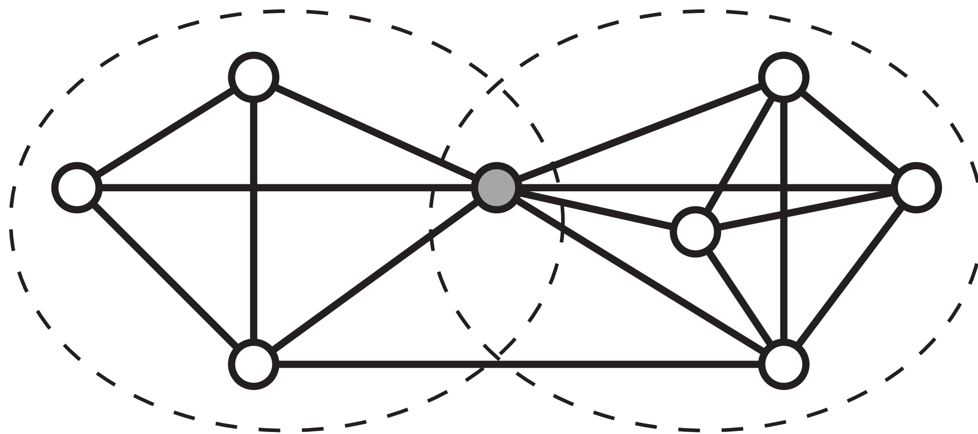
CHALMERS



Entity Disambiguation in Anonymized Graphs Using Graph Kernels

Master of Science Thesis in the Programme Computer Science: Algorithms, Languages and Logic

LINUS HERMANSSON
TOMMI KEROLA

Chalmers University of Technology
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2013.

The Authors grant to Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet. The Authors warrant that they are the authors to the Work, and warrant that the Work does not contain text, pictures or other material that violates copyright law.

The Authors shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Authors has signed a copyright agreement with a third party regarding the Work, the Authors warrant hereby that they have obtained any necessary permission from this third party to let Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

Entity Disambiguation in Anonymized Graphs Using Graph Kernels

LINUS HERMANSSON
TOMMI KEROLA

©LINUS HERMANSSON, June 2013.
©TOMMI KEROLA, June 2013.

Examiner: DEVDATT DUBHASHI

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Local neighborhood graph of ambiguous vertex *Chris Anderson*, as in the example of Chapter 1.

Department of Computer Science and Engineering
Gothenburg, Sweden, June 2013.

Abstract

In recent years, the explosion of available online information has brought forth new data mining applications into the spotlight, such as automated querying about real-world entities. This requires extraction of identifiers such as names and places from text. The problem, however, is complicated by the non-uniqueness of identifiers. A motivating example is the name *Chris Anderson*, which could either refer to Chris Anderson, the curator of TED Talks, or Chris Anderson, the former editor-in-chief of WIRED Magazine. Both individuals work in overlapping fields, and deciding whom is referred to could be a difficult task, even when considering context. Correctly identifying and resolving such ambiguous identifiers is crucial for enabling such applications to advance from the research lab into practical usage.

This master's thesis presents a novel method for entity disambiguation in anonymized graphs based on local neighborhood structure. Most existing approaches leverage node information, which might not be available in several contexts due to privacy concerns, or information about the sources of the data. We consider this problem in the supervised setting where we are provided only with a base graph and a set of nodes labeled as ambiguous or unambiguous. We characterize the similarity between two nodes based on their local neighborhood structure using graph kernels; and efficiently solve the resulting classification task using a support vector machine (SVM), a standard machine learning technique.

Leveraging kernels is a powerful method for extending linear SVM classifiers to non-linear classification tasks. Recently, a number of graph kernels have been proposed for classifying graph structures. In this thesis, we present extensions of two existing graphs kernels, namely, the direct product kernel and the shortest-path kernel, with significant improvements in accuracy. For the direct product kernel, our extension also provides significant computational benefits.

A key concern today is scalability of algorithms to web-scale datasets. This poses new challenges for designing new machine learning methods. We use GraphLab, a framework for distributed computing, to allow our extended kernels to be computed in parallel. This ensures scalability and allows our method to handle large-size data.

We test our method on two real-world datasets, comparing our approach to a state-of-the-art method. We show that using less information, our method is significantly better in terms of either speed or accuracy or both.

Acknowledgments

We would like to express our gratitude towards our supervisor Vinay Jethava and also Fredrik Johansson for their motivating discussions and inspiring ideas. A word of thanks goes to Devdatt Dubhashi for providing helpful comments and curiosity in the work. We would like to thank Recorded Future, a software company, and their employees for their insight and help and for providing data. We would also like to thank Bhavishya Goel and Jacob Lidman for helping us with setting up the experiments on a computer cluster.

Contents

1 Introduction **1**

1.1 Purpose and Aims 4
1.2 Scope 4
1.3 Method 5
1.4 Thesis Outline 6

2 Background **7**

2.1 GraphLab 7
2.1.1 Gather, Apply, Scatter 8
2.2 Support Vector Machines 9
2.2.1 Duality 11
2.2.2 Kernels 13
2.3 The Subgradient Method 15
2.4 Pegasos: Primal Estimated sub-GrAdient SOlver for SVM 16
2.4.1 Kernelized Version 17
2.5 Kronecker Product 18
2.6 Direct Product Graph 19
2.7 Graph Kernels 20
2.7.1 Direct Product Kernel 20
2.7.2 Shortest Path Kernel 20
2.7.3 Graphlet Kernel 21
2.8 Malin's Random Walk Method 21

3 Related Research **26**

4 Our Approach **29**
4.1 Method Overview . . . 30
4.2 Kernel Extensions . . . 32
4.2.1 Truncated Direct Product Kernel . . . 32
4.2.2 Binned Shortest Distance Kernel . . . 35
4.2.3 Normalization . . . 36
4.3 Enabling Fast Computation . . . 36
4.3.1 Explicit Knowledge of ϕ . . . 36
4.3.2 Batch Computation . . . 37
4.4 Graph Kernel Implementations . . . 37
4.4.1 Truncated Direct Product Kernel . . . 38
4.4.2 Binned Shortest Distance Kernel . . . 39

5 Applications **43**

6 Experiments **44**
6.1 Recorded Future News Data . . . 44
6.2 Internet Movie Database . . . 45
6.3 Experimental Setup . . . 46
6.4 Results . . . 47

7 Discussion **54**

8 Conclusions and Future Work **56**

A Mathematical Notation **58**

B Mathematical Proofs **60**
B.1 Proof of Equation 2.18, p. 19 . . . 60
B.2 Proof of Equation 2.19, p. 19 . . . 61
B.3 Derivation of Equation 4.10, p. 35 . . . 61
B.4 Derivation of Equation 4.5, p. 33 . . . 62

List of Figures

1.1 Local neighborhood example. 2

2.1 Hyperplane separating two linearly separable classes. 9
2.2 Non-linear kernel map example. 14
2.3 Original graph in Malin's method. 24
2.4 Graph focused on node of interest in Malin's method. 24

4.1 Local neighborhood example. (Restated) 30
4.2 Method overview. 31

6.1 Histogram of edge weights in the RF and IMDb datasets. 52
6.2 Parallel speedup of BSD kernel. 53

List of Tables

6.1 Classification accuracy on experiments. 47
6.2 Timing results in experiments. 48

List of Algorithms

1 The Pegasos algorithm. . . . 17
2 Kernelized version of Pegasos. . . . 18
3 Malin's random walk method. . . . 22
4 Detect ambiguous nodes. . . . 32
5 Calculate the number of random walks. . . . 33
6 Detect ambiguous nodes. Fast version. . . . 37
7 TDP kernel ϕ-calculation in GraphLab. . . . 39
8 TDP vertex program gather phase. . . . 39
9 TDP vertex program apply phase. . . . 39
10 BSD kernel ϕ-calculation in GraphLab. . . . 41
11 BSD vertex program gather phase. . . . 41
12 BSD vertex program apply phase. . . . 41
13 BSD vertex program scatter phase. . . . 42

1

Introduction

IN RECENT YEARS, THE EXPLOSION of available online information has brought forth new data mining applications into the spotlight. In combination with modern machine learning techniques, this allows for extraction of vast amounts of information about real-world entities, such as people, places or companies. For example, a user might be interested to know which cities Barack Obama is going to visit this year. A machine is able to answer such a request due to increasingly refined machine learning methods. Such a system requires automatic extraction of identifiers such as names and places from text. The problem, however, is complicated further by the non-uniqueness of identifiers. A motivating example is the name *Chris Anderson*, which could either refer to Chris Anderson, the curator of TED Talks, or Chris Anderson, the former editor-in-chief of WIRED Magazine. TED Talks is an organization hosting talks about subjects such as technology, entertainment and design [1]. WIRED Magazine is a magazine covering how new technology affects culture, economy and politics [2]. Thus, both individual Chrises work in overlapping fields, and deciding whom is referred to could be a difficult task, even when considering context. Correctly identifying and resolving such ambiguous identifiers is crucial for enabling such applications to advance from the research lab into practical usage.

Resolving ambiguities in data is a well-studied problem, including methods for entity resolution [5, 6, 24, 46], entity matching [9, 42] and entity disambiguation [16, 21,

[1] http://www.ted.com/
[2] http://www.wired.com/magazine/

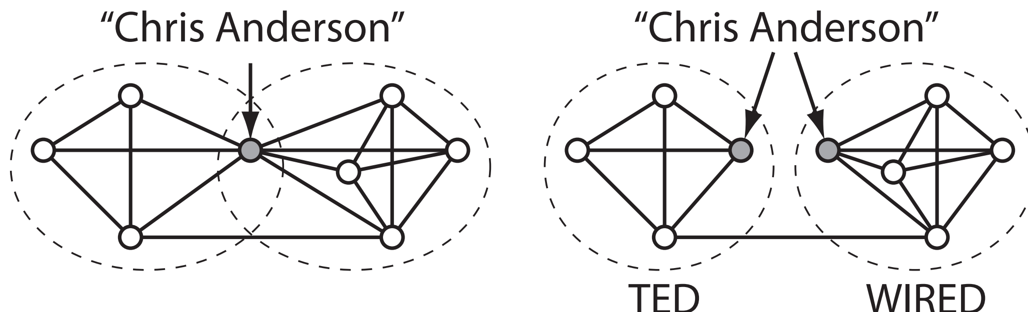


Figure 1.1: (Left) Local neighborhood (fictive) of the ambiguous vertex Chris Anderson. (Right) Correct splitting of the vertex into its two true underlying entities.

22, 39]. These are all aimed towards associating references in text sources with their correct underlying entities. These methods typically make use of similarities in names [5, 6], meta-data [9] or source information [39], to decide which entities underly which references. *Relational entity disambiguation* makes use of network structure between entities [3, 4, 5, 39], sometimes together with additional information.

Big data analysis increasingly faces the challenge of how to preserve user anonymity [1]. While a number of privacy preserving mechanisms have been studied [23], the practical applications are still at a nascent stage [35]. Often, a simple approach to address this privacy concern is using anonymization at source, prior to subsequent data mining, by assigning pseudorandom identifiers to entities. In this setting, existing techniques [5, 6, 9, 16, 21, 22, 24, 42, 46], building on similarity of entity attributes, are rendered inapplicable, while the method presented in this thesis is inherently well-suited. To the best of our knowledge, very little work has been done in this setting. Moreover, our approach learns the nature of ambiguous nodes from the data, making it suitable for more advanced anonymization techniques such as *k-Degree* anonymization [35].

In this thesis, we present a novel method for anonymized relational entity disambiguation that leverages graph structure for detecting ambiguous identifiers. In our setting we have a graph where the entities have been assigned a (possibly ambiguous) identifier. A node in the graph represents one or several underlying entities, and an edge symbolizes a connection between identifiers, such as co-occurrence in articles. It should be noted that in our setting, graph structure and edge weights are the only data available. We target the scenario where the full original data is unavailable or expensive to access. Such situations occur when large amounts of data are parsed in an online fashion, not to be looked at again. We direct our attention to Figure 1.1,

illustrating our problem with the above mentioned Chris Anderson. In the figure, the *two* individuals have been assigned *one* common identifier, incorrectly merging them into a single node in the graph. This error inevitably creates a strong connection between the communities of TED and WIRED, something that we might not expect to hold in reality.

In this thesis, we introduce a novel formulation of relational entity disambiguation as a classification problem in an anonymized graph. The graph contains two classes of nodes, with some labeled as either *ambiguous* (several underlying entities) or *unambiguous* (only one underlying entity). We seek to detect ambiguous nodes in such a graph, predicting new nodes to belong to one of the two above classes. Unlike most other related research, our method works without using meta-data that reveals node information. Our method works in the anonymized setting, making it suitable for scenarios where privacy is an issue.

The method in this thesis solves our classification problem by leveraging kernel methods and a highly scalable support vector machine (SVM) implementation. Given a graph with some nodes labeled as ambiguous or unambiguous, our method trains an SVM classifier based on graph kernels, using the local neighborhood of labeled nodes as input. The method is designed to operate in a distributed setting and is implemented in GraphLab, a framework for doing distributed computation. The usage of GraphLab allows for web-scale data to be processed.

Additionally, we extend two existing graph kernels, designing a domain-specific adaptation of the shortest path kernel [10], and a fast appropriate algorithm for the direct product kernel [27] for unlabeled graphs. We show both theoretical and empirical evidence of the computational benefits of the extensions, as compared with the original graph kernels. We evaluate our method on two real-world datasets, comparing the performance of different graph kernels, showing that our extensions outperform their original counterparts both in terms of speed and accuracy. Our experimental results also show that implementing our kernels in GraphLab gives a significant speedup.

We compare our method against a state-of-the-art method [39] at our task of detecting ambiguous nodes. Our experiments on a well-studied public dataset show that our method is significantly better than the state-of-the-art, either in terms of speed or accuracy or both. Moreover, our method requires less information as it does not demand the existence of data sources.

1.1 Purpose and Aims

With this thesis we aim to explore relational entity disambiguation in an anonymized setting using graph kernels. The purpose of this is to create discussion in the field of entity disambiguation as well as to contribute to current graph kernel research.

Our goals can be summarized by the following three points.

- Apply recent research and implement a scalable method for relational entity disambiguation.
- Generalize and improve upon the above method.
- Perform experiments on real-world anonymized data.

1.2 Scope

There has been a lot of research in entity disambiguation, but most methods leverage node information, and are thus not usable in the anonymized setting. Our focus is not on such models, or the comparison of them. Rather, we limit ourselves to anonymized relational entity disambiguation in the supervised setting. We wish to introduce graph kernels into the field of entity disambiguation, and also create a competitor to other models.

While there exist methods for associating ambiguous identifiers with their correct underlying entities, we do not pursue them here. We limit ourselves as to only focus on the problem of detecting ambiguous nodes, leaving the actual association process as future work.

We limit ourselves to develop our method in the setting where the original source data is not available. For comparison with the state-of-the-art, we look at a paper by Malin [39], which to the best of our knowledge, contains the only methods that are applicable in our setting with limited information. In Malin [39], two methods applicable to our problem are presented; one based on random walks and one based on hierarchical clustering. Since the random walk method was reported to give the best result, we use it as a state-of-the-art comparison for our approach. We denote this method MALIN. As MALIN demands knowledge of sources, we make an exception and allow it usage of sources, as it is our only means of comparison. It should be

noted that when doing the comparative experiments, our method will never be able to see the source data.

We limit our choice of graph kernels for investigation to the shortest path kernel [10], direct product kernel [27] and graphlet kernel [52] and their extensions. For completeness, we note that several other graph kernels exist, such as Weisfeiler-Lehman kernels [51] and fast subtree kernels [50]. We do however not investigate these graph kernels in this thesis.

1.3 Method

This thesis project has been carried out as a collaboration between Chalmers and Recorded Future, a web intelligence company, who provided us with one of the datasets. The authors became interested in the thesis work partly because it was in close connection with current research at Chalmers, and partly because the topic concerned a real problem that companies such as Recorded Future experience on a daily basis.

When starting the project, we began by studying relevant theory. We found Stephen Boyd's [3] two online courses *Convex Optimization I* and *Convex Optimization II* particularly helpful for establishing a solid theoretical ground in optimization. For machine learning and SVMs, we found Tristan Fletcher's tutorial *SVM Explained* [25] very helpful as an initial starting point. We also studied various other sources in order to be prepared before starting on the central part of the project.

Our method is based on SVMs and the GraphLab framework, making these a natural starting point for the thesis. We study these concepts in Chapter 2. It is crucial for us to get a full understanding of these in order to implement our method and to be able to extend it.

We also need to study various graph kernels in order to be able to implement these for our method, and also later in the thesis, extend them.

A related topic is the state-of-the-art SVM solver Pegasos [48], which is important for us to study in order to create a fast and scalable method. We initially set foot towards creating a parallel implementation of Pegasos in GraphLab, but discovered later that our graph kernel extensions had properties for making a parallel implementation unnecessary. More information about this can be found in Chapter 4.

[3] `http://www.stanford.edu/~boyd/`

All graph kernels except the direct product and graphlet kernel were implemented in GraphLab in `C++`. The direct product kernel and also MALIN was implemented in Python. For the graphlet kernel, we used MATLAB. This was due to time constraints.

Finally, we performed a large number of experiments on two real-world datasets, comparing our different kernels as well as MALIN in terms of speed and accuracy. One of the datasets use Recorded Future data, while the other one was parsed from raw data by ourselves. At the initial learning stage of the project, we performed experiments on synthetic datasets, but as we found these results uninteresting, we decided to only focus on the more challenging real-world datasets for this thesis.

As part of the project, we also wrote a paper, which is to be presented at the ICML workshop [4] SLG 2013 in Atlanta, USA.

1.4 Thesis Outline

The rest of this thesis is outlined as follows.

In Section 2, we give a detailed account of the theoretical background on which our approach is based.

Section 3 introduces research related to our problem. We compare other approaches with ours and explain why some approaches are not applicable in our setting.

In Section 4 we present, in detail, our approach for relational entity disambiguation.

Section 5 presents different applications for our method.

Section 6 presents experimental results, comparing the performance of the different approaches we investigated.

Section 7 discusses results and the thesis as a whole.

Section 8 draws conclusions based on our work and presents future work.

Finally, the appendices include sections on mathematical notation as well as proofs.

[4] `https://sites.google.com/site/slgworkshop2013/cfp`

2

Background

THIS CHAPTER CONTAINS background knowledge necessary for understanding the main results of this thesis. Readers already familiar with the concepts in this chapter may skip them without affecting understanding of the final results of the thesis.

2.1 GraphLab

GraphLab is a framework for distributing graph algorithms [36, 37]. Programs for GraphLab are written in in `C++`. The important part, however, is writing the code according to the GraphLab framework. This includes calling standard GraphLab functions and writing the program as a *vertex program*. The vertex program should be written according to the GAS layout [1]. Programs that follow the GAS layout need to work in three phases: Gather, Apply and Scatter. The program needs to be constructed according to these phases. Once this is done this allows GraphLab to parallelize and scale the algorithm freely. A spin-off of GraphLab, called GraphChi, is also available [34]. GraphChi utilizes an efficient disk access algorithm in order to make computing on large graphs feasible on just a normal PC.

[1] `http://graphlab.org/home/abstraction/`

2.1.1 Gather, Apply, Scatter

In version 2.0 of GraphLab, a new design methodology for writing distributed programs was introduced. Programs in GraphLab are vertex-centric, meaning that one writes code that should be run in parallel on all vertices of a graph. A vertex program object is created for each vertex, allowing code to store private data in the vertex program. Public data, which is meant for other vertices to access however, must be stored in the vertex's own data. Based on analysis of common patterns in graph-parallel algorithms, GraphLab programs should be written in this new layout, called the *GAS layout* (gather, apply, scatter). The GAS layout consists of the following phases, which are run sequentially in parallel on each vertex:

Init
: Any messages received from adjacent vertices may be processed and saved to the vertex program. The vertex data may not be modified in this phase.

Gather
: Information from either in-, out- or in-and-out edges and their vertices are collected and summed together. Since a custom summing operator (`+=`) can be defined, the user has a lot of freedom in designing what `+=` on certain information actually should mean in practice. The vertex data may not be modified in this phase.

Apply
: The summed-together data from the gather phase is given, allowing the user to write code that modifies the vertex's and/or the vertex program's data.

Scatter
: Opposite of the gather phase, the scatter phase allows the user to write code that signals (sends messages) to adjacent vertices along in-, out- or in-and-out edges. If several messages are sent to the same vertex, the messages are summed together. Similar to the gather phase, the user is given the freedom to define the practical meaning of `+=` on messages. The vertex data may not be modified in this phase.

To run GraphLab with the GAS layout, a graph engine is first created. The graph engine can then signal either all or certain selected vertices so that the vertex program for those vertices is run, these vertices are said to be *activated*. The graph engine will run as long as there exists vertices that have incoming signals. When there is no active vertex left, the graph engine is said to have converged.

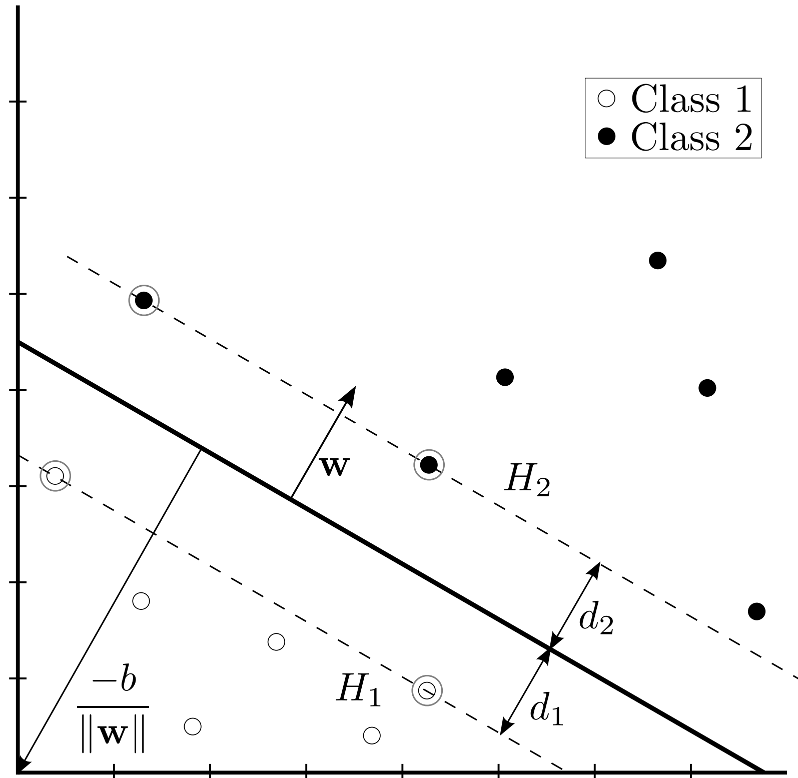


Figure 2.1: Hyperplane separating two linearly separable classes. $\mathbf{w}$ is the normal to the hyperplane and $\frac{b}{\|\mathbf{w}\|}$ is the perpendicular distance from the origin to the hyperplane. H_1 and H_2 are the hyperplanes that the support vectors lie on. Note that the support vectors lie exactly on the margin boundary. $d_1 = d_2$ is the SVM margin.

2.2 Support Vector Machines

A support vector machine (SVM) is a supervised learning model for separating two classes of input data [17, 25]. Its current form was introduced in 1992 by Boser et al. [12] and also in a 1995 paper by Cortes and Vapnik [20], where it was shown to have high generalization ability (i.e. low error rate on test sets).

It is important for a learning machine to have good generalization ability. A classifier that suffers from *overfitting* [8], meaning that it remembers the training samples far too well, is not desirable. Of course, neither is the contrary. As an example, let us draw a parallel by considering a zoologist trying to classify animals into categories. Overfitting would be exemplified by the savant zoologist with perfect memory. When

presenting a giraffe to the savant, he would reject it and says it is not a giraffe, just because it does not have as many spots on its neck as the other giraffes he has seen. The other extreme end would be illustrated by the savant's carefree cousin, saying that if it has four legs, it is a giraffe. It goes without saying that neither of them are able to generalize well, most likely resulting in a few number of correctly classified giraffes.

The SVM will find a hyperplane that separates the two classes of training data with the largest margin possible, as can be seen in Figure 2.1. The hyperplane can be described by $\mathbf{w}^T\mathbf{x} + b = 0$ where $\mathbf{w}$ is normal to the hyperplane, $\frac{b}{||\mathbf{w}||}$ is the perpendicular distance from the hyperplane to the origin, b is a bias and $\mathbf{x}$ is the input pattern. An important concept in SVMs is the so called *margin*, which is the minimum distance from the hyperplane to the support vectors from both classes. Support vectors are the training examples that lie closest to the separating hyperplane from each class, i.e. the examples lying exactly on the margin boundary. Given new input data $\mathbf{x}^{(t)}$ and the learned parameter $\mathbf{w}$, the SVM predicts which of the two classes the data belongs to ($y_t \in \{\pm 1\}$) by checking which side of the separating hyperplane the input is located on by

$$y_t = \text{sgn}(\mathbf{w}^T\mathbf{x}^{(t)} + b) \tag{2.1}$$

where sgn is the signum function.

An SVM is usually trained with pairs $(\mathbf{x}^{(i)}, y_i)$, of a training pattern $\mathbf{x}^{(i)}$ and an associated "truth" y_i, by finding the parameter $\mathbf{w}$ by solving the SVM primal problem

$$\mathbf{w} = \text{argmin}\, f(\mathbf{w}) = \frac{\lambda ||\mathbf{w}||^2}{2} + \ell_{emp}(\mathbf{w}) \tag{2.2}$$

where $\ell_{emp}(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m} \ell((\mathbf{x}^{(i)}, y_i); \mathbf{w})$, m is the number of samples in the training set and $\ell(\cdot)$ is a loss function that penalizes misclassified samples. A good choice for ℓ might be the so called *hinge-loss*

$$\ell((\mathbf{x}^{(i)}, y_i); \mathbf{w}) = \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x}^{(i)} + b)) \tag{2.3}$$

which has been shown to have appealing properties for SVMs [40], such as good generalization capabilities. In the case where the input data is not linearly separable (e.g. due to noise), the SVM can be allowed to find a hyperplane that separates the data as much as possible, while keeping the number of misclassified samples down. λ is a regularization parameter that scales the regularization term $\frac{||\mathbf{w}||^2}{2}$ and thus controls how important we think it should be to find an hyperplane with large

margin, versus finding one that separates many training examples correctly. Some other literature instead use a misclassification parameter C that scales $\ell_{emp}(\cdot)$. We can relate these by $\lambda = \frac{1}{mC}$. [40]

Note that the parameter λ does not depend on the algorithm, but is *problem-dependent*. This means that λ has to be set differently depending on the dataset used. Finding a reasonable value is usually done by cross-validation [2]. It might be instructive to point out that in general, a small λ (large C) causes us to harshly penalize misclassifications. If we find that a small λ is necessary for good performance, it might be an indication of that the problem the dataset models is in itself difficult [40].

An excellent survey of recent developments in using SVMs for large-scale learning can be found in [40].

2.2.1 Duality

The primal SVM formulation (2.2) can be transformed into a dual, which can then be solved efficiently using a quadratic program (QP) solver [13]. This transformation is done using the Lagrangian duality [13], transforming the convex primal SVM formulation into the following concave dual formulation of the SVM optimization problem:

$$\begin{aligned} \boldsymbol{\alpha} = \operatorname{argmax} f(\boldsymbol{\alpha}) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{subject to } 0 \leq \alpha_i \leq \frac{1}{\lambda m} \end{aligned} \tag{2.4}$$

where $\boldsymbol{\alpha}$ is a dual variable to be optimized. The parameter $\mathbf{w}$ can then be recovered by

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i \mathbf{x}^{(i)} \tag{2.5}$$

[2]The term k-fold cross-validation refers to the practice of leaving out one part of the training set and then training the model on the remaining $k-1$ parts. The error rate is then measured by testing on the left-out part. When there is a lack of proper test data, this procedure can give a good estimate of the model's generalization ability by repeating the above procedure, alternating the left-out part, and then measuring the average error. [8]

which is due to the *representer theorem* [32]. The representer theorem states that the optimal hyperplane can be expressed as a linear combination of the training samples of the classifier in some high-dimensional space.

Slater's Condition The SVM dual is used to solve the SVM primal problem in the dual space, which is often preferred, as it consists of maximizing a concave function. This is efficiently done by a QP solver [13]. However, one has to mind the *duality gap*, which is the difference between the optimal primal and dual values [13]. If *strong duality* holds, then the duality gap is zero and solving the dual problem directly gives the solution of the primal problem. Slater's condition conditions a case where the duality gap is zero, i.e. when we have strong duality. If we have a convex primal problem on the form

$$\begin{aligned} \text{minimize} \quad & f_0(\mathbf{x}) \\ \text{subject to} \quad & f_i(\mathbf{x}) \leq 0,\ i = 1, \dots, m \\ & \mathbf{Ax} = \mathbf{b} \end{aligned} \tag{2.6}$$

where f_0 is the objective function to minimize, $f_i, \forall i$ are convex inequality constraints and $\mathbf{Ax} = \mathbf{b}$ are equality constraints that must hold for the optimal $\mathbf{x}$. Slater's condition is that there exists a strictly feasible point, i.e. a solution such that

$$\begin{aligned} & f_i(\mathbf{x}) \leq 0,\ i = 1, \dots, k \\ & f_i(\mathbf{x}) < 0,\ i = k+1, \dots, m \\ & \mathbf{Ax} = \mathbf{b} \end{aligned} \tag{2.7}$$

where the k first f_i are affine functions and the rest convex. The condition is then true if the affine constraints hold with inequality and the convex inequality constraints hold with *strict inequalities*. Slater's theorem then states that strong duality holds if Slater's condition holds. Note that this condition is sufficient but not necessary. It is possible for problems that do not satisfy Slater's condition (although rare) to still have zero duality gap [13]. As we can easily find a point satisfying Slater's condition for the primal problem (2.2), we have strong duality for the SVM problem.

KKT Conditions The *Karush-Kuhn-Tucker* (KKT) conditions indicate certain properties for the solution $\mathbf{x}$ that has been found through the dual. The KKT

conditions are the following [13]:

$$
\begin{aligned}
f_i(\mathbf{x}) \leq 0,\ i = 1, \ldots, m \\
h_i(\mathbf{x}) = 0,\ i = 1, \ldots, p \\
\lambda_i \geq 0,\ i = 1, \ldots, m \\
\lambda_i f_i(\mathbf{x}) = 0,\ i = 1, \ldots, m \\
\nabla f_0(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i f_i(\mathbf{x}) + \sum_{i=1}^{p} \nu_i h_i(\mathbf{x}) = 0
\end{aligned}
\tag{2.8}
$$

where f_0 is the objective function, f_i are convex constraints, h_i are affine constraints and λ_i, ν_i are Lagrangian multipliers.

If the primal problem is convex, then any $(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ that satisfy the KKT conditions are optimal. For this reason, many algorithms for solving such problems as above (include the SVM problem) focus on solving the KKT conditions [13]. Moreover, as many QP solvers will provides the parameters $(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu})$ as part of their solution, one can also use the KKT conditions as a certificate of that the $\mathbf{x}$ returned by a black-box solver actually is a valid solution.

2.2.2 Kernels

A kernel function can be used to enhance the SVM so that it can solve problems where the data is not linearly separable [45]. Kernel functions map two inputs to a scalar value, indicating how similar the inputs are. $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \langle \phi(\mathbf{x}^{(i)}), \phi(\mathbf{x}^{(j)}) \rangle = K_{ij}$ denotes a kernel, where $\phi : \mathcal{X} \mapsto \mathcal{H}$ is a map from a feature space $\mathcal{X}$ into a Hilbert space $\mathcal{H}$ [45]. As a generalization of the Euclidean space, the Hilbert space can be of possibly infinite dimensionality and is equipped with a dot product. See Figure 2.2 for a simple example of a kernel mapping data into a higher dimensional space, where the data becomes linearly separable.

Mercer Kernels Mercer kernels [43, 45] are kernels that satisfy the following conditions:

1. They are continuous.
2. They are symmetric, $K_{ij} = K_{ji}$.

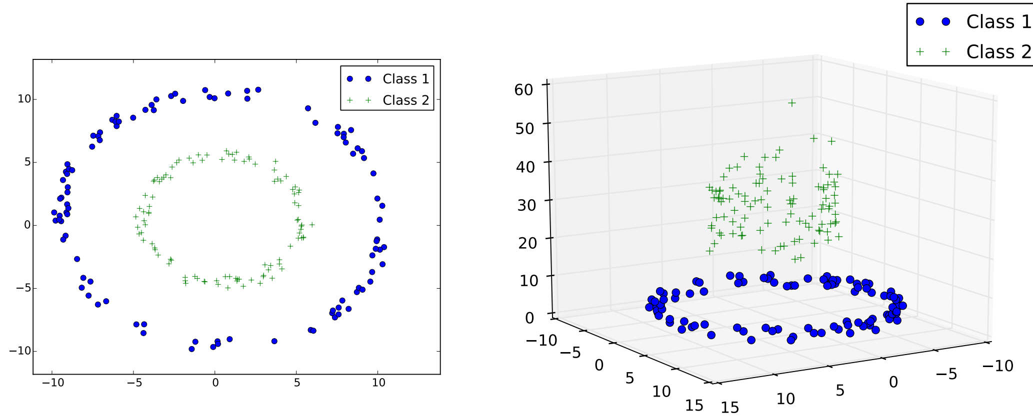


Figure 2.2: Simple example of how using a kernel with a non-linear map ϕ can make data linearly separable in a higher dimensional space. (Left) Linearly inseparable data in $\mathbb{R}^2$. (Right) Using a kernel to re-map the same data to $\mathbb{R}^3$, making it linearly separable.

3. They are positive semi-definite. That is, the $m \times m$ kernel matrix $\mathbf{K}$ must satisfy $\mathbf{v}^T\mathbf{K}\mathbf{v} \geq 0,\ \forall \mathbf{v} \in \mathbb{R}^m$. This essentially means that all eigenvalues of $\mathbf{K}$ are non-negative.

Common for all Mercer kernels is that they can be represented by an inner product in *some* Hilbert space $\mathcal{H}$.

The Kernel Trick The ability for kernels to be representable as an inner product in the higher dimensional space is an important advantage. Since the SVM dual (2.4) only contains the input vectors $\mathbf{x}$ expressed as dot products, the dual can be rewritten as [12, 25]

$$\begin{aligned}\boldsymbol{\alpha} = \operatorname{argmax} f(\boldsymbol{\alpha}) &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \\ &= \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{H} \boldsymbol{\alpha} \\ \text{subject to } 0 \leq \alpha_i &\leq \frac{1}{\lambda m}\end{aligned} \tag{2.9}$$

with $H_{ij} = y_i y_j K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. This is an important property, as computing the kernel mapping ϕ explicitly can sometimes be prohibitively expensive as the mapped vector $\phi(\mathbf{x})$ can be of very high (sometimes infinite!) dimensionality. As a means of reducing computational cost, the fact that the input vectors only occur as dot products in the dual is taken advantage of. By only considering dot products, we never have to explicitly compute ϕ but still get the correct answer. This kernel trick enables us to map the input data up into a very high dimensional space without having to worry about the existence of the mapping ϕ. For training the SVM, all the work we have to do before using a QP solver is creating the matrix $\mathbf{H}$. By satisfying the conditions for Mercer kernels, we know that the kernel is expressible as an inner product in *some* space $\mathcal{H}$, which is sufficient for using the kernel in the SVM classifier.

Examples of commonly used kernels include

- The radial basis function (RBF) kernel. $K_{ij} = \exp\left(-\frac{\|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2}{2\sigma^2}\right)$
- The polynomial kernel. $K_{ij} = (\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle + a)^b$
- The sigmoid kernel. $K_{ij} = \tanh(a\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle - b)$
- The linear kernel. $K_{ij} = \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$

where σ, a, b are parameter controlling the behavior of the kernel. These are usually set using cross-validation.

2.3 The Subgradient Method

The subgradient method [14] can be used for solving minimization problems where the objective function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is not differentiable. Basically, the method performs, similar to gradient descent, an update of the current best $\mathbf{x}$ as

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha_k \mathbf{g}^{(k)} \tag{2.10}$$

where $\mathbf{g}$ is *any* subgradient of $f(\mathbf{x}^{(k)})$. Note that in the gradient descent method, $\mathbf{g}$ is always the derivative of $f(\mathbf{x}^{(k)})$. A subgradient of $f(\mathbf{x}^{(k)})$ is any vector $\mathbf{g}$ that satisfies $f(\mathbf{y}) \geq f(\mathbf{x}) + \mathbf{g}^T(\mathbf{y} - \mathbf{x})$. That is, $\mathbf{g}$ is a global underestimator for f at the point $\mathbf{x}$. However, the subgradient method is *not* a descent method, which means that even though we subtract a multiple of $\mathbf{g}$ at each iteration, the value of the objective function may not actually decrease [14]. In fact, it often increases. It is therefore

common to keep track of a point with the smallest function value found so far. The parameter α_k is the step size at iteration k. Several different rules for how the step size changes can be used, but in this thesis we will only focus on step sizes that are square summable, but not summable. That is, the step sizes α_k have the following properties.

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty, \ \sum_{k=1}^{\infty} \alpha_k = \infty \tag{2.11}$$

One example of such a step size is one on the form $\alpha_k = \frac{1}{\lambda+k}, \ \lambda \geq 0$. For such a step size, it can be shown [14] that the subgradient method converges as $k \to \infty$.

While the formulation of the subgradient method is very simple, its drawback is that is converges slowly. Indeed, for any choice of step size it has a lower suboptimality bound of $\Omega(\frac{1}{\sqrt{T}})$, if the algorithm runs for T iterations [14].

2.4 Pegasos: Primal Estimated sub-GrAdient SOlver for SVM

Pegasos [47] is a fast state-of-the-art SVM solver [3] that uses the iterative subgradient method for solving the following optimization problem:

$$\mathbf{w} = \text{argmin}\, f(\mathbf{w}) = \frac{\lambda ||\mathbf{w}||^2}{2} + \frac{1}{m} \sum_{i=1}^{m} \ell((\mathbf{x}^{(i)}, y_i); \mathbf{w}) \tag{2.12}$$

where

$$\ell((\mathbf{x}^{(i)}, y_i); \mathbf{w}) = \max(0, 1 - y_i(\mathbf{w}^T \mathbf{x}^{(i)})) \tag{2.13}$$

The algorithm performs T iterations and then stops, so there is no guarantee that the Pegasos algorithm actually solves (2.12); it is an approximative randomized algorithm. The number of iterations required for obtaining a solution of accuracy ϵ is bounded by $\tilde{\mathcal{O}}(\epsilon^{-1})$, giving it a superior rate of convergence [47]. As Pegasos is a randomized algorithm, it in each iterative step selects a random subset $A_t \subseteq S$, where $|A_t| = k$ and S is the training dataset. The algorithm with $k = 1$ is stated

[3] An implementation of the Pegasos algorithm can be found at `http://www.cs.huji.ac.il/~shais/code/pegasos.tgz`

Algorithm 1 The Pegasos algorithm.

Input: S - training set.
Input: λ - regularization parameter.
Input: T - the number of iterations.
  Set $\mathbf{w}^{(1)} = 0$
  for $t = 1,2,\ldots,T$ **do**
    Choose $i_t \in \{0,\ldots,|S|\}$ uniformly at random.
    Set $\eta_t = \frac{1}{\lambda t}$
    if $y_{i_t}\langle \mathbf{w}^{(t)}, \mathbf{x}^{(i_t)}\rangle < 1$ **then**
      Set $\mathbf{w}^{(t+1)} = (1-\eta_t\lambda)\mathbf{w}^{(t)} + \eta_t y_{i_t}\mathbf{x}^{(i_t)}$
    else
      Set $\mathbf{w}^{(t+1)} = (1-\eta_t\lambda)\mathbf{w}^{(t)}$
    end if
  end for
  return $\mathbf{w}^{(T+1)}$
Output: $\mathbf{w}^{(T+1)}$ - learned parameters.

in Algorithm 1. The original paper includes a projection step, but as new analysis indicates that it does not affect results [48], we omit it. Since the algorithm works on subsets of size k, its time complexity is independent on the size of the training dataset, which makes Pegasos suitable for huge datasets. Given the updated analysis in the journal version of the paper [48], $k = 1$ seems to be an appropriate choice, which is the choice of k that we will use in this thesis.

Regarding large-scale learning, one particular interesting aspect of Pegasos, as stated in the surprising discovery of [49], is that the training time of Pegasos should in fact *decrease* as the size of the training set increases. This astonishing fact utilizes the properties of SVM solvers based on stochastic gradient descent and that such solvers only process a random example at a time, thus making them independent of the training data size. A summary of this discovery, along with a general overview of the current state-of-the-art, can be found in [40].

2.4.1 Kernelized Version

The standard version of the Pegasos algorithm does not work with kernels [48]. However, the algorithm can be modified to enable this usage. We take a look at Algorithm 2, the kernelized version of the Pegasos algorithm, as stated in [48]. Here,

Algorithm 2 Kernelized version of Pegasos.

Input: S - training set.
Input: λ - regularization parameter.
Input: T - the number of iterations.
  Set $\boldsymbol{\alpha}^{(1)} = 0$
  for $t = 1,2,\ldots,T$ **do**
    Choose $i_t \in \{0,\ldots,|S|\}$ uniformly at random.
    For all $j \neq i_t$, set $\alpha_j^{(t+1)} = \alpha_j^{(t)}$
    if $y_{i_t} \frac{1}{\lambda t} \sum_{j=1}^{m} \alpha_j^{(t)} y_j K(\mathbf{x}^{(i_t)}, \mathbf{x}^{(j)}) < 1$ **then**
      Set $\alpha_j^{(t+1)} = \alpha_j^{(t)} + 1$
    else
      Set $\alpha_j^{(t+1)} = \alpha_j^{(t)}$
    end if
  end for
  return $\boldsymbol{\alpha}^{(T+1)}$
Output: $\boldsymbol{\alpha}^{(T+1)}$ - learned parameters.

$\alpha_j^{(t+1)}$ counts how many times element $\mathbf{x}^{(j)} \in S$ has been selected and had non-zero loss for the last t iterations [48]. It is defined as:

$$\alpha_j^{(t+1)} = |\{t' \leq t : i_{t'} = j \wedge y_j \langle \mathbf{w}^{(t')}, \phi(\mathbf{x}^{(j)}) \rangle < 1\}| \tag{2.14}$$

where ϕ is a map into a Hilbert space $\mathcal{H}$. Further, we recover the parameter $\mathbf{w}$ using

$$\mathbf{w}^{(t+1)} = \frac{1}{\lambda t} \sum_{j=1}^{m} \alpha_j^{(t+1)} y_j \phi(\mathbf{x}^{(j)}) \tag{2.15}$$

It should be noted that, unlike the standard version, the kernelized version of Pegasos has a direct dependence on the number of training samples m.

2.5 Kronecker Product

The Kronecker product ($\otimes$) on a $p \times q$ matrix $\mathbf{A}$ and an $r \times s$ matrix $\mathbf{B}$ is defined to create a block matrix as follows.

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{11}\mathbf{B} & \cdots & A_{1q}\mathbf{B} \\ \vdots & \ddots & \vdots \\ A_{p1}\mathbf{B} & \cdots & A_{pq}\mathbf{B} \end{bmatrix} \tag{2.16}$$

Kronecker products have the following property ([41], pp. 60):

$$\begin{gathered} (\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD} \Rightarrow \\ (\mathbf{A} \otimes \mathbf{B})(\mathbf{A} \otimes \mathbf{B}) = \mathbf{AA} \otimes \mathbf{BB} \Leftrightarrow \\ (\mathbf{A} \otimes \mathbf{B})^2 = \mathbf{A}^2 \otimes \mathbf{B}^2 \end{gathered} \tag{2.17}$$

From this, a property for matrix powers of Kronecker products can be derived:

$$(\mathbf{A} \otimes \mathbf{B})^n = \mathbf{A}^n \otimes \mathbf{B}^n \tag{2.18}$$

See Appendix B.1 for the proof.

For the sum of the Kronecker product, we have the following:

$$\sum_{i,j} [\mathbf{A} \otimes \mathbf{B}]_{ij} = \mathbf{e}^T\mathbf{A}\mathbf{e}\mathbf{e}^T\mathbf{B}\mathbf{e} \tag{2.19}$$

See Appendix B.2 for the proof.

2.6 Direct Product Graph

The direct product graph $G_\times = (V_\times, E_\times)$ of two graphs $G^{(1)} = (V^{(1)},E^{(1)})$ and $G^{(2)} = (V^{(2)},E^{(2)})$ is defined so that an edge exists between two nodes in $G_\times$ iff there exists a similar edge in *both* $G^{(1)}$ and $G^{(2)}$, and the nodes on that edge are similarly labeled in both $G^{(1)}$ and $G^{(2)}$. More formally, it is defined as [10, 27]:

$$\begin{aligned} V_\times(G^{(1)} \times G^{(2)}) = & \quad \{(v_1,w_1) \in V^{(1)} \times V^{(2)} : \\ & \quad \mathrm{label}(v_1) = \mathrm{label}(w_1)\} \\ E_\times(G^{(1)} \times G^{(2)}) = & \quad \{((v_1,w_1),(v_2,w_2)) \in V_\times^2(G^{(1)} \times G^{(2)}) : \\ & \quad (v_1, v_2) \in E^{(1)} \wedge (w_1, w_2) \in E^{(2)} \\ & \quad \wedge\ \mathrm{label}(v_1, v_2) = \mathrm{label}(w_1, w_2)\} \end{aligned} \tag{2.20}$$

It should be noted that using product graphs might be quite costly, as they may contain $|V^{(1)}| \times |V^{(2)}|$ nodes.

2.7 Graph Kernels

Graph kernels are kernels defined on the input space of all graphs $\mathcal{G}$. A graph kernel $K : \mathcal{G} \times \mathcal{G} \mapsto \mathbb{R}$ takes two graphs as input and produce a scalar output value, indicating the similarity of the graphs. There are several existing graph kernels (see Chapter 3). In this thesis, however, we focus on the following three.

2.7.1 Direct Product Kernel

The direct product (DP) kernel compares two graphs based on the number of walks they have in common [27]. The common walks are computed by creating the *direct product graph* [27] $G_\times = (V_\times, E_\times)$ of two graphs $G^{(1)} = (V^{(1)}, E^{(1)})$ and $G^{(2)} = (V^{(2)}, E^{(2)})$, The kernel value is computed using

$$K_{DP}(G^{(1)}, G^{(2)}) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda^n A(G_\times)^n \right]_{ij} \tag{2.21}$$

where $\lambda < \Delta(G_\times)^{-1} \in \mathbb{R}^+$ is a decay factor for making the sum converge.

2.7.2 Shortest Path Kernel

The shortest path (SP) kernel compares graphs based on the similarity of their shortest paths [10]. All the shortest paths in a graph can be obtained in $\mathcal{O}(n^3)$ time using the Floyd-Warshall algorithm [26]. We let S_{ij}^{G} denote the shortest *distance* between nodes v_i and v_j in graph G. For unweighted graphs, the shortest distance between v_i and v_j is the number of steps on the shortest $v_i \to v_j$ path, and for weighted graphs it is the sum of all weights on the minimum weight $v_i \to v_j$ path. The shortest-path kernel is defined as:

$$K_{SP}(G^{(1)}, G^{(2)}) = \sum_{i,j,p,q} k_{walk}^{(1)}(S_{ij}^{G^{(1)}}, S_{pq}^{G^{(2)}}) \tag{2.22}$$

where $k_{walk}^{(1)}$ is a positive definite kernel on edge walks of length 1.

2.7.3 Graphlet Kernel

The graphlet (GL) kernel compares graphs through their distributions of *graphlets* of size $3-5$ [52], i.e. induced subgraphs of $k \in L = \{3,4,5\}$ nodes. As exhaustive enumeration of all graphlets is infeasible, the graphlet distribution is approximated using sampling, and by considering a finite number of values of k. Shervashidze et al. [52] show that for $\delta > 0$, $\epsilon > 0$,

$$n = \left\lceil \frac{2(\log 2 \cdot |L| + \log(\frac{1}{\delta}))}{\epsilon^2} \right\rceil \tag{2.23}$$

samples suffice to ensure that $P\{\|D-\hat{D}^n\|_1 \geq \epsilon\} \leq \delta$, where D is the true distribution and $\hat{D}^n$ is the approximate distribution using n samples. The distribution of graphlets for a graph $G^{(i)}$ is then denoted by $\mathbf{d}^{(G^{(i)})}$. The graphlet kernel is then defined as follows:

$$K_{GL}(G^{(1)}, G^{(2)}) = \mathbf{d}^{(G^{(1)})^T}\mathbf{d}^{(G^{(2)})} \tag{2.24}$$

2.8 Malin's Random Walk Method

Malin [39] has presented two methods that are applicable to our problem, one based on random walks and one based on hierarchical clustering. Since the random walk method was reported to give the best result, we use it as a state-of-the-art comparison for anonymized relational entity disambiguation. We denote this method MALIN. In this section, we will give a detailed description of the method, as understanding of MALIN is crucial for the outcome of our experimental results in Section 6.

MALIN is a random walk method that aims to group all references of an identifier of interest into clusters, where each cluster represents one underlying entity. Thus the resulting number of clusters is the predicted number of underlying entities. The method can be seen in Algorithm 3. A graph $G = (V,E)$ is first created, where references to same identifier have been merged into a single vertex. An edge is placed between two vertices if they co-occur in a source. For example, a source could refer to a news article, in which two persons are mentioned together.

Next, for each node of interest v_i, i.e. a node that we want to disambiguate, put all references to the identifier of v_i in a set $D^{(i)}$. Then create a graph $G^{(i)} = (V^{(i)},E^{(i)})$ that is identical to G, except that v_i has been removed and been replaced with

Algorithm 3 Malin's random walk method.

Input: $\mathcal{S} = \{S_1 = \{s_1, s_2, \dots\}, S_2, \dots\}$ - sources.
Input: T - the maximum number of steps per walk.
Input: N - the number of walks to make per node of interest.
Input: τ - similarity threshold.

  Let $v_i = s_i \Leftrightarrow \text{Ident}(v_i) = \text{Ident}(s_i)$
  Set $V = \{v_i : v_i = s_i \wedge s_i \in S_i \in \mathcal{S}\}$
  Set $E = \{(v_i, v_j) : (v_i, v_j) \in V \times V \wedge v_i = s_i \wedge v_j = s_j \wedge \exists S_k \in \mathcal{S}\ (s_i, s_j \in S_k)\}$
  Set $G = (V, E)$
  for $v_i \in V$ **do**
    Set $D^{(i)} = \{v_{S_k} : \text{Ident}(v_{S_k}) = \text{Ident}(S_k) \wedge \exists s_k\ (v_i = s_k \wedge s_k \in S_k \in \mathcal{S})\}$
    Set $V^{(i)} = \{v_j : v_j \neq v_i \wedge v_j \in V\} \cup D^{(i)}$
    Set $E^{(i)} = \{(v_p, v_q) : (v_p, v_q) \in V^{(i)} \times V^{(i)} \wedge v_p = s_p \wedge (v_q = s_q \wedge$
      $\exists S_k \in \mathcal{S}\ (s_p, s_q \in S_k)) \vee (\exists S_k \in \mathcal{S}\ (\text{Ident}(v_q) = \text{Ident}(S_k) \wedge s_p \in S_k)))\}$
    Set w_{pq} as in (2.25).
    Set $G^{(i)} = (V^{(i)}, E^{(i)})$
    for $v_{S_k} \in D^{(i)}$ **do**
      Do N random walks with step probability as in (2.26) until reaching a node
        $v_{S_l} \neq v_{S_k},\ v_{S_l} \in D^{(i)}$ or for maximum T steps.
      Calculate $P(v_{S_k} \Rightarrow v_{S_l}),\ \forall k, l$
      Set sim_{kl} as in (2.27).
    end for
    Set $V_d^{(i)} = \{v_{S_k} : v_{S_k} \in D^{(i)}\}$
    Set $E_d^{(i)} = \{(v_{S_k}, v_{S_l}) : (v_{S_k}, v_{S_l}) \in V_d^{(i)} \times V_d^{(i)} \wedge sim_{kl} \geq \tau\}$
    Set $G_d^{(i)} = (V_d^{(i)}, E_d^{(i)})$
    Let $\text{Comp}(v_{S_k})$ denote the set of vertices in the same component as $v_{S_k} \in V_d^{(i)}$
    Set $C^{(i)} = \{\text{Comp}(v_{S_k}) : v_{S_k} \in V_d^{(i)}\}$
  end for
  return $\{C^{(1)}, \dots\}$

Output: $\{C^{(1)}, \dots\}$ - clusters for each node of interest.

one vertex for each reference in $D^{(i)}$. Edge weights correspond to a normalized co-occurrence count. The weight between two vertices, p and q, is defined as

$$w_{pq} = \sum_{S_k \in \mathcal{S}} \frac{\theta_{pqk}}{|S_k|} \tag{2.25}$$

where $\mathcal{S}$ is the set of all sources, and θ_{pqk} is an indicator variable which is 1 if the identifiers p and q are both in source S_k and 0 otherwise.

In graph $G^{(i)}$, do random walks from each vertex in $D^{(i)}$ with the probability of walking from vertex p to the neighboring vertex q as

$$P(p \to q|p) = \frac{w_{pq}}{\sum_r w_{pr}} \tag{2.26}$$

A walk stops once it reaches a vertex in $D^{(i)}$, different from the one the walk started in, or when a maximum number of steps has been reached. When the walks are done, the posterior probability $P(v_{S_k} \Rightarrow v_{S_l})$ for a walk from v_{S_k} reaching v_{S_l} can be estimated for all vertices in $D^{(i)}$. Then, we define the similarity measure between two vertices as the average of their posterior probabilities:

$$sim_{kl} = \frac{P(v_{S_k} \Rightarrow v_{S_l}) + P(v_{S_l} \Rightarrow v_{S_k})}{2} \tag{2.27}$$

Finally, we create a graph $G_d^{(i)} = (V_d^{(i)}, E_d^{(i)})$ that contains only the vertices in $D^{(i)}$, with edges defined between nodes only if the similarity measure is higher than the specified threshold τ. If the similarity measure is higher than the threshold, then these vertices are deemed to refer to the same entity, and will belong to the same component in $G_d^{(i)}$. The references in the same component are then clustered together, and put in a set $C^{(i)}$. Each element of $C^{(i)}$ is then a clustering of references to a single underlying entity.

When using Malin for comparison, we predict any identifier vertex v_i with $|C^{(i)}| = 1$ as unambiguous ($-$), and ambiguous ($+$) otherwise.

Example Consider the following small example as a way of understanding Malin. In the example, let us have $\mathcal{S}$, the set of all sources, contain the sources S_1, S_2, S_3, with

$$\begin{aligned} S_1 &= \{s_1, s_3\} \\ S_2 &= \{s_1, s_2\} \\ S_3 &= \{s_1, s_3\} \end{aligned} \tag{2.28}$$

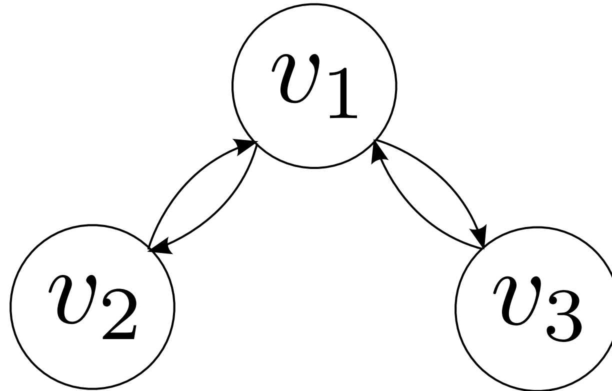


Figure 2.3: The original graph G in MALIN, where each vertex refers to one (possibly ambiguous) identifier. Vertices are connected through co-occurrence in sources.

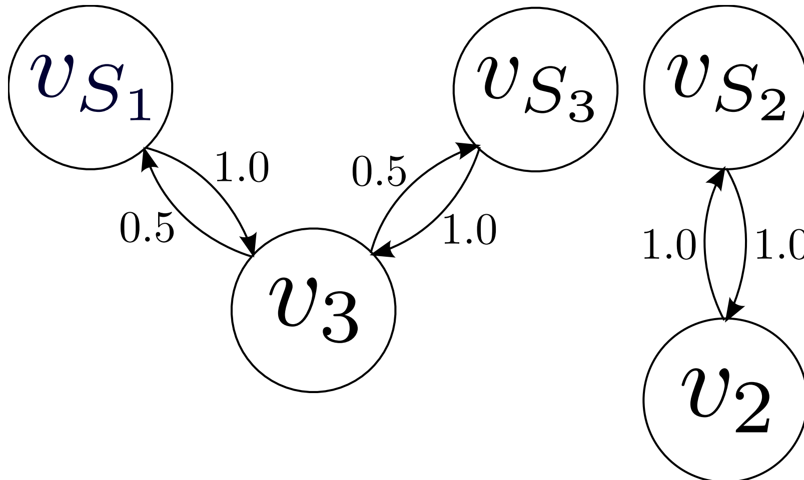


Figure 2.4: Graph with v_1 as the focused node of interest in MALIN. v_1 has thus been split into three different source vertices, v_{S_1},v_{S_2},v_{S_3}. Edge weights represent the walk probabilities that have been calculated using (2.26).

where s_i is an identifier to vertex v_i. Using the terminology of Algorithm 3, the original graph G would look like Figure 2.3. If we consider letting the vertex of interest be v_1, the first step is to split v_1 into one vertex for each of its sources. These new vertices get the identifiers v_{S_1}, v_{S_2} and v_{S_3}, named after the sources they occurred in. These are put in the set $D^{(1)} = \{v_{S_1}, v_{S_2}, v_{S_3}\}$. Let us call the vertices in this set *source vertices*. The walk probabilities for all vertices are then calculated and the final graph is shown in Figure 2.4, where the edge weights represents the

walk probabilities. The walk probability for any vertex is calculated using (2.26).

Once the graph is complete, a series of random walks are started from each vertex in $D^{(1)}$, either until reaching another vertex in $D^{(1)}$, or until the walk has reached a maximum number of steps. The algorithm keeps track of how many walks that started in a certain source vertex, that reached another source vertex. As can be seen in Figure 2.4, the walks starting from v_{S_1} have a high probability of reaching v_{S_3}. Likewise, walks starting in v_{S_3} have a high probability of reaching v_{S_1}. However, walks starting in v_{S_2} will never reach another vertex in $D^{(1)}$, since any walk will simply walk back and forth between v_{S_2} and v_2 until the maximum number of steps is reached.

For the sake of the example, consider the likely case that 100% of the walks that started in v_{S_1} reached v_{S_3}, no walk starting in v_{S_2} ever reached another source vertex and 100% of the walks starting in v_{S_3} reached v_{S_1}. Thus, the posterior probabilities become

$$\begin{aligned} P(v_{S_1} \Rightarrow v_{S_2}) &= 0.0 \\ P(v_{S_1} \Rightarrow v_{S_3}) &= 1.0 \\ P(v_{S_2} \Rightarrow v_{S_1}) &= 0.0 \\ P(v_{S_2} \Rightarrow v_{S_3}) &= 0.0 \\ P(v_{S_3} \Rightarrow v_{S_1}) &= 1.0 \\ P(v_{S_3} \Rightarrow v_{S_2}) &= 0.0 \end{aligned} \tag{2.29}$$

Given these results, the symmetric similarity measures then become

$$\begin{aligned} sim_{12} = sim_{21} &= 0.0 \\ sim_{13} = sim_{31} &= 1.0 \\ sim_{23} = sim_{32} &= 0.0 \end{aligned} \tag{2.30}$$

The final step of Malin is then to decide which of the vertices $v_{S_1}, v_{S_2}, v_{S_3}$ refer to the same entity. This is done using a simple threshold, where all vertices that have a similarity measure that is greater than the threshold are considered to be the same entity. In our example, any threshold lower than 1.0 would mean that the vertices v_{S_1} and v_{S_3} are predicted to refer to the same entity. No matter what threshold we choose, however, it is impossible to predict that v_{S_2} is referring to the same entity as any other source vertex. Choosing a threshold below 1.0 will then cause Malin to output two clusters, i.e. $|C^{(1)}| = 2$ and $C^{(1)} = \{\{v_{S_1}, v_{S_3}\}, \{v_{S_2}\}\}$. The threshold is usually chosen with cross-validation.

3

Related Research

THE PROBLEM OF ENTITY DISAMBIGUATION has been investigated in several related works, although not quite in the fashion that we formulate the problem in this thesis. Bhattacharya and Getoor solve the problem of entity resolution by deciding if two separate graphs actually represent the same entity. Their method uses string kernels and edge- and neighborhood similarity measures in graphs [5]. They also present an unsupervised approach in [6], based on latent Dirichlet allocation (LDA) and Gibbs sampling. Chen et al. consider a similar problem in context of database cleaning, often referred to as object consolidation [19]. The problem has also been referred to as reference disambiguation [30], record deduplication [7, 18] or record linkage [4, 29]. Kalashnikov et al. [30] look at reference disambiguation, which is the problem of correctly associating references to the correct entities. While their work seems similar to our, their problem is fundamentally different, as they are trying to find the entities (e.g. nodes in a graph) associated with a particular entity. In contrast, our problem in this thesis is that of deciding whether one identifier actually has more than one underlying entity.

Entity Resolution A large family of techniques are devoted to *entity resolution*, the process in which references are matched with their underlying entities. This problem has two difficulties, 1) the same identifier may be used for several entities (e.g. Chris Anderson (TED) and Chris Anderson (WIRED)), and 2) the same entity may be referred to using several identifiers (e.g. Chris Anderson, Mr. Anderson). Bhat-

tacharya and Getoor approach the problem using a probabilistic graphical model [6] and hierarchical clustering [5]. In both methods, similarity between identifiers is used, which makes them unusable in the anonymized setting. This notion can be generalized to include methods using any set of entity attributes in the resolution process [9, 16, 21, 22, 24, 42, 46]. In this thesis, entities are anonymized and have no such attributes. Further, we approach only the first difficulty stated above, rendering the above methods unsuitable for direct comparison.

Relational Entity Disambiguation A specialized problem, related to entity resolution, is *relational entity disambiguation*. Here, various kinds of network structure between entities are exploited. Bekkerman and McCallum [3] use link structure of personal web pages to disambiguate people in social networks. Bhattacharya and Getoor [5] use author lists of research papers, forming a network, and name similarity, to disambiguate authors. Both of these methods however, leverage information that is not available in our setting.

Malin [39] approaches the problem of disambiguating entities based on network structure alone. In his setting, entities are related through a set of sources. His canonical example is that of entities being actors and sources being movies. Two actors are deemed connected if they appear in the same movie. Malin makes two attempts to solve the problem, one using hierarchical clustering, and one based on random walks. Since Malin's random walk method performed the best in his experiments, and that to the best of our knowledge no other methods than those presented in Malin [39] are applicable to our problem, we use Malin's random walk method for a state-of-the-art comparison in our experiments.

Graph Classification This work attempts to classify nodes in a network as ambiguous or unambiguous. Building a classifier on nodes demands for a way of comparing them. While there are kernels for straight-forward node comparison in graphs, such as the diffusion kernel [33], these are defined on the full entity graph and consequently prohibitively computationally expensive for real-world graphs. Instead, we may compare the neighborhood structure of nodes, resulting in a graph classification (rather than node classification) problem.

Common graph kernels include shortest path kernels [10], direct product kernels [27] and graphlet kernels [52]. We evaluate the performance of all of them for the entity disambiguation task in our experiments. Moreover, we make extensions to the shortest-path and direct product kernels and evaluate them in terms of computation

speed and classification accuracy. While graph kernels have been used for e.g. protein structure prediction [11] or character recognition [2], to the best of our knowledge, no existing work uses this approach for the problem of relational entity disambiguation. Gärtner et al. [27] have also shown some hardness results, indicating that the computation of graph kernels that cover the full information in the graph is NP-hard, whereas graph kernels based on walks can be computed in polynomial time.

4

Our Approach

In this chapter we develop a novel formulation of relational entity disambiguation as a graph classification problem, suitable for anonymized data with a network structure. We also present our method for training our SVM for doing the classification.

We let the term *entity* refer to a person or a company etc. while an *identifier* is a name or a label. If several entities have the same identifier, we say that the identifier is *ambiguous*. While a single entity may have several identifiers, we do not address this here; we focus only on ambiguities. In our setting, entities have hidden relationships which are observed through co-occurring identifiers such as names mentioned in news articles. We let an *identifier graph* be the graph with one node for each identifier and an edge between every pair of co-occurring identifiers. Edges are weighted by the significance of the relationships, such as number of co-occurrences.

This setting leads us to the definition of anonymized relational entity disambiguation as the following classification problem.

Problem Definition 1. *Given an undirected identifier graph $G = (V,E)$ with edge weights $w_{ij} \in \mathbb{R}^+$ and training data $S = \{(v_i, y_i) : 1 \leq i \leq m, v_i \in V, y_i \in \{\pm 1\}\}$ that labels certain nodes as ambiguous $(+)$ or unambiguous $(-)$, classify new nodes as $+1$ or -1. Each node of G may refer to a single entity or several underlying entities. The weight of an edge signifies the importance of the connection between two nodes.*

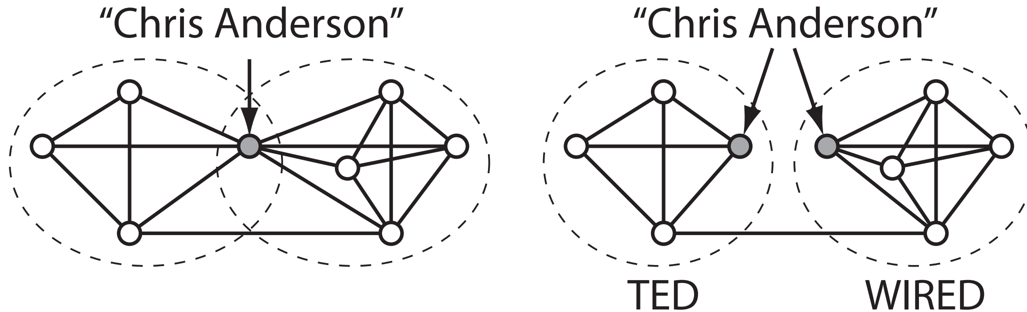


Figure 4.1: (Left) Local neighborhood (fictive) of the ambiguous vertex Chris Anderson. (Right) Correct splitting of the vertex into its two true underlying entities.

We now explain, in detail, an example of the problem we are solving. Our problem is illustrated by the example of Chris Anderson, stated in the introduction and further explained by Figure 4.1, re-shown in this section for reader convenience. In the figure, *two* individuals called Chris Anderson have been assigned only *one*, common identifier and thus *one* common node in the graph. This error creates a strong connection between the two communities, TED and WIRED, something we would not expect in reality. It is exactly this type of unexpected property of the network that we aim to capture with our classifier. Although this example involves only people, we would like to emphasize that nodes can represent any type of entity; an equally troublesome example would be that of the two *cities* Paris, France and Paris, Texas.

4.1 Method Overview

We approach the problem of identifying ambiguous nodes with the intuition that the neighborhoods of ambiguous nodes have structure different from those of unambiguous nodes. While Figure 4.1 is only an example, it illustrates the concept of two communities having fused around an ambiguous node. This motivates us to build a classifier using features of the neighborhood structure. In a graph $G = (V,E)$, we define the neighborhood $\mathcal{N}_G^{(i)}$ of a node $v_i \in V$ to be the subgraph of G induced by set of nodes connected to v_i through an edge. This notion can easily be extended to larger neighborhoods by considering neighbors of neighbors. In general, we define the κ-neighborhood $\mathcal{N}_{G,\kappa}^{(i)}$ of $v_i \in V$ as the subgraph induced by the set of nodes $\{v_j\} \subseteq V$ for which there exists a path from v_i to v_j with $s(v_i,v_j) \leq \kappa$ edges. Leaving

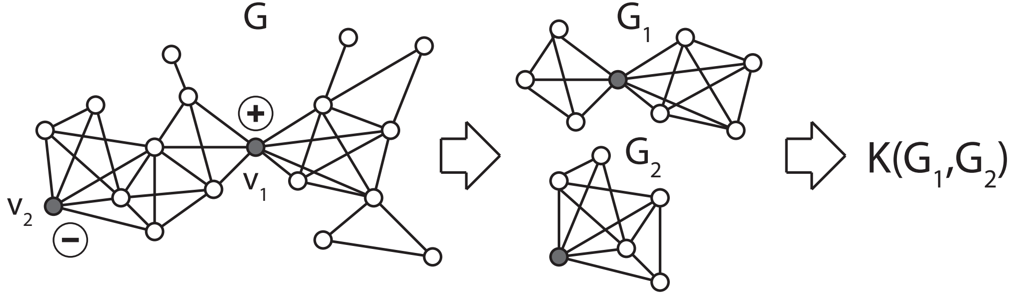


Figure 4.2: Method overview. In a three-step process, our approach considers a partially labeled co-occurrence graph (left), extracts local neighborhoods of nodes of interest (middle) and computes the kernel values of these nodes (right). The kernel values are then used for classification. In the figure, only labeled nodes are selected, as would be the case when training the classifier.

out the subscript G for convenience, we have,

$$\begin{aligned} V_\kappa^{(i)} &= \{v_i\} \cup \{v_j \in V : s(v_i, v_j) \leq \kappa\} \\ E_\kappa^{(i)} &= \{(v_p, v_q) : (v_p, v_q) \in E \wedge v_p, v_q \in V_\kappa^{(i)}\} \\ \mathcal{N}_\kappa^{(i)} &= (V_\kappa^{(i)}, E_\kappa^{(i)}) \end{aligned} \tag{4.1}$$

With $\kappa = 1$ we recover the common neighborhood consisting of the distinguished node and its immediate neighbors.

Our method consists of three steps, illustrated in Figure 4.2. First, an identifier graph $G = (V,E)$ is constructed from the raw data. Each node represents one identifier that may correspond to one or several underlying entities. Some of the nodes are labeled $+$ (ambiguous) or $-$ (unambiguous) respectively. An edge between two nodes is present if the two corresponding identifiers are related in some way, and the edge weight represents the strength of the relation. For example, two identifiers may be related by co-occurring in the same article. The number of such co-occurrences is the weight of the edge. Note that we do not assume to have access to information about in *which* articles two identifiers co-occurred, only that they co-occurred in *some* articles. In the second stage, we extract the local neighborhoods of the nodes that we want to classify. For these local neighborhoods we then compute one of several graph kernels K in order to measure similarity between nodes. New nodes are labeled using a standard classifier. The overall approach is summarized in Algorithm 4.

Algorithm 4 Detect ambiguous nodes.

Input: $G = (V,E)$
Input: κ - Neighborhood size.
for $v_i \in V$ **do**
  Set $G^{(i)} = \mathcal{N}_{\kappa}^{(i)}$ according to (4.1)
end for
Compute graph kernel matrix $\mathbf{K} = [K_{ij}]$, $\forall G^{(i)}, G^{(j)}$
Train an SVM with K and labels $\{y_i : y_i \in \{\pm 1\}\}$.
Output: SVM classifier.

4.2 Kernel Extensions

In this section, we design simple extensions of the DP and SP kernels, introduced in Sections 2.7.1 and 2.7.2, which we show can be computed quickly by explicitly knowing the kernel mapping ϕ.

4.2.1 Truncated Direct Product Kernel

As the product graph $G_\times$ of two graphs $G^{(1)} = (V^{(1)}, E^{(1)})$ and $G^{(2)} = (V^{(2)}, E^{(2)})$ might contain up to $|V^{(1)}| \times |V^{(2)}|$ vertices, taking powers $A(G_\times)^n$ might be very costly, as even for a sparse graph, taking powers might make the adjacency matrix full [10]. In order to speed up computation, we utilize the mathematical properties of the Kronecker product in order to avoid actually computing the product graph, but still get the correct result. For unlabeled graphs, the DP kernel (2.21) can be decomposed into components that can be calculated independently for each graph. For our proof, we need first to note that [53]

$$A(G_\times) = A(G^{(1)}) \otimes A(G^{(2)}) \tag{4.2}$$

Kronecker product powers have the property that [15](p.775)

$$(\mathbf{A} \otimes \mathbf{B})^n = \mathbf{A}^n \otimes \mathbf{B}^n \tag{4.3}$$

From the definition of the Kronecker product we get

$$\sum_{i,j} [\mathbf{A} \otimes \mathbf{B}]_{ij} = \mathbf{e}^T \mathbf{A} \mathbf{e} \mathbf{e}^T \mathbf{B} \mathbf{e} \tag{4.4}$$

Algorithm 5 Calculate the number of random walks.

Input: $G = (V,E)$
Input: K - Maximum walk length.
Initialize $t_i^{(0)} = 1,\ \forall i : v_i \in V;\ u_0 = |V|$
for $n \in \{1, \ldots, K\}$ **do**
  Set $t_i^{(n)} = \sum_{j:(v_j,v_i)\in E} t_j^{(n-1)},\ \forall i : v_i \in V$
  Set $u_n = \sum_{i:v_i\in V} t_i^{(n)},\ \forall i : v_i \in V$
end for
Output: $\mathbf{u}$ - Random walk counts.

Now, we can derive a useful representation of the DP kernel as follows:

$$\begin{aligned} K_{DP}(G^{(1)}, G^{(2)}) &= \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda^n A(G_\times)^n \right]_{ij} \\ &= \sum_{n=0}^{\infty} (\lambda^{\frac{n}{2}} u_n^{(1)})(\lambda^{\frac{n}{2}} u_n^{(2)}) \end{aligned} \tag{4.5}$$

where $u_n^{(i)} = \mathbf{e}^T A(G^{(i)})^n \mathbf{e}$. For the full derivation, please see Appendix B.4. The last expression of (4.5) can be written as an inner product $\langle \phi(G^{(1)}), \phi(G^{(2)}) \rangle$, making it a valid kernel [45]. We approximate (4.5) by only considering walks up to a finite length K, setting the kernel value of all longer paths to zero. This makes the dimension of the feature vectors ϕ finite, and trivially, the kernel is still valid. We call this the *truncated direct product* (TDP) kernel. For clarity, the kernel is defined as:

$$K_{TDP}(G^{(1)}, G^{(2)}) = \sum_{n=0}^{K} (\lambda^{\frac{n}{2}} u_n^{(1)})(\lambda^{\frac{n}{2}} u_n^{(2)}) \tag{4.6}$$

The vector $\mathbf{u}^{(i)} = [u_0^{(i)}, \ldots, u_K^{(i)}]^T$ for graph $G^{(i)}$ can be calculated using Algorithm 5. Note that making the sum in (4.5) finite allows us to set $\lambda > \Delta(G_\times)^{-1}$, making the TDP kernel defined where the DP kernel is not. As the average degree of a graph is $2\frac{|E|}{|V|}$, the time complexity for calculating the TDP kernel becomes $\mathcal{O}(K|E|)$.

For general graphs, exact approaches take $\mathcal{O}(|V|^6)$ [27] and $\mathcal{O}(|V|^3)$ [53] time to calculate the DP kernel for exact solutions. If the graph is sparse, i.e. has $\mathcal{O}(|V|)$ edges, then the calculation can be done in $\mathcal{O}(|V|^2)$ [53] time. For an approximate solution

in the general case, the kernel can be calculated in $\mathcal{O}(|V|^2)$ [31] time. Therefore, our approach is advantageous if $K \ll |V|$ and the graph is sparse.

Long walks (large K) tend to result in a phenomenon known as *tottering* in which the walks of the DP kernel will go back and forth along the same nodes, over and over. Tottering reduces the expressivity of the kernel as the same cycles of nodes will be visited repeatedly [38]. This suggests that for good generalization performance, as suggested empirically by Borgwardt and Kriegel [10], K should be small.

Distinguished Vertex The TDP kernel can be modified further to suit to our specific classification problem. We note that the graphs we are comparing are in fact pointed graphs, in that they are the neighborhood of one distinguished vertex. Instead of counting the number of random walks from all vertices, we can choose to only count walks from the distinguished vertex in the middle of the local neighborhood graph (i.e. the vertex "Chris Anderson" in Figure 4.1). This enables us to collect more specific information concerning only the local neighborhood of the vertex of interest. Thus, we create a distinguished vertex modification of the TDP kernel as follows:

$$K_{TDP_d}(G^{(1)}, G^{(2)}) = \sum_{n=0}^{K} (\lambda^{\frac{n}{2}} u_n^{(1)})(\lambda^{\frac{n}{2}} u_n^{(2)}) \tag{4.7}$$

where $u_n^{(i)} = \mathbf{e}_d^T A(G^{(i)})^n \mathbf{e}$ if the distinguished vertex is that of index d. Essentially, only the definition of the vector $\mathbf{u}$ has changed. Algorithm 5 can be easily modified to account for walks only from the distinguished vertex by setting $t_i^{(0)} = 1$ only for the distinguished vertex in the initialization step of the algorithm, and 0 for all other vertices.

We make a note that this definition is a special case of the alternative definition of the DP kernel in Vishwanathan et al. [53]. They incorporate starting and stopping probabilities $p_\times$ and $q_\times$ for the random walks, giving the following graph kernel:

$$K(G^{(1)}, G^{(2)}) = \sum_{n=0}^{\infty} \mu(n) q_\times^T \mathbf{W}_\times^n p_\times \tag{4.8}$$

where μ is a discrete measure and $\mathbf{W}_\times$ is the weight matrix associated with $A(G_\times)$. By ignoring edge weights and setting $\mu(n) = \lambda^n$, $p_\times = \mathbf{e}$ and $q_\times = \mathbf{e}_d$, we recover (4.7) with $K = \infty$.

4.2.2 Binned Shortest Distance Kernel

We seek to construct a computationally efficient version of the SP kernel for weighted graphs. Consider using the indicator function for $k_{walk}^{(1)}$ in the shortest-path kernel as in Borgwardt and Kriegel [10]. Inserting this into (2.22) leaves us with,

$$K_{SPI}(G^{(1)}, G^{(2)}) = \sum_{i,j,p,q} \mathbb{1}\left[S_{ij}^{G^{(1)}} = S_{pq}^{G^{(2)}}\right] \tag{4.9}$$

For unweighted or integer weighted graphs, S_{ij} are integers, making the indicator function a reasonable choice of kernel. However, for real weighted graphs, this formulation is less sensible as it involves comparing real numbers for equality. Therefore, we design a simple heuristic extension of (4.9) for general weighted graphs with the idea of comparing rounded-off values of the real-valued weights. Formally, we define a function $h : \mathbb{R}^+ \mapsto \{1, 2, \ldots, M\}$ that maps distance values to a finite set of M bins. This gives us the definition of the *binned shortest distance* (BSD) kernel as:

$$K_{BSD}(G^{(1)}, G^{(2)}) = \sum_{k=1}^{M} \sum_{i,j} \theta_{ijk}^{G^{(1)}} \sum_{p,q} \theta_{pqk}^{G^{(2)}} \tag{4.10}$$

where $\theta_{ijk}^{G} = \mathbb{1}[h(S_{ij}^{G}) = k]$. For the full derivation, please see Appendix B.3. This definition can be thought of as a relaxation of (4.9) in which the indicator has value 1 if the compared values are similar enough. Equation (4.10) can easily be written as the inner product $\langle \phi(G^{(1)}, \phi(G^{(2)}) \rangle$, showing that it is a valid kernel [45].

Binning By applying different types of binning (choosing the function h), the kernel can be made to consider similar distances to be equal, as opposed to the indicator version of the SP kernel (4.9). We believe this is advantageous in weighted graphs, as we will show empirically in our experiments. A simple binning function is the linear binning defined by

$$h(S_{ij}) = \left\lfloor \frac{M S_{ij}}{S_{max} + \epsilon} \right\rfloor + 1 \tag{4.11}$$

where $S_{max} = \max_{G,i,j} S_{ij}^{G}$ is the maximum shortest distance S_{ij}^{G} encountered in the dataset and $\epsilon \in \mathbb{R}^+$ is a small constant.

4.2.3 Normalization

We can make the TDP and BSD kernels less sensitive to the graph size by normalizing the feature vectors, giving the kernels the form,

$$K_{ij} = \frac{\phi(G^{(i)})^T \phi(G^{(j)})}{\| \phi(G^{(i)})\| \| \phi(G^{(j)})\|} \tag{4.12}$$

This definition of the TDP and BSD kernels is equivalent to the cosine similarity measure [44]. We show in our experiments that normalization indeed helps increase classification performance.

4.3 Enabling Fast Computation

In this section we present a few tricks for speeding up our method.

4.3.1 Explicit Knowledge of ϕ

Since the kernel mappings ϕ are explicitly known for the TDP and BSD kernels, we avoid expensive computation of the kernel values K_{ij} for each *pair* of graphs. In fact, we do not need to compute the Gram matrix $\mathbf{K}$ at all, since explicitly knowing ϕ allows us to use a fast linear SVM solver, making our graph kernel-based approach usable in $\mathcal{O}(mT)$ time. This procedure is shown in Algorithm 6. We use Pegasos [48], a state-of-the-art iterative subgradient method for training the SVM classifier. Pegasos has the property of being independent of the training set size m when using linear kernels, which is to our advantage, as previously described. Note that this independence only holds when ϕ is known, as the Pegasos algorithm becomes directly dependent on m when using kernels in the general case [48].

We make the computation of ϕ scalable by using GraphLab [36], a library for doing distributed computation. By following the restrictions of the GraphLab framework, scalable computation on graphs can be easily achieved. Since we explicitly know the mapping ϕ, we can easily design algorithms that fit the GraphLab framework for computing ϕ for each graph in an efficient manner.

Algorithm 6 Detect ambiguous nodes. Fast version.

Input: $G = (V,E)$
Input: κ - Neighborhood size.
for $v_i \in V$ **do**
  Set $G^{(i)} = \mathcal{N}_{\kappa}^{(i)}$ according to (4.1)
end for
Compute feature mappings $\phi(G^{(i)})$, $\forall G^{(i)}$
Train a linear SVM with labels $\{y_i : y_i \in \{\pm 1\}\}$.
Output: SVM classifier.

4.3.2 Batch Computation

We here mention a trick to speed up the parallel computation the graph kernels for all samples in the training set when using GraphLab. Suppose that we want to compute the graph kernels for m graphs. A naïve way would be to, in a for loop, compute each of the m graph kernels. In case the graphs are of relatively small size, the actual iterative step in each graph kernel algorithm mentioned above might take less time than the GraphLab communication overhead. In order to circumvent this, we create a supergraph $\mathcal{S}$, containing each input graph $G \in \mathcal{G}$, $|\mathcal{G}| = m$ as subgraphs. We say that each graph G is a subgraph in $\mathcal{S}$. We then run one of the above graph kernel algorithms just one single time on $\mathcal{S}$. When computing the vectors $\phi(G^{(i)})$, we instead create one per subgraph, efficiently computing the graph kernels for the whole training set in one go. We have seen empirically that this method is faster than the naive one, which seems reasonable, as GraphLab will be able to schedule idle machines to work on several $G \in \mathcal{G}$ at once, something that would not have been possible using a serial implementation. Results of this batch computation mode can be seen in Section 6.4. Note that with this trick, we parallelize both over the training set *and* over each individual graph.

4.4 Graph Kernel Implementations

In this section we show how we have implemented the calculation of the mapping ϕ for the TDP and BSD kernels in GraphLab. As we know the mapping explicitly for both kernels, this is all we need to do in order to efficiently calculate the kernel values.

4.4.1 Truncated Direct Product Kernel

Following the ideas in (4.6), the mapping ϕ of the TDP kernel is implemented in GraphLab using the GAS layout as follows.

Gather phase
Gather on in-edges, summing together the data of each adjacent vertex.

Apply phase
Set the data of the vertex to the above calculated sum.

Scatter phase
Not used.

The vertex program works by the idea that the initial number of random walks of length 0 to a node is 1, as a zero-length walk can only go to the node itself. Then, the number of walks of length 1 can be found by summing the number of random walks of length 0 of all adjacent vertices connected by in-edges to this vertex. More formally, the random walk algorithm can be expressed as

$$\begin{aligned} t_i^{(k+1)} &= \sum_{v_j \in V, \text{ s.t. } (v_j, v_i) \in E} t_j^{(k)} \\ t_i^{(0)} &= 1 \end{aligned} \tag{4.13}$$

where $t_i^{(k)}$ is number of random walks of length k to vertex i. This recurrence is exactly what taking the matrix power of the adjacency matrix does; a standard way of calculating the number of random walks of certain lengths from the adjacency matrix of a graph [28, 53]. This vertex program is then wrapped around an external loop that runs the vertex program of each vertex one step, whereafter it collects the number of random walks. The algorithm will trivially converge after $\log_\lambda \epsilon = K$ steps. The correctness of the algorithm follows directly from that (4.13) is equivalent to calculating random walks using powers of the graph adjacency matrix. Pseudocode can be seen in Algorithms 7, 8 and 9.

The time complexity becomes $\mathcal{O}(\frac{1}{k}K|V|\Delta_{in})$, where Δ_{in} is the maximum in-degree of the vertices and k the number of machines. This can be simplified to $\mathcal{O}(\frac{1}{k}K|E|)$, as the average degree of a graph is $\mathcal{O}(\frac{|E|}{|V|})$. The memory complexity becomes $\mathcal{O}(K+|V|)$, as we need to store only one value per iteration, and a value for each vertex.

Algorithm 7 TDP kernel ϕ-calculation in GraphLab.

Input: $G = (V,E)$ - graph to compute ϕ for.
Input: λ - decay parameter.
Input: $\epsilon = \frac{1}{100000}$ - stopping parameter.
  Set v.data $= 1,\ \forall v \in V$;
  Set $n = \log_\lambda \epsilon$;
  for $i \in \{0, \ldots, n\}$ **do**
    Set $u_i = \sqrt{\lambda^i} \sum_{v \in V} v$.data;
    Signal all vertices using the GraphLab engine;
    Start the GraphLab engine with Algorithms 8 and 9 on G, run one step;
  end for
  return u;
Output: **u** - the vector $\phi(G)$.

Algorithm 8 TDP vertex program gather phase.

Input: v - This vertex.
Input: $e = (v_{adj},v)$ - In-edge. ▷ Gather only on in-edges.
  return v_{adj}.data;
Output: Data of v_{adj}: a scalar.

4.4.2 Binned Shortest Distance Kernel

Before we begin, we will in the next paragraph briefly describe a data type called a *Shortest Path Map*.

Shortest Path Map A Shortest Path Map is similar to a normal map data structure, with the extra functionality that it implements the += operator. This operator, when applied on two maps, will add the keys and values of the second map to the first one. When conflicting keys are present, the value is set to the minimum of the

Algorithm 9 TDP vertex program apply phase.

Input: v - This vertex.
Input: *total* - The summed-together value from the gather phase.
  Set v.data $= total$;
Output: Nothing.

two. Then, for each pair (*key*,*val*) in the map, *key* represents a vertex id, from which we know that of all paths we have tried *so far*, there was a locally shortest path of length *val*.

Now, we describe the implementation of the ϕ-calculation for the BSD kernel. Using the idea in (4.10), it was implemented in GraphLab using the GAS layout as follows.

Gather phase

Add the length of the edge to the adjacent vertex's Shortest Path Map and return a copy of it. Due to the definition of the `+=` operator for Shortest Path Maps, the resulting summed-together Shortest Path Map presented in the apply phase will only contain the shortest paths from each key (vertex id) to this vertex. In this sense, when the vertex receives a new Shortest Path Map, we know that the map contains the shortest path to this vertex found *so far* by asking the neighboring vertices. However, we might already know a shorter path that was stored previously in this vertices's Shortest Path Map.

Apply phase

Only update the shortest path in the vertex's own map if the newly found map actually contained a shorter path. If we found one, we indicate that this vertex was updated and that it thus has gotten some new information.

Scatter phase

Signal adjacent vertices only if we actually learned something new during the apply phase. The idea is that a vertex will spread the latest gossip about a new shortest path, but it will not talk to its neighbors about old news.

Following this thought, the algorithm will converge when no new shortest path has been found for any vertex, indicating that the final Shortest Path Map in each vertex then actually is optimal. Pseudocode can be seen in Algorithms 10, 11, 12 and 13.

The time complexity then becomes $\mathcal{O}(\frac{1}{k}|V|^2(\Delta_{in} + |V|))$, as each Shortest Path Map has to propagate to each vertex from each vertex, where each vertex program needs to run the gather and scatter phase (each taking $\mathcal{O}(\Delta_{in})$ time) and the apply phase (taking $\mathcal{O}(|V|)$ time). Δ_{in} denotes the maximum in-degree of the vertices. Since we parallelize over vertices, using k machines, we can express the average work done per machine by diving by k. This can be reduced to simply to $\mathcal{O}(\frac{1}{k}|V|^3)$. Since each vertex holds a Shortest Path Map that at the termination of the algorithm will hold $|V|$ entries, we get a memory complexity of $\mathcal{O}(\frac{1}{k}|V|^2)$, as the distributed graph representation of GraphLab will divide the vertices upon the k machines.

Algorithm 10 BSD kernel ϕ-calculation in GraphLab.

Input: $G = (V,E)$ - graph to compute ϕ for.
Input: M - the number of histogram bins.
Signal all vertices using the GraphLab engine;
Start the GraphLab engine with Algorithms 11, 12 and 13 on G, run until convergence;
Compute a histogram with M bins in $\mathbf{u}$ of shortest paths, $\forall v \in V$;
return $\mathbf{u}$;
Output: $\mathbf{u}$ - the vector $\phi(G)$.

Algorithm 11 BSD vertex program gather phase.

Input: v - This vertex.
Input: $e = (v_{adj},v)$ - In-edge. ▷ Gather only on in-edges.
return A copy of v_{adj}'s Shortest Path Map with the length of edge e added;
Output: Shortest Path Map with the length of edge e added.

Algorithm 12 BSD vertex program apply phase.

Input: v - This vertex.
Input: spm_{new} - The summed-together Shortest Path Map from the gather phase.
Let spm_v be the Shortest Path Map contained in the vertex program of vertex v.
if spm_v does not contain an entry for the trivial path from v to v **then**
  Add the pair $(id_v, 0)$ to spm_v;
end if
for each pair $(id_{from}, dist_{new}) \in spm_{new}$ **do**
  if $(id_{from},dist_{old}) \notin spm_v$ **then**
    Add the pair $(id_{from}, dist_{new})$ to spm_v;
    Indicate that v has changed during this phase;
  else if $(id_{from},dist_{old}) \in spm_v \wedge dist_{new} < dist_{old}$ **then**
    Replace $dist_{old}$ with $dist_{new}$ for key id_{from} in spm_v;
    Indicate that v has changed during this phase;
  end if
end for
Output: Nothing.

Algorithm 13 BSD vertex program scatter phase.

Input: v - This vertex.
Input: $e = (v, v_{adj})$ - Out-edge. ▷ Scatter only on out-edges.
  if v changed during the apply phase **then**
    Signal v_{adj};
  end if
Output: Nothing.

5

Applications

The graph kernels described in this thesis can be applied to Recorded Future's data for entity disambiguation, a real problem which is looked at by the company. The method presented in this thesis can of course also be applied to datasets of similar structure describing a different scenario, such as the IMDb dataset. Since our method requires very minimal information, it is applicable to many different types of datasets.

One particularly nice feature of our graph kernel-based approach, is that it works in a fully anonymized setting, in which we know nothing about the names of any of the related persons, places etc. This means that our approach can be used in situations where information such as the names of people are sensitive and cannot be released. This makes our approach applicable in many settings where previous methods cannot be applied.

Since we have parallelized the computation of the graph kernels in GraphLab, our method will be able to scale on large datasets where other methods might take a long time to compute. In fact, as is shown in Section 6, some of our graph kernels are faster than the state-of-the-art by a significant factor.

6

Experiments

We evaluate our approach by comparing it, using a large selection of graph kernels, to the random walk method of Malin [39], denoted Malin. We address two questions, 1) how well have our extensions improved existing graph kernels, 2) how well does our method fair against a state-of-the-art method. The experiments are conducted on two real-world datasets, one of which is readily available.

6.1 Recorded Future News Data

We investigate a proprietary dataset (RF) from Recorded Future [1], a company specializing in web intelligence and predictive analytics. The data has been gathered using automatic processing of articles from news and social media. When an entity, e.g. a person, place or company, was found in an article, it was assigned a (possibly ambiguous) canonical identifier.

The RF dataset contains information from around 10 million articles. The identifier graph of the set has a node for each identifier in the set of articles and an edge (v_i,v_j) if identifiers v_i and v_j have co-occurred in an article. Each edge is associated with a weight w_{ij} equal to the number of times i and j have co-occurred. The graph contains 2,155,893 nodes and 13,935,815 edges and has 91 nodes that have been

[1] `http://www.recordedfuture.com/`

manually labeled as either ambiguous (+1) or unambiguous (−1). The average size for each local neighborhood graph of the labeled nodes was 267 nodes and 5,830 edges. 39.6% of the labeled nodes have a positive label. Note that in this dataset we do not have access to the actual articles that is the cause of the relations, only the fact that certain names co-occurred in a number of articles, making Malin's random walk method inapplicable to this dataset.

6.2 Internet Movie Database

The Internet Movie Database [2] (IMDb) is an online database gathering information about the televised entertainment industry, including movies, actors and production personnel.

Following Malin [39], we artificially introduced ambiguities by treating all actors with the same last name as one, merging them into a single node corresponding to a single identifier e.g. "Smith". An edge was added between two nodes if the corresponding identifiers were part of the same cast list in a movie. Edges were weighted by the total number of times a pair of nodes starred in movies together. Additionally, we only included movies with more than one actor, thus ensuring that every node is adjacent to at least one other node. Actors who had not starred together with any other actor were removed from the dataset. Just like in Malin [39], we only considered movies between 1994-2003. It is important to note that our dataset is not the exact same as the one used in Malin [39]. This is due to the fact that the IMDb database has been updated since. The dataset used in by Malin [39] had ∼37,000 movies and ∼180,000 distinct entities, while our dataset consists of 52,004 movies and 2,272,504 distinct entities.

When all actors with the same last name had been merged, the resulting graph contained 150,072 nodes, corresponding to the distinct last names, and 9,283,233 edges. In Malin [39] they reported ∼85,000 distinct last names. We created a training set by randomly selecting 100 identifiers, such that half of them were ambiguous. The average size for each local neighborhood graph of the labeled nodes was 156 nodes and 12,028 edges. We denote this dataset IMDb1.

In IMDb1 there exists several identifiers that appear in one movie only. In fact, out of the 100 identifiers chosen, 38 was part of only one source. Out of these 38 identifiers, 36 were non-ambiguous, while 2 were ambiguous. Consistent with

[2] `http://www.imdb.com/interfaces`

intuition, it is very unlikely that an actor appearing in only one movie is ambiguous. This, perhaps undesired, property of the dataset led us to create a second set based on the IMDb data. This dataset contains the same data, the only difference being that the 100 identifiers, chosen for evaluation, are required to be a part of more than one movie. In the resulting set, the average number of nodes and edges in each local neighborhood were 279 and 24,550 respectively. We denote this alternative dataset IMDb2.

6.3 Experimental Setup

We evaluate the performance of five different graph kernels (GL, DP, SP, TDP, BSD), including extensions, within our approach using 10-fold cross validation with the Pegasos SVM solver [48]. For all kernels we use only the 1-neighborhood ($\kappa = 1$) for comparing nodes. This reduces computation time, but is also motivated from the intuition that the larger the neighborhood included, the less specific it is to the node of interest.

We transform all edge weights w_{ij} with a function $\tau : \mathbb{R}^+ \mapsto \mathbb{R}^+, \tau' < 0$ so that strong connections (many co-occurrences) become small weights, in the belief that shortest paths should go along strong connections. In our experiments, we try logarithmic and inverse edge transforms, defined as: $\tau_{\text{log}}(w_{ij}) = \ln w_{max} - \ln w_{ij}$ and $\tau_{\text{inv}}(w_{ij}) = w_{ij}^{-1}$, where w_{max} is the maximum edge weight in the graph. Note that the transformations are applied prior to the computation of the distances S_{ij}^G with the Floyd-Warshall algorithm. The transforms only affect the BSD and SP kernels, as the other kernels do not consider edge weights. For the BSD kernel, we use only linear binning, as defined in (4.11).

We believe that using linear binning is sufficient, since the edge transform τ already preprocesses the edges in a non-linear manner. The logarithmic edge transform allows our kernel to capture the intuition that there exist many weak connections and few very strong connections. Achieving good binning resolution in both cases motivates the use of the logarithm. Figure 6.1, depicting the weight distribution of the two real-world datasets, supports this intuition.

For the TDP kernel, we perform a grid search over the parameters, setting $\lambda \in \{0.2, 0.4, 0.6, 0.8\}$ and $K = \lfloor \ln_\lambda 10^{-5} \rfloor$, giving $K \in \{7, 12, 22, 51\}$. For the BSD kernel, we try setting the number of bins $M \in \{5, 10, 25, 50\}$. As both the above kernels can be normalized, we try enabling and disabling normalization for these.

Kernel	RF	IMDB1	IMDB2
GL	81.0	84.0	**71.0**
SP	73.0	77.0	60.0
BSD_n	**82.0**	82.0	61.0
BSD	78.0	78.0	61.0
DP	54.9	37.0	-
TDP_n	75.0	74.0	62.0
TDP	75.0	65.0	62.0
TDP_{dn}	76.0	77.0	65.0
TDP_d	72.0	76.0	55.0
MALIN		**86.0**	64.0

Table 6.1: Classification accuracy (%) on 10-fold cross-validation. The subscript n stands for normalization and d for the distinguished vertex version of the TDP kernel. The DP kernel on IMDB2 did not finish within in 48 hours and was left out. Note that MALIN is not applicable to the RF dataset as explained in Section 6.1.

For the GL kernel, we used $\epsilon = \delta = 0.05$. The parameter λ of the DP kernel was set for each dataset so that $\lambda < \Delta(G_\times)^{-1}$, giving $\lambda = 0.00067$ for RF and $\lambda = 0.00098$ for IMDB1. The DP kernel on IMDB2 did not finish within 48 hours and was left out because of this. For all kernels, we optimize the Pegasos SVM parameters using cross-validation.

For MALIN, we started 100 and 1000 random walks from each source of the node being classified, and for each walk we did 50 steps. Early tests showed that increasing the number of steps did not increase performance. Note that in Malin [39], only 100 walks were used, albeit on a different dataset. On IMDB1 and IMDB2, however, more walks could be needed, as they are of larger size. We then tested for the best threshold from the set $\{0.00, 0.01, \dots, 1.00\}$.

6.4 Results

The best results of our experiments, in terms of classification accuracy on the three datasets, are shown in Table 6.1. The GL, SP and DP kernels are the original

Kernel†	RF	IMDB1	IMDB2
GL	424	1254	1221
SP	1010	5247	14151
BSD_n	1721	6706	19325
BSD	1690	8274	12143
DP	14980	131329	-
TDP_n	3	9	25
TDP	3	9	24
TDP_{dn}	8	9	19
TDP_d	8	9	25
MALIN		2787	49362

Table 6.2: Wall clock time for detecting ambiguities for all training examples, measured in seconds [3]. The subscript n stands for normalization and d for the distinguished vertex version of the TDP kernel. † The GL kernel was implemented in MATLAB, and MALIN and DP in Python. All others were run in C++ using GraphLab. The DP kernel on IMDB2 did not finish within in 48 hours and was left out. Note that MALIN is not applicable to the RF dataset as explained in Section 6.1.

kernels as described in Section 2. The DP kernel was computed using the Sylvester equation method of Vishwanathan et al. [53] and the SP kernel used the indicator function as in (4.9). The BSD and TDP kernels are the extensions of the SP and DP kernels respectively, proposed in Section 4. In Table 6.2, the wall-clock times needed for detecting ambiguities using the various methods are presented.

Preliminary results indicated that the performance did not improve with $\kappa > 1$. Further experiments with $\kappa > 1$ were not performed because of this fact and the intuition that the larger the size of the extracted neighborhoods, the less they will describe the specific characteristics of the central vertex. When using larger neighborhoods, the graph sizes increase rapidly, reaching close to the size of the original identifier graph already for $\kappa = 2$. It was thus infeasible to run these experiments within reasonable time.

We proceed to compare the results of our method with MALIN based on the results

[3] 8 threads on a cluster with 5 computation nodes and 8 Intel Xeon 2.4 GHz cores per node was used.

in Table 6.1. In the RF dataset, there is no source information available, making it is impossible to run experiments for Malin on this dataset.

We now summarize the parameters giving the best performance during experiments. The TDP kernel performed best with $\lambda = 0.2$ for the RF dataset, $\lambda = 0.8$ on the IMDb1 dataset and $\lambda = 0.6$ on the IMDb2 dataset. The SP kernel performed best with inverse edge transform on the RF dataset and logarithmic edge transform on the IMDb1 and IMDb2 datasets. The BSD kernel with logarithmic edge transform on the RF dataset gave the best result, using 10 linear bins. On the IMDb1 and IMDb2 datasets, the BSD kernel with logarithmic edge transform and 25 and 5 linear bins, respectively, gave the best result.

For Malin, on IMDb1, the best number of walks was 100, and the best threshold 0.00. On IMDb2, Malin got the best accuracy with 1000 walks and the threshold being 0.17.

Our approach (GL, BSD) gets comparable results to Malin on IMDb1, while being significantly faster (GL), see Table 6.2. Our approach outperforms Malin by a wide margin on IMDb2. IMDb1 contains identifiers that only appeared in one movie, which in our setting means that we have very little information to base the classification on. Also, Malin predicts all single-source nodes to be non-ambiguous by default. Since most of these nodes actually are non-ambiguous, this means that Malin works quite well on the IMDb1 dataset.

In fact, any classifier that has access to the sources would be able to get a very good result on IMDb1 by classifying all nodes with more than one source as ambiguous and the rest as non-ambiguous. Our method however does not have access to the sources and still gives similar performance, with many of our kernels taking significantly smaller amount of time. This simple classification method, for classifiers that have access to the sources, is however not very useful on the more challenging dataset IMDb2, and thus Malin gets a worse result on that dataset.

We also note that Malin takes significantly longer time to run on the IMDb2 dataset, compared to IMDb1. This is because of the fact that IMDb2 contains more sources per node on average, increasing the number of walks performed by Malin. Also, Malin got the best result with 1000 walks per source node, instead of 100 like on IMDb1, this also increased the running time significantly. With 100 walks Malin got only 56% accuracy and took 3,957 seconds to run.

In our graph kernel-based approach, the time complexity for training the classifier is independent of the number of sources the data contained. Although the compu-

tation time will increase if the number of edges increases, increasing the number of sources will not increase the computation time unless it introduces new edges into the graph. New edges will be added if a source introduces a new pair of identifiers not previously encountered together in a source. New sources without any new pairs of identifiers will then only increase the weight of edges and therefore not affect the computation time of our method. This behavior is in contrast to MALIN, which increases linearly in computation time with the number of sources. Since the number of sources potentially can be very big in real world datasets, this time dependence on the number of sources could in some cases be a problem. If we assume a scenario where we have access to a very large amount of sources, then as long as a new source does not introduce a new edge, including that source in the dataset will not increase the computation time of our graph kernel approach, giving us a *significant* advantage over MALIN in this setting. When it comes to running time, the GL and TDP kernels finished faster than MALIN on both datasets, with the TDP kernel being particularly noteworthy.

On a final note, given the speed of our method, it could be used as a potential preprocessing step, whereafter a clustering method such as MALIN could be run on the filtered smaller selection of nodes, dividing them into their correct underlying entities. This, of course, assumes a scenario where the sources are observed. It would not work on the RF dataset, for which further investigation needs to be done.

Evaluation of Kernel Extensions In terms of accuracy, the BSD and GL kernels outperform the other kernels when it comes to the RF dataset. On the IMDB1 dataset GL performs the best of the kernels, with the normalized BSD kernel close behind. On the more challenging dataset IMDB2, the GL kernel performed the best.

We note that both of our extended kernels outperformed their original counterparts. For the DP kernel, this is especially noteworthy due to the timing results of Table 6.2. The TDP kernel variations all perform significantly better to a fraction of the computational cost. A possible reason for why the TDP kernel would perform better than the DP kernel is *tottering*, as described in Section 2.21. Since calculating the DP kernel is the same as calculating all walks of length up to infinity, tottering actually outweighs the relevant walks, making the DP kernel perform worse than a kernel that only calculates walks up to a certain length, i.e. the TDP kernel.

Using normalization helped the BSD and TDP kernels get better or equal accuracy on all datasets, indicating that the normalization might help the kernels become less sensitive to graph size. We also note that the BSD kernel outperforms the SP

kernel on both datasets, indicating that binning seems to help the kernel get a better classification accuracy on weighted graphs.

The experiments with the distinguished node for the TDP kernel (TDP_{dn}), gave a slight increase in performance, but not enough to say that the TDP_{dn} kernel is better than the TDP_{n} kernel.

The experiments indicate that the edge transform $\tau_{\log}$ generally performs better than τ_{inv}. The only exceptions to this was the BSD kernel on the IMDb2 dataset and the SP kernel on the RF dataset.

Speedup of BSD Kernel Using Parallel Implementation As the BSD and TDP kernels can be calculated algorithmically an order of magnitude faster than naïve pairwise kernel computation, it is uninteresting to compare this increase in computation speed. Indeed, the TDP kernel runs much faster than any of the other approaches in our experiments. However, with a running time of a few seconds, investigating the scalability of the TDP kernel would require a larger dataset. Rather, we focus on investigating how well the computation of shortest distances for the BSD kernel scales when using GraphLab. The experiments were carried out on the RF dataset. A cluster with 5 computation nodes with 8 Intel Xeon 2.4 GHz cores and 23.5 GB RAM per node was used. Figure 6.2 shows a comparison with linear speedup. Speedup was defined as $\frac{t_1}{t_k}$, where t_k is the time required to precompute the graph kernels using k CPUs. In the figure, *serial* means running GraphLab on each local neighborhood graph separately, whereas *batch* denotes putting all local neighborhoods in one big graph, on which GraphLab is run just once. *Spread* denotes running the threads evenly divided on several physical machines, whereas *tight* denotes running as many threads per machine as there are CPU cores. We see that the batch-tight computation mode gets the best speedup, increasing almost linearly up until 8 CPUs.

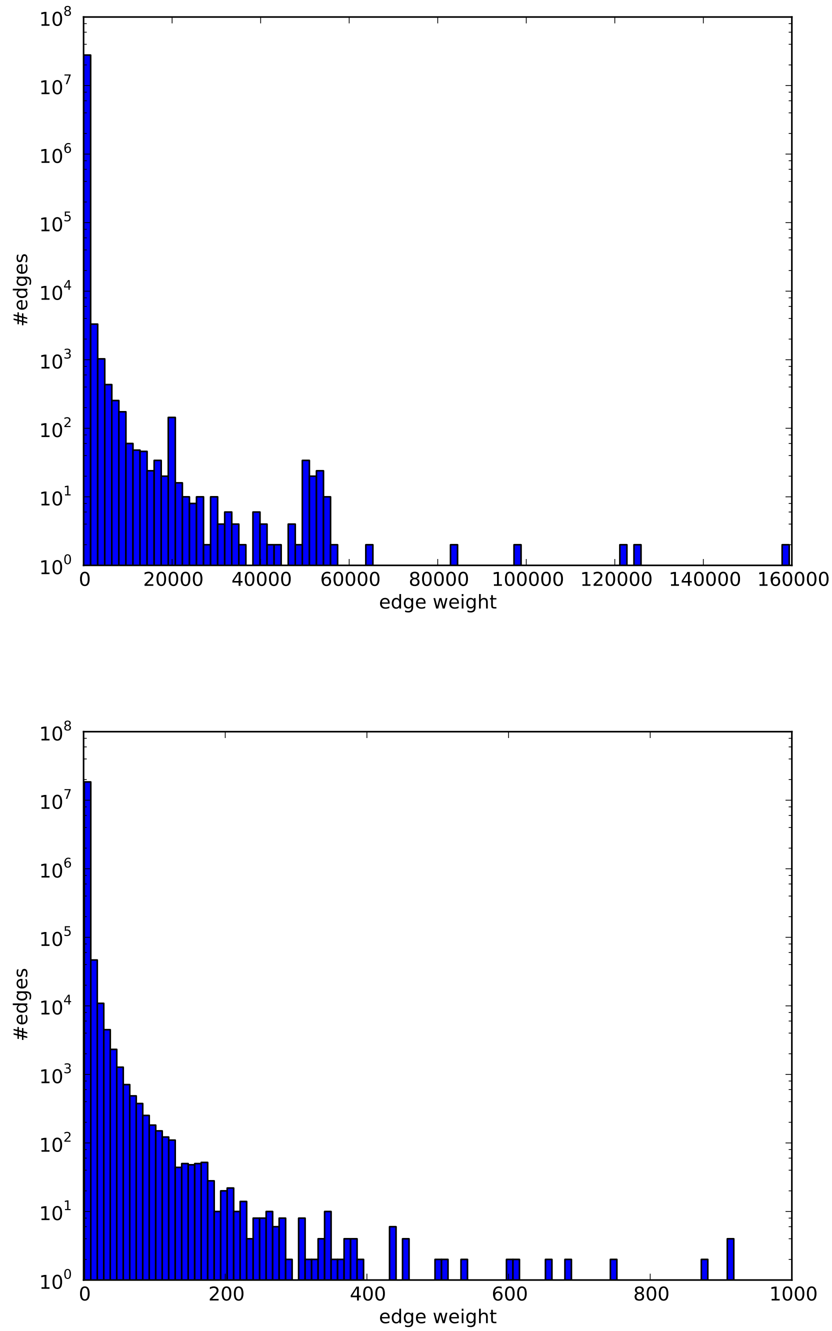


Figure 6.1: Histogram of edge weights in the RF (top) and IMDB (bottom) datasets, with the number of edges on the y-axis and edge weight magnitude on the x-axis. Note that the y-axis is on a logarithmic scale.

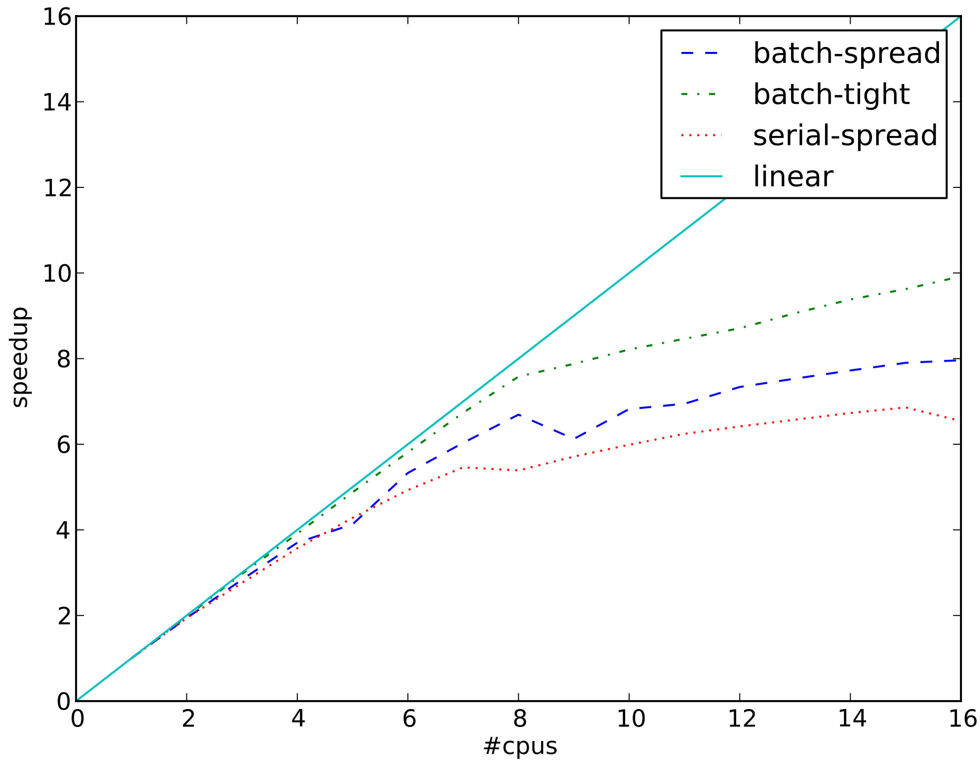


Figure 6.2: Parallel speedup of BSD kernel precomputation as the number of CPUs increase on the RF dataset. Linear speedup is shown by the solid line. Speedup is defined as $\frac{t_1}{t_k}$, where t_k is the time required to precompute the graph kernels using k CPUs. We let *serial* denote running GraphLab on each local neighborhood graph separately, whereas *batch* denotes putting all local neighborhoods in one big graph, on which GraphLab is run just once. *Spread* denotes running the threads evenly divided on several physical machines, whereas *tight* denotes running as many threads per machine as there are CPU cores.

7

Discussion

In this thesis we have designed extensions to the DP and SP kernels, creating the TDP and the BSD kernels. During our experiments in Section 6, we showed that these kernels perform similarly or better in terms of speed, accuracy or both, compared to their original versions. The run-time decrease of the TDP kernel was particularly noteworthy. On the RF dataset, the running time of the DP kernel was over 4 hours, while the running time of the TDP kernel was a mere 3 seconds. The TDP kernel also outperformed the DP kernel in terms of accuracy, most likely due to the problem of tottering. The BSD kernel also performed very well compared to its original version, the SP kernel. The BSD kernel had better accuracy than the SP kernel on all datasets. While having a longer running time than the SP kernel in the experiments, the increase was not significant enough to say that the BSD kernel is always slower. The difference in running time could be due to differentiating loads on the server used for the experiments. Normalizing the kernels helped them become less sensitive to graph size and gave increased performance. In some cases, the performance increase was quite significant. The version of the TDP kernel which only considers the random walks of the distinguished node, the TDP_{dn} kernel, showed a slight increase in accuracy, but the increase was not judged to be enough for drawing the conclusion that the TDP_{dn} kernel is better than the TDP_{n} kernel.

When comparing our kernel-based approach to Malin, we can note that we outperform Malin by a wide margin on the IMDb2 dataset. For reasons described in Section 6, it is very difficult for a method that does not use source information, such

as our kernel-based method, to beat any method which considers the sources, be that MALIN or any other source-aware method. Despite this, we perform very similar in terms of accuracy, compared to MALIN, and several of the kernels are able to run in a fraction of the time used by MALIN.

We have also presented a way for significantly reducing the running time of our approach, by explicitly knowing the mapping ϕ. Given ϕ, we are able to calculate the time-consuming part of the kernel for each *single* graph, instead of for each *pair* of graphs, reducing the running time by an order of magnitude. Since the most computationally demanding part of our approach is calculating the kernel values — not solving the SVM optimization problem — this was something which greatly helped decrease the running time of our approach.

During our work we also looked into parallelizing the Pegasos algorithm, but the parallelized algorithm, implemented in GraphLab, actually took longer time to run than the original Pegasos algorithm. This because of the fact that the overhead incurred from GraphLab was bigger than the time saved by calculating the different steps in parallel.

During all of the experiments, the GL kernel performed very well in terms of accuracy. This despite the fact that the GL kernel does not consider the weight of edges. The GL kernel is also competitive in terms of running-time, beaten only by the TDP kernel. This makes the GL kernel very interesting for more detailed studies and designing extensions for it. This was however not done in this thesis due to time constraints.

8

Conclusions and Future Work

In this thesis we have developed a novel formulation of anonymized relational entity disambiguation as a graph classification problem. We have devised a method for detecting ambiguities in graph data based on local neighborhoods using graph kernels. Our approach was compared to a state-of-the-art method, showing significantly better performance in terms of either accuracy or speed or both.

We have shown how to enable fast computation of the direct product (DP) kernel by extending it to a truncated direct product (TDP) kernel that outperformed the DP kernel in our experiments. We have also presented a simple extension of the shortest path kernel (SP), creating the binned shortest distance (BSD) kernel, as a way of measuring similarity between general weighted graphs. The BSD kernel was shown to give good classification results, outperforming the base SP kernel on weighted graphs, which motivates the need for binning. Normalization of the TDP and BSD kernels also helped boost performance. We show a significant speedup in the computation of the kernels when using GraphLab and by explicitly knowing the kernel mapping ϕ.

Future research should examine the possibility of applying our method in a semi-supervised setting, by considering unlabeled samples. Other aspects to be looked into is the problem of actually associating ambiguous identifiers with their underlying entities. It should be noted that our method can be used as a fast indicator of which identifiers to examine more closely for ambiguity. Larger datasets should be examined, in order to further investigate the scalability of our method. Given

the good performance of the GL kernel in our experiments, it is of great interest to attempt at designing an extension to the GL kernel, which considers weighted graphs.

A

Mathematical Notation

THIS APPENDIX lists the usage of mathematical notation, including variables, conventions and assumptions used in this thesis. The reader is advised to get familiar with this section before indulging their curiosity about the technical parts of the thesis.

We use lower-case bold letters $\mathbf{a} = [a_1, \ldots, a_n]^T$ to denote vectors and a_i denotes the i-th element of the vector.

We use bold upper-case letters $\mathbf{A}$,$\mathbf{B}$,$\mathbf{C}, \ldots$ to denote matrices with A_{ij} referring to the element at the i-th row and j-th column of a $p \times q$ matrix:

$$\mathbf{A} = \begin{bmatrix} A_{11} & \cdots & A_{1q} \\ \vdots & \ddots & \vdots \\ A_{p1} & \cdots & A_{pq} \end{bmatrix} \tag{A.1}$$

Let $\mathbf{A}^n$ denote the n-th power of $\mathbf{A}$. Let $\mathbf{a}^{(n)}$ denote the n-th vector in a set of vectors.

The Euclidean norm of $\mathbf{a}$ is denoted by $\|\mathbf{a}\|$.

Let $\mathbb{1}(\cdot)$ denote the indicator function such that

$$\mathbb{1}(x) = \begin{cases} 1 & \text{if the expression } x \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Let $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ denote the floor and ceiling functions, respectively.

$[m]$ denotes the set with integer elements $\{1, 2, \ldots, m\}$.

We use $\mathbf{e} = [1, \ldots, 1]^T$ to denote a vector of all 1's of the appropriate size.

We let $\mathbf{e}_i = [0, \ldots, 1, \ldots, 0]^T$ denote a vector with the i-th element set to 1 and all other elements zero.

We let $\mathbf{A} \otimes \mathbf{B} = \mathbf{C}$ (of size $mp \times nq$) denote the Kronecker product [15] of matrices $\mathbf{A}$ (of size $m \times n$) and $\mathbf{B}$ (of size $p \times q$).

We let $G = (V,E)$ denote a graph with vertex set $V = \{v_1, \ldots, v_n\}$ and edge set $E = \{e_k : e_k = (v_i, v_j) \Leftrightarrow v_i \sim v_j,\ v_i,v_i \in V\}$, and $A(G)$ the adjacency matrix of G

$\Delta(G)$ denotes the maximum degree of G.

Let $V^{(i)} \times V^{(j)}$ denote the Cartesian product of vertex sets $V^{(i)}$ and $V^{(j)}$.

We let $S_{ij}^{(G)}$ denote the shortest path from node i to j in graph G.

$K(G^{(i)}, G^{(j)}) = \langle \phi(G^{(i)}), \phi(G^{(j)}) \rangle = K_{ij}$ denotes a kernel [45] on graphs $G^{(i)}$, $G^{(j)}$, where $\phi : \mathcal{X} \mapsto \mathcal{H}$ is a map into a Hilbert space $\mathcal{H}$ [45]. Whenever clear from context, we will omit G.

B

Mathematical Proofs

THIS APPENDIX contains detailed proofs and derivations for some of the equations stated in this thesis.

B.1 Proof of Equation 2.18, p. 19

Proof.

(From (2.17)) Base case: $(\mathbf{A} \otimes \mathbf{B})^2 = \mathbf{A}^2 \otimes \mathbf{B}^2$

Induction step: $(\mathbf{A} \otimes \mathbf{B})^{n+1} = (\mathbf{A} \otimes \mathbf{B})^n (\mathbf{A} \otimes \mathbf{B}) =$
$(\mathbf{A}^n \otimes \mathbf{B}^n)(\mathbf{A} \otimes \mathbf{B}) = \mathbf{A}^{n+1} \otimes \mathbf{B}^{n+1}$

□

B.2 Proof of Equation 2.19, p. 19

Proof.

$$\sum_{i,j} [\mathbf{A} \otimes \mathbf{B}]_{ij} = \mathbf{e}^T(\mathbf{A} \otimes \mathbf{B})\mathbf{e} = \mathbf{e}^T \begin{bmatrix} A_{11}\mathbf{B} & \cdots & A_{1q}\mathbf{B} \\ \vdots & \ddots & \vdots \\ A_{p1}\mathbf{B} & \cdots & A_{pq}\mathbf{B} \end{bmatrix} \mathbf{e} =$$

$$\mathbf{e}^T(A_{11} + \cdots + A_{1q} + \cdots + A_{p1} + \cdots + A_{pq})\mathbf{B}\mathbf{e} =$$

$$\mathbf{e}^T(\mathbf{e}^T\mathbf{A}\mathbf{e})\mathbf{B}\mathbf{e} = \mathbf{e}^T\mathbf{A}\mathbf{e}\mathbf{e}^T\mathbf{B}\mathbf{e}$$

□

B.3 Derivation of Equation 4.10, p. 35

Proof.

$$\begin{aligned} K_{SP}(G^{(1)}, G^{(2)}) &= \sum_{i,j,p,q} \mathbb{1}\left[S_{ij}^{G^{(1)}} = S_{pq}^{G^{(2)}}\right] \\ &= \sum_{i,j,p,q} \sum_{k=1}^{M} \mathbb{1}\left[S_{ij}^{G^{(1)}} = k\right] \mathbb{1}\left[S_{pq}^{G^{(2)}} = k\right] \\ &= \sum_{k=1}^{M} \sum_{i,j,p,q} \mathbb{1}\left[S_{ij}^{G^{(1)}} = k\right] \mathbb{1}\left[S_{pq}^{G^{(2)}} = k\right] \\ &= \sum_{k=1}^{M} \sum_{i,j} \mathbb{1}\left[S_{ij}^{G^{(1)}} = k\right] \sum_{p,q} \mathbb{1}\left[S_{pq}^{G^{(2)}} = k\right] \end{aligned}$$

Replacing $\mathbb{1}\left[S_{ij}^{G^{(1)}} = k\right]$ with $\theta_{ijk}^{G^{(1)}}$ gives us the BSD kernel:

$$K_{BSD}(G^{(1)}, G^{(2)}) = \sum_{k=1}^{M} \sum_{i,j} \theta_{ijk}^{G^{(1)}} \sum_{p,q} \theta_{pqk}^{G^{(2)}}$$

□

B.4 Derivation of Equation 4.5, p. 33

Proof. For convenience, we denote $A(G^{(i)})$ with $\mathbf{A}^{(i)}$.

$$K_\times(G^{(1)}, G^{(2)}) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda^n A(G_\times)^n \right]_{ij} =$$

$$\sum_{n=0}^{\infty} \lambda^n \sum_{i,j=1}^{|V_\times|} \left[A(G_\times)^n \right]_{ij} = \sum_{n=0}^{\infty} \lambda^n \sum_{i,j=1}^{|V_\times|} \left[(\mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)})^n \right]_{ij} =$$

$$\sum_{n=0}^{\infty} \lambda^n \sum_{i,j=1}^{|V_\times|} \left[\mathbf{A}^{(1)^n} \otimes \mathbf{A}^{(2)^n} \right]_{ij} = \sum_{n=0}^{\infty} \lambda^n \mathbf{e}^T \mathbf{A}^{(1)^n} \otimes \mathbf{A}^{(2)^n} \mathbf{e} =$$

$$\sum_{n=0}^{\infty} \lambda^n \mathbf{e}^T (\mathbf{e}^T \mathbf{A}^{(1)^n} \mathbf{e}) \mathbf{A}^{(2)^n} \mathbf{e} = \sum_{n=0}^{\infty} \lambda^n (\mathbf{e}^T \mathbf{A}^{(1)^n} \mathbf{e})(\mathbf{e}^T \mathbf{A}^{(2)^n} \mathbf{e})$$

□

Bibliography

[1] Charu C Aggarwal and S Yu Philip. *A condensation approach to privacy preserving data mining*. Springer, 2004.

[2] Francis R. Bach. Graph kernels between point clouds. In *Proc. of ICML*, 2008.

[3] Ron Bekkerman and Andrew McCallum. Disambiguating web appearances of people in a social network. In *Proc. of WWW*, 2005.

[4] Indrajit Bhattacharya and Lise Getoor. Iterative record linkage for cleaning and integration. In *Proc. of DMKD*, 2004.

[5] Indrajit Bhattacharya and Lise Getoor. Entity resolutions in graphs. In *Mining Graph Data*. Wiley, 2006.

[6] Indrajit Bhattacharya and Lise Getoor. A latent dirichlet model for unsupervised entity resolution. In *Proc. of SDM*, 2006.

[7] Mikhail Bilenko and Raymond J Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 39–48. ACM, 2003.

[8] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[9] Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. Linda: distributed web-of-data-scale entity matching. In *Proc. of CIKM*, 2012.

[10] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Prof. of ICDM*, 2005.

[11] Karsten M Borgwardt, Cheng Soon Ong, Stefan Schönauer, SVN Vishwanathan, Alex J Smola, and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl 1):i47–i56, 2005.

[12] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, pages 144–152. ACM, 1992.

[13] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[14] Stephen Boyd and Almir Mutapcic. Subgradient methods. *Lecture notes of EE364b, Stanford University, Winter Quarter*, 2007.

[15] John Brewer. Kronecker products and matrix calculus in system theory. *Circuits and Systems, IEEE Trans. on*, 25(9):772–781, 1978.

[16] Razvan Bunescu and Marius Pasca. Using encyclopedic knowledge for named entity disambiguation. In *Proc. of EACL*, 2006.

[17] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[18] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 313–324. ACM, 2003.

[19] Zhaoqi Chen, Dmitri V Kalashnikov, and Sharad Mehrotra. Exploiting relationships for object consolidation. In *Proceedings of the 2nd international workshop on Information quality in information systems*, pages 47–58. ACM, 2005.

[20] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[21] Silviu Cucerzan. Large-scale named entity disambiguation based on wikipedia data. In *Proc. of EMNLP-CoNLL*, 2007.

[22] Christopher P Diehl, Lise Getoor, and Galileo Namata. Name reference resolution in organizational email archives. In *Prof. of SDM*, 2006.

[23] Cynthia Dwork. Differential privacy: A survey of results. In *Proc. of TAMC*. 2008.

[24] Tamer Elsayed, Douglas W Oard, and Galileo Namata. Resolving personal names in email using context expansion. In *Proc. of ACL*, 2008.

[25] T. Fletcher. Support vector machines explained. *Tutorial paper., Mar*, 2009.

[26] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[27] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. *Learning Theory and Kernel Machines*, pages 129–143, 2003.

[28] Christopher David Godsil, Gordon Royle, and CD Godsil. *Algebraic graph theory*, volume 8. Springer New York, 2001.

[29] Liang Jin, Chen Li, and Sharad Mehrotra. Efficient record linkage in large data sets. In *Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings. Eighth International Conference on*, pages 137–146. IEEE, 2003.

[30] Dmitri V Kalashnikov, Sharad Mehrotra, and Zhaoqi Chen. Exploiting relationships for domain-independent data cleaning. In *SIAM data mining (SDM) conf*, 2005.

[31] U Kang, Hanghang Tong, and Jimeng Sun. Fast random walk graph kernel. In *Proc. of SDM*, 2012.

[32] George Kimeldorf and Grace Wahba. Some results on tchebycheffian spline functions. *Journal of Mathematical Analysis and Applications*, 33(1):82–95, 1971.

[33] Risi Imre Kondor and John Lafferty. Diffusion kernels on graphs and other discrete structures. In *Proc. of ICML*, 2002.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale graph computation on just a PC. OSDI, 2012.

[35] Kun Liu, Kamalika Das, Tyrone Grandison, and Hillol Kargupta. Privacy-preserving data analysis on graphs and social networks. In Hillol Kargupta, Jiawei Han, Philip S. Yu, Rajeev Motwani, and Vipin Kumar, editors, *Next Generation of Data Mining*. Chapman and Hall/CRC, 2008.

[36] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. of VLDB*, 2012.

[37] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[38] Pierre Mahé, Nobuhisa Ueda, Tatsuya Akutsu, Jean-Luc Perret, and Jean-Philippe Vert. Extensions of marginalized graph kernels. In *Proc. of ICML*, 2004.

[39] Bradley Malin. Unsupervised name disambiguation via social network similarity. In *Workshop on link analysis, counterterrorism, and security*, volume 1401, pages 93–102, 2005.

[40] Aditya Krishna Menon. Large-scale support vector machines: algorithms and theory. *Research Exam, University of California, San Diego*, 2009.

[41] Kaare Brandt Petersen and Michael Syskind Pedersen. The matrix cookbook. *Technical University of Denmark*, pages 7–15, 2008.

[42] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[43] Carlos C Rodriguez. The kernel trick. `http://omega.albany.edu:8008/machine-learning-dir/notes-dir/ker1/ker1-l.html`, 2004.

[44] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[45] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. MIT press, 2001.

[46] Prithviraj Sen. Collective context-aware topic models for entity disambiguation. In *Proc. of WWW*, 2012.

[47] S. Shalev-Shwartz, Y. Singer, and N. Srebro. Pegasos: Primal estimated subgradient solver for SVM. In *Proceedings of the 24th international conference on Machine learning*, pages 807–814. ACM, 2007.

[48] Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro, and Andrew Cotter. Pegasos: primal estimated sub-gradient solver for SVM. *Mathematical Programming*, 127(1):3–30, 2011.

[49] Shai Shalev-Shwartz and Nathan Srebro. Svm optimization: inverse dependence on training set size. In *Proceedings of the 25th international conference on Machine learning*, pages 928–935. ACM, 2008.

[50] Nino Shervashidze and Karsten M. Borgwardt. Fast subtree kernels on graphs. In Yoshua Bengio, Dale Schuurmans, John D. Lafferty, Christopher K. I. Williams, and Aron Culotta, editors, *NIPS*, pages 1660–1668. Curran Associates, Inc., 2009.

[51] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[52] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proc. of AISTATS*, 2009.

[53] SVN Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *JMLR*, 2010.