learning associative Markov networks. In *ICML '04: Proceedings of the 21st international conference on Machine learning*, page 102, New York, NY, USA, 2004. ACM.

[104] Ben Taskar, Carlos Guestrin, and Daphne Koller. Max-margin Markov networks. In *Advances in Neural Information Processing Systems (NIPS) 16*. 2003.

[105] Ivor W. Tsang and James T. Kwok. Distance metric learning with kernels.

In *International Conference on Artificial Neural Networks (ICANN)*, pages 126–129, June 2003.

[106] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML '04: Proceedings of the 21st international conference on Machine learning*, 2004.

[107] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. Large margin methods for structured and interdependent output variables. *JMLR*, 6:1453–1484, 2005.

[108] Marc Vilain, John Burger, John Aberdeen, Dennis Connolly, and Lynette Hirschman. A model-theoretic coreference scoring scheme. In *MUC-6*, pages 45–52. Morgan Kaufmann, 1995.

[109] S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, and Kevin P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, 2006.

[110] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schroedl. Constrained k-means clustering with background knowledge. In *ICML '01: Proceedings of the 18th international conference on Machine learning*, 2001.

[111] Kilian Q. Weinberger, John Blitzer, and Lawrence K. Saul. Distance metric learning for large margin nearest neighbor classification. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Advances in Neural Information Processing Systems (NIPS) 18*, pages 1473–1480, Cambridge, MA, 2006. MIT Press.

[112] Eric Xing, Andrew Y. Ng, Michael Jordan, and Stuart Russell. Distance metric learning, with application to clustering with side-information. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems (NIPS) 15*, volume 15, pages 505–512. MIT Press, 2003.

[113] Chun-Nam Yu and Thorsten Joachims. Training structural svms with kernels using sampled cuts. In *ACM SIGKDD-2008*, August 2008.

[114] Chun-Nam Yu, Thorsten Joachims, Ron Elber, and Jaroslaw Pillardy. Support vector training of protein alignment models. In *RECOMB*, 2007.

[115] Yisong Yue, Thomas Finley, Filip Radlinski, and Thorsten Joachims. A support vector method for optimizing average precision. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 271–278, New York, NY, USA, 2007. ACM.

[116] Yisong Yue and Thorsten Joachims. Predicting diverse subsets using structural svms. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 1224–1231, New York, NY, USA, 2008. ACM.