LARGE SCALE DATA MINING WITH APPLICATIONS IN SOCIAL COMPUTING

by

Shagun Jhaver

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Latifur Khan, Chair

Dr. Farokh B. Bastani

Dr. Bhavani Thuraisingham

This material is based upon work supported by
National Science Foundation under Award No. CNS 1229652,
and the Air Force Office of Scientific Research under Award No.
FA-9550-09-1-0468 and Award No. FA-9550-12-1-0077.
We thank Dr. Robert Herklotz for his support.

LARGE SCALE DATA MINING WITH APPLICATIONS IN SOCIAL COMPUTING

by

SHAGUN JHAVER, B. Tech

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2014

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Professor Latifur Khan for his continuous support of my research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. One simply could not wish for a better or kinder supervisor.

Besides my advisor, I would like to thank the rest of my thesis committee: Professor Farokh B. Bastani and Professor Bhavani Thuraisingham, for their encouragement and insightful comments.

I am grateful to Professor Haim Schweitzer and Professor Balaji Raghavachari for their moral support through some difficult times. I would also like to thank Professor Yang Liu and Professor Vibhav Gogate for helping me develop my background in natural language processing and probabilistic graphical models. I thank my fellow labmates in the Data Mining Group: Swarup Chandra, Khaled Alnaami, Ahsanul Haque and Rakib Solaimani for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last years.

Last but not the least, I would like to thank my family: my beloved parents, Balkrishna and Uma Jhaver, for their unconditional love and care and continuous spiritual guidance.

November 2014

LARGE SCALE DATA MINING WITH APPLICATIONS IN SOCIAL COMPUTING

Publication No. _____

Shagun Jhaver, MS
The University of Texas at Dallas, 2014

Supervising Professor: Dr. Latifur Khan

The aim of this thesis is to analyze large scale data mining and its applications in the domain of social computing. This study sought to investigate the following case studies:

Firstly, a description of a new framework for stream classification is presented. This framework predicts class labels for a set of instances in a data stream and uses various machine learning techniques to perform this classification. The framework is evaluated using both real-world and synthetic datasets, including a dataset used to perform a website fingerprinting attack by viewing it as a setwise classification problem.

Secondly, an investigation of the parallelization of calculating edit distance for a large set of string pairs using the MapReduce framework is presented. This study demonstrates how large scale data mining opens new avenues of designing for dynamic programming algorithms.

Thirdly, a comparative analysis of classifiers predicting politeness in a framework proposed by Danescu-Niculescu-Mizil et al is detailed. An application of this framework to study politeness in various web-logs is also presented.

Finally, a discussion of different approaches to sentiment analysis of Twitter posts is presented. An application of this processing to predict the rating of newly released movies is also developed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This chapter introduces the case studies investigated through the course of research for this thesis. A brief description of the problems analyzed in each case study, the implementations chosen, and the results observed is presented.

1.1 Setwise Stream Classification

Traditional stream data classification involves predicting a class label for each data instance in a data stream. However, such inference of class labels may not be practical if a label is associated with a set of data instances rather than a single instance. Characteristics of a class can be represented by a data distribution in the feature space over which stochastic models can be developed to perform classification for sets of data instances. Further, data mining on streaming data exhibits multiple challenges such as concept drift and concept evolution. In this study, we develop a generic framework to perform setwise stream classification where class labels are predicted for a set of instances rather than individual ones. In particular, we design a fixed-size ensemble based classification approach using various machine learning techniques to perform classification. We use both real-world and synthetic datasets, including a dataset used to perform a website fingerprinting attack by viewing it as a setwise classification problem, to evaluate the framework. Our evaluation shows an improved performance, for all datasets, over an existing method. A detailed description of this study is presented in Chapter 2.

1.2 Edit Distance calculation

Given two strings X and Y over a finite alphabet, the edit distance between X and Y, $d(X, Y)$ is the number of elementary edit operations required to edit X into Y. A dynamic programming algorithm elegantly computes this distance. In this study, we investigate the parallelization of calculating edit distance for a large set of strings using MapReduce, a popular parallel computing framework. We propose SIM_MR and PRE_MR algorithms, parallel versions of the dynamic programming solution, and present implementations of these algorithms. We study different cases by varying algorithm parameters, input size and number of parallel nodes, and analytically and experimentally confirm the superiority of our methods over the usual dynamic programming approach. This study demonstrates how MapReduce parallelization opens new avenues of designing for dynamic programming algorithms. This study is detailed in Chapter 3.

1.3 Politness Classifiers

This study presents an implementation of a computational framework for identifying and characterizing politeness markings in text documents proposed in the Paper 'A computational approach to politeness with application to social factors' (Danescu-Niculescu-Mizil et al., 2013) by Danescu-Niculescu-Mizil et al. A comparative analysis of the classifiers for this task constructed using a variety of different algorithms, filters and features is presented. The framework has also been used to study the politeness levels in a variety of web-logs. The details of this study are presented in Chapter 4.

1.4 Sentiment Analysis of Twitter Messages

This study describes the implementations of a variety of techniques to process data from the social network website Twitter, and generate useful information from it. It is assumed that

the vast and free use of social networks like Twitter generates data that is unskewed and unbiased, and is ripe for predictive analytics.

We show an application of twitter data processing. Three different approaches have been taken to mine the tweets for the year 2012 to predict the rating (on a scale of 0 - 10) and box office performance of movies released in 2012. First, the tweets that are related to the movies released in 2012 are filtered. The first approach to mining these tweets uses a list of positive words and a list of negative words to classify each tweet. The second approach uses the sentiment140 api for the classification. The third approach creates a classification model using a training-and-testing data with naive Bayes method, and then uses this model to classify each tweet, thereby rating it. After the rating for each tweet has been generated, an average over the ratings of each movie is calculated to predict the overall rating for each movie. Finally, the rating for each movie is shown alongside a standard (IMDB) rating for comparative analysis. A detailed description of this study is presented in Chapter 5.

CHAPTER 2

A FRAMEWORK FOR ENSEMBLE BASED SETWISE STREAM CLASSIFICATION [1]

This chapter develops a framework to perform setwise stream classification where class labels are predicted for a set of instances rather than individual instances in a stream. We also design a fixed-size ensemble based classification approach using various machine learning techniques to perform classification. We evaluate this framework using both real-world and synthetic datasets.

2.1 Introduction

In recent years, ubiquitous availability of streaming data from sources such as web, social networks and sensor networks has spurred a wide interest in discovering meaningful patterns in such data, among research communities as well as technology businesses. An imminent need to make this data usable has lead to design of new algorithms for stream analysis, and their use in applications such as anomaly detection (Lee and Stolfo, 1998), market analysis, etc. In particular, data classification involves an assignment of a class label to each data instance in a dataset. A stochastic model or classifier is developed and is used to predict a class label for each data instance in the dataset (Domingos and Hulten, 2000). However, in certain cases, assignment of a class label to an individual data instance may not be valid. A class label may instead be associated with certain patterns of data distribution that reflect characteristics of the class. In such cases, a class label may only be associated with a set of data instances rather than individual instances. This set of data instances is called an

[1] Authors: Shagun Jhaver, Swarup Chandra, Khaled Al-Naami, Latifur Khan and Charu Aggarwal

entity. Classification of an entity can be performed by inferring the distributional pattern of its corresponding data instances in the feature space. This classification problem is called *Setwise Classification* as the label prediction is performed for sets of data instances. In this chapter, we focus on the problem of *Setwise Stream Classification* where the data instances occur continuously in a data stream.

A setwise classification problem can be realized in many real-world situations. For instance, a *Website Fingerprinting attack* can be considered as a setwise stream classification problem. In Website Fingerprinting (Liberatore and Levine, 2006; Dyer et al., 2012; Juarez et al., 2014), the task of an attacker is to identify the webpage accessed by a user, using only the encrypted network trace captured by a man-in-the-middle, and without using the source and destination IP (as this may not always be the address hosting the webpage). A successful attack is a breach of user privacy and security. Therefore, it is important to study different attack schemes and provide countermeasures to ensure the safety of users accessing the web. Here, a network trace contains a set of packets (Burst) with encrypted payload. Each burst can be viewed as a data instance having features such as byte length and time. A set of packets exchanged (uplink and downlink) between the user and server for loading a complete webpage from a trace forms an entity. Each entity is associated with a webpage name forming a class label. Here, a single packet may not be associated with a class label since similar packets can occur over multiple webpages. Instead, we associate a class label to a trace.

Similarly, consider the problem of predicting demographic, or location information of Twitter users', using the content of their tweets. Such a prediction technique can be used for targeted advertising, and in recommendation systems. Here, the data stream consists of user tweets where each tweet can be considered as an individual data instance. The features of each data instance may consist of user details, type of tweet, words used, hash tags used, etc. Here, the userID associated with each tweet can be considered as an entity, representing the

user. Each user has an associated demographic or location that forms a class label. Feature distribution for each user based on his/her tweets can be used to determine the class label.

Designing a stochastic model to represent a non-static data generation process presents unique challenges. In addition, the assumptions of setwise stream add to these challenges so that the traditional stream classifiers cannot be employed. Data instances associated with each entity can occur at different times along the stream. Further, each class has multiple entities associated with it.

A robust classification method should be able to model characteristic behaviors of the entities. Therefore, the challenges in designing a setwise stream classifier are as follows:

1. Insufficient Statistics: As data instances belonging to various entities are interleaved, their distribution characteristics need to attain sufficient statistics to be used for classification.

2. Concept Drift (Klinkenberg, 2003): Data distribution may change over time with newer streaming data.

3. Concept Evolution (Masud et al., 2010): New class labels may appear later in the stream.

4. Storage Limitation: It is impossible to store and use all historical data for mining in memory when the data stream is assumed to be unbounded.

In this chapter, we propose a new approach to perform setwise stream classification by designing an ensemble of models to address the above mentioned challenges. In particular, the characteristics of an entity can be represented by the spatial distribution of data instances in its d-dimensional feature space, denoted by $\mathcal{D}$. This distribution is called the entity's *fingerprint*. We utilize the k-means clustering method to obtain k clusters of data instances. For each entity, the fingerprint is defined as the distribution of its data instances within

these k clusters. We now view an entity as a k-dimensional vector, where each element denotes the fraction of data instances associated with the corresponding cluster. This view of the setwise classification problem projects an entity-fingerprint into a data point in the k-dimensional space, denoted by $\mathcal{K}$. Class label is now predicted for these data points in the $\mathcal{K}$ space. An ensemble of models is generated along the data stream to capture concept drift and concept evolution of entities in the $\mathcal{K}$ spaces. In addition, only the statistical summary of data instances is stored to address the challenge of unbounded stream.

The contributions of the chapter are as follows:

1. We present a new framework to perform setwise stream classification. We view the distribution of a set of data instances belonging to an entity in its d-dimensional feature space, as a projection onto a k-dimensional space where the entity is represented as a data point. The prediction of class labels is then performed in this k-dimensional space using a suitable classifier.

2. Our approach overcomes challenges such as concept drift, concept evolution, and storage limitations, with respect to the entities in the k-dimensional space, using an ensemble of models.

3. We evaluate our proposed approach with real-world and synthetic datasets, using various stochastic classification algorithms, and compare their accuracies for each dataset.

The chapter is organized as follows. We first summarize related studies in Section 2.2. We then describe our proposed approach in Section 2.3 and provide a detailed algorithm. Evaluation of this approach is provided in Section 2.4 including the dataset description and results. Discussion of various observations and future work is provided in Section 2.5. Finally, we conclude our chapter in Section 2.6.

2.2 Related Work

In this section, we survey related studies on stream data classification, including a recent work on setwise stream classification.

A data stream is an ordered set of data instances obtained continuously and periodically from a data generation process. Classification of this streaming data is typically performed on individual instances, where a class label is estimated, as they arrive. This is done by either using a single model (Glymour et al., 1997) or an ensemble of models (Wang et al., 2003), which is trained using a set of training data instances in a supervised manner. Some approaches use incremental learning methods (Syed et al., 1999) where the classifier first estimates a class label for a test data instance, and then is retrained before further classification is performed (Gaber et al., 2005). However, in case of setwise classification, classifying each instance, directly from the stream, is not possible. These existing methods are therefore not suitable for classification of entities.

A recent study (Aggarwal, 2014) introduces an approach to perform setwise stream classification by designing a clustering methodology (Aggarwal, 2012) to extract distribution characteristics of data instances in its feature space. Using data instances belonging to a set of training entities, this approach performs k-means clustering on an initial set of data instances, from multiple training entities, to obtain k cluster-centroids (or *anchor points*). Each data instance (both training and test), occurring later in the data stream, is then associated with a cluster having the centroid closest to it. Distribution of data instances among these clusters represents an entity, which is used to further represent a class distribution. Class labels of test entities are predicted by checking the class label of the closest (or nearest) training distribution. Further, using an initial set of anchor points throughout the classification process, an attempt is made to capture drifting concepts by considering it as a different distribution. However, this may not capture such drifts effectively. In addition, if all class labels are not known a priori, the initial set of anchor points will not be able to

capture new classes appearing at a later stage in the stream since class profiles for these labels would not be created.

In our approach, we provide a generic framework to perform setwise stream classification. Our method not only performs classification using the nearest neighbor algorithm, but also using other stochastic algorithms such as decision trees, SVMs etc. In addition, we also address the challenges of concept drift and concept evolution by constructing an ensemble of models having anchor points constructed at regular intervals while processing the data stream, to represent the changing data distribution. We selectively include anchor points to build models along the stream and capture the distributional changes or new distributions in incoming data.

2.3 Setwise Stream Classification

In this section, we present our approach to perform setwise classification, and describe how we use the ideas of ensemble methods in machine learning to classify test entities in a data stream.

2.3.1 Stream

While performing setwise stream classification, we assume a finite set of entities in the stream. An unbounded number of data instances may however exist. The input data stream consists of data instances with dimensionality d, each associated with an entity $\mathcal{E}_i$ where $1 \leq i \leq N$, and N is the total number of entities. An entity is associated with a class label $l \in \{1 \ldots L\}$. It is also assumed that $L \ll N$ so that there exists multiple entities belonging to the same class. The dataset consists of training and test entities where the classification problem is to determine class labels for test entities using a model trained with the training entities. Data instances are assumed to continually arrive in a streaming fashion, and the data instances of test and training entities are interleaved in the stream. We process a set of data instances,

occurring sequentially in the stream at a time. This set of data instances is called a *Block*. Each block can have data instances belonging to training and test entities.

A data instance consists of d feature values along with an entity identifier, and a class label. This is denoted as $< Y_r, \mathcal{E}_r, label_r >$ where r represents a data instance received at time t_r. Y_r is a d-dimensional tuple with each dimension representing a feature of a data instance. $\mathcal{E}_r$ is an entity identifier denoting the entity to which the data instance Y_r belongs. The class label denoted as $label_r$ represents the class of the entity. $label_r \in \{1 \dots L\}$ in case of a training entity, else $label_r = -1$ for a test entity.

2.3.2 Vector Representation

Data streams often deliver elements very rapidly, and the processing needs to be done in real time, or we lose the opportunity to process at all, without accessing the archival storage. Therefore, it is crucial that the stream mining algorithm we use executes in main memory, without access to secondary storage or with only rare accesses to secondary storage. A number of data-based and task-based techniques have been proposed to meet the challenges of data stream mining (Dietterich, 2000). Data-based techniques entail summarizing the whole data-set or choosing a subset of the incoming stream to be analyzed. The setwise classification task cannot assign a class label to each instance in the data stream. Since the class label is associated with a set of data instances belonging to an entity, we require an entity to be represented as a data point in the $\mathcal{K}$ space. Such a representation would facilitate the use of various well-known classification algorithms in estimating class labels for the entity.

A simple statistic, such as sum or average, that combines the d-dimensional feature values of each data instance belonging to an entity would lose the spatial distribution information in its feature space. We therefore use a histogram-like data structure that summarizes the incoming data streams. In order to construct such a data structure, we perform k-means

clustering, using an initial set of random data instances belonging to various training entities in each block considered, to estimate a set of k centroids, called *anchor points*. These anchor points are used for constructing k clusters consisting of data instances from the stream. Each new data instance in subsequent blocks is assigned to the closest cluster according to its Euclidean distance in the $\mathcal{D}$ space. The anchor points, once built, remain fixed throughout the rest of the streaming process. A histogram-like data structure is constructed for each entity, known as its *fingerprint*, which is defined as follows:

> A set of r data instances belonging to an entity $\mathcal{E}$ are partitioned into k clusters (denoted as $C_1 \dots C_k$, where $k << r$) whose centroids or anchor points are given as $a_1 \dots a_k$. Each data instance is assigned to a cluster with the centroid closest to it. The *fingerprint* of $\mathcal{E}$ is a k-dimensional tuple $[f_1, f_2, ..., f_k]$. Each f_i, with $i \in \{1 \dots k\}$ is equal to the cluster frequency f_i of cluster C_i. Here, $\sum_{i=1}^{k} f_i = 1$.

Fingerprints are thus defined with respect to a set of k anchor points. Since the streaming data is received continuously, data instances of different entities may appear randomly. Now, a fingerprint of an entity $\mathcal{E}$ can be seen as a data point in $\mathcal{K}$ space. This provides a set of training and testing data points in $\mathcal{K}$ space, each corresponding to the training and testing entity fingerprints. The classification problem now translates to predicting the class labels of the test data points, using the training data points to train a classifier.

The vector representation is illustrated in Figure 2.1 as an example. Streaming data instances are grouped into 2 blocks. $Block_i$ is first considered for which 4 anchor points (a_i^1 to a_i^4) are generated. Fingerprint for an entity $\mathcal{E}$ is computed by counting the number of associated data instances for each cluster in $Block_i$. The data instances of $\mathcal{E}$ are represented as shaded circles in the data stream. The 4-dimensional vector is then projected as a data point in $\mathcal{K}_i$ space. The process is repeated for $Block_{i+1}$, using new sets of anchor points (a_{i+1}^1 to a_{i+1}^4) from this block. Therefore, a new $\mathcal{K}_{i+1}$ space is formed with fingerprint of $\mathcal{E}$ projected onto it as a data point. Further, new data instances of $\mathcal{E}$ appearing in $Block_{i+1}$ are also clustered using the anchor points of $Block_i$. The fingerprint of $\mathcal{E}$ is updated to include these data instances.

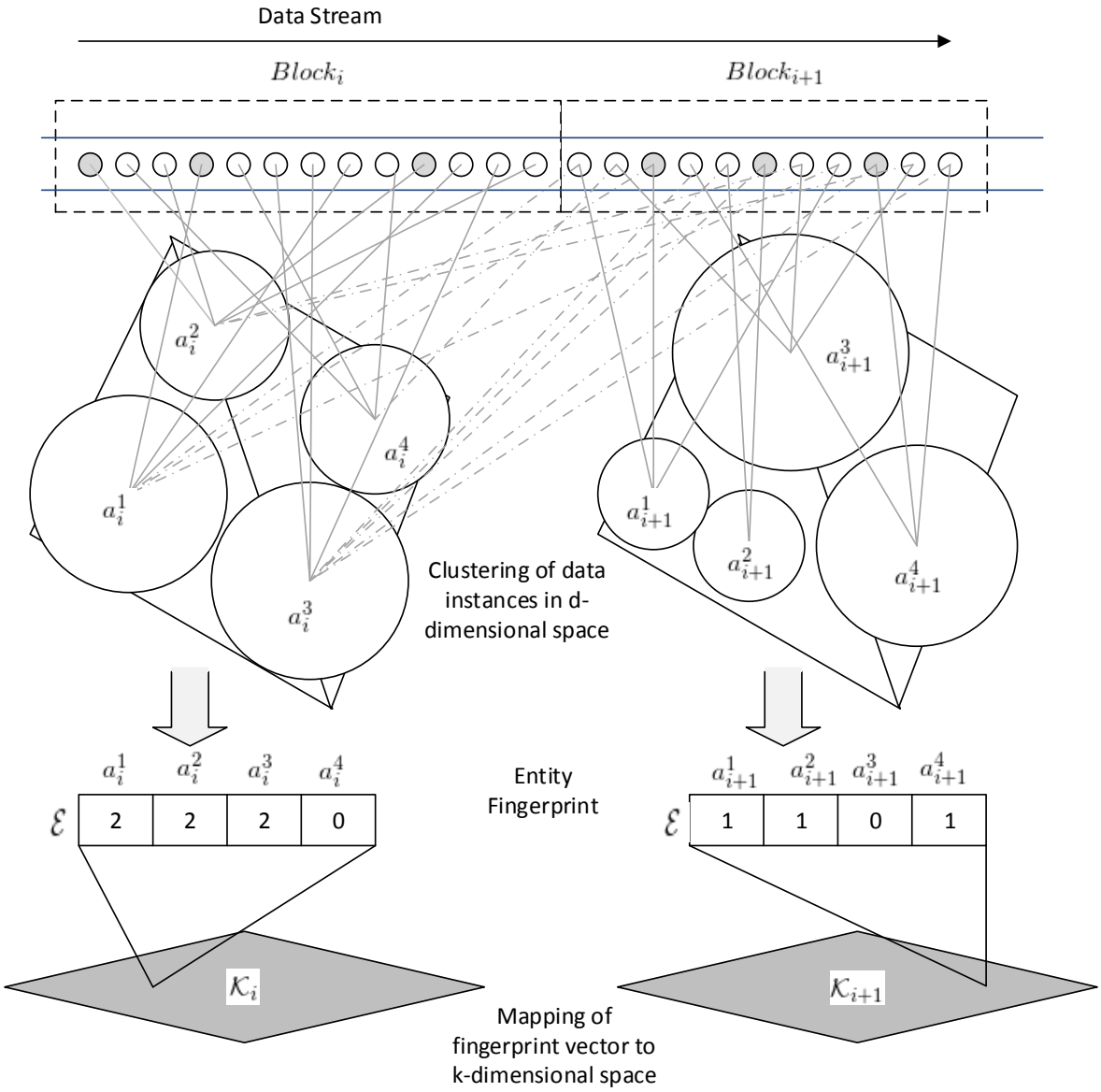


Figure 2.1. An example constructing an entity fingerprint using its d-dimensional data instances which are shaded. Here, the fingerprints of entity $\mathcal{E}$ are 4-dimensional vectors constructed using 4 anchor points from corresponding blocks. The fingerprints are represented as points in the k-dimensional spaces $\mathcal{K}_i$ and $\mathcal{K}_{i+1}$ respectively.

The previous study (Aggarwal, 2014) provides a classification algorithm which aggregates similar distributions of the same class. In addition to entity fingerprint, each entity also

involves another type of data structure called *class profile*. A class distribution or *class profile* is constructed from a set of entity fingerprints belonging to the same class. It is defined as follows:

> Given a set S_c of k-dimensional fingerprints of the same class, a *class profile* c is represented by a $(k+2)$-tuple $< AG(S_c), |S_c|, label_c >$ where $AG(\cdot)$ is a k-dimensional vector constructed by summing corresponding dimension values in S_c, $|S_c|$ denotes the number of fingerprints in S_c, and $label_c$ is its associated class label.

Since a class may have multiple distribution patterns, multiple class profiles are created for each class label, where each profile is constructed using a subset of fingerprints associated with *label*. Each fingerprint is used in the construction of a unique class profile. A class label prediction of a test entity is performed by assigning a class label of a class profile with the least cosine distance to the test entity fingerprint. In $\mathcal{K}$ space, this algorithm can be viewed as clustering entity data points belonging to the same class, and then performing 1-nearest-neighbor (1NN) to predict the class of a test entity. The class profile ($Profile_j$) construction represents clustering of S_c training entity data points of the same class. Therefore, $Profile_j$ is yet another k-dimensional data point. A test entity data point is then assigned the class label of the nearest class profile point.

2.3.3 Ensemble Model

We now describe an ensemble based classification model to perform setwise stream classification.

Traditional ensemble classifiers are learning methods that construct a set of classifiers and then classify new data points by taking a weighted vote of their predictions (Kolter and Maloof, 2007). A necessary and sufficient condition for an ensemble of classifiers to be more accurate than any of its individual members is if the classifiers are accurate and diverse (Hansen and Salamon, 1990). Classification in a setwise data stream is more challenging than classifying individual data instances. An entity fingerprint needs to be updated

as new data instances arrive in the data stream. However, a limitation in using entities for training or testing classifiers may be that its fingerprint does not have sufficient statistical information. In order to overcome this situation, we use a method similar to the one used in (Aggarwal, 2014), where an entity fingerprint is not considered for training a classifier, or for testing, until enough data instances belonging to the entity are seen.

The setwise classification is a two phase process involving vector representation of fingerprints, and classification of entities in the $\mathcal{K}$ space. Anchor points generated from each block construct a $\mathcal{K}$ space in which entity classification can be performed. We use this k-dimensional space to define a *Model* as follows.

> Given a set of k-dimensional anchor points constructed from a block b, a *model* $\mathcal{M}_b$ is a $(k + N + 1)$ tuple $< \mathbf{A}, \mathbf{E}, z >$, where $\mathbf{A}$ is a set of k anchor points, $\mathbf{E}$ is a set of N entity-fingerprints which are projected on the $\mathcal{K}_b$ space, and z is the accuracy of the classifier trained on the training entities to predict the class labels for test entities in $\mathcal{K}_b$ space.

We use a subscript to a model to denote the block from which its elements were constructed. For instance, if the ensemble $\mathcal{Q}$ equals $\{\mathcal{M}_1, \mathcal{M}_3, \mathcal{M}_4\}$, then the anchor points of $\mathcal{M}_1$ is obtained using the 1^{st} block of data instances. Similarly, anchor points of $\mathcal{M}_3$ is sampled from the 3^{rd} block, and that of $\mathcal{M}_4$ from the 4^{th} block. In this case, the ensemble $\mathcal{Q}$ contains three models. This approach forms the ensemble-based setwise classification.

Ensemble-based Setwise Stream Classification (ESSC)

Anchor points of a model define the distribution of data instances in the $\mathcal{K}$ space. This affects the distribution of fingerprints projected in this space. In addition, the fingerprint update process changes the location of data points in $\mathcal{K}$ space. Projection of new entities onto the $\mathcal{K}$ space, as the stream progresses, may not effectively capture its distribution since the anchor points are only constructed using data instances from a single block. This is especially true if entities of a new class are introduced after the model attains sufficient statistics. To meet this set of challenges, our proposed approach, called *Ensemble-based*

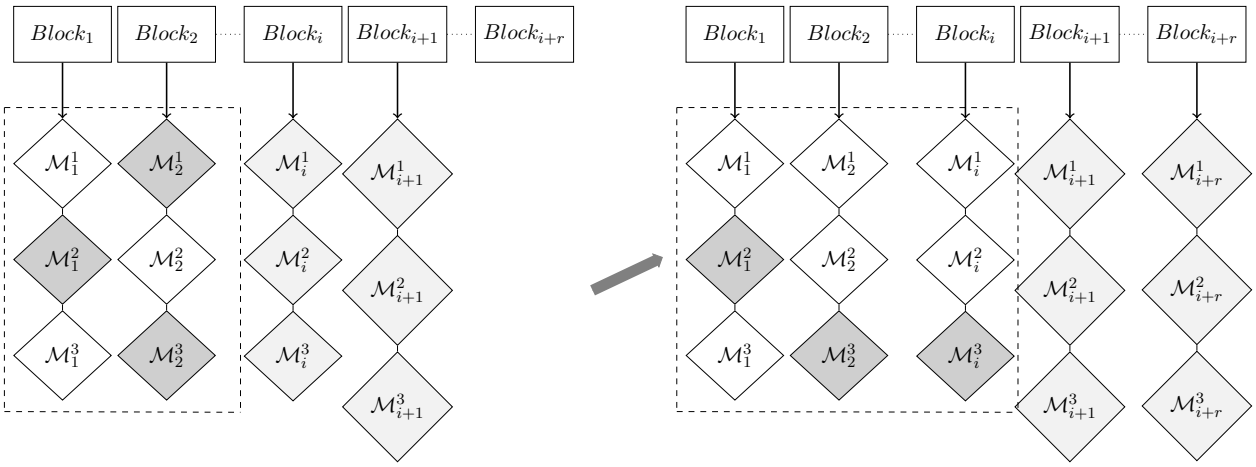


Figure 2.2. Ensemble update procedure on reading $Block_{i+r}$ at which $Block_i$ attains sufficient statistics. Here, $v = 3$ models are generated per block. $\mathcal{M}_b^v$ represents a model constructed with anchor points constructed using a random set of training data instances in $Block_b$. Models in the ensemble are represented with dark shades, and the models which hasn't attained sufficient statistics are lightly shaded. Models having no shade are removed or forgotten.

Setwise Stream Classification (ESSC), maintains a fixed-size ensemble of best performing models, while at the same time constructs new models from the most recent data instances in the stream. The performance is measured by the accuracy of a classifier predicting class labels of test entities within a model.

Similar to individual entities, a model created from a block also needs to attain sufficient statistics before a classifier can be trained and tested on entity data points projected on the model. In order to address this issue, we require a model $\mathcal{M}_b$, created at block b, to update its statistics for r blocks further in the stream before it is considered to be included in the ensemble. Each data instance encountered upto $b+r$ blocks is used to update the fingerprint of its entity. After the fledgling models are updated from the b^{th} block to $(b+r)^{th}$ block, a classifier is trained using the updated training entities and is tested on the updated testing entities, in the $\mathcal{K}_b$ space. The accuracy of this classifier is then associated with z of $\mathcal{M}_b$. Such a classification is performed at the end of each block, following the $(b+r)^{th}$ block for $\mathcal{M}_b$. Figure 2.2 depicts an example of this ensemble update and classification procedure when a

new block is processed. As illustrated, the models in the ensemble (darkly shaded regions) are the best chosen from $Block_1$ to $Block_{i-1}$, prior to processing $Block_{i+r}$. Since we assume that a model attains sufficient statistics after r blocks from its creation, the models created from $Block_i$ through $Block_{i+r-1}$ are buffered in a set $\mathbb{M}$. These are illustrated as lightly shaded regions in the figure. After processing $Block_{i+r}$, $Block_i$ is assumed to attain sufficient statistics. Label predictions of test entities are carried out using the corresponding models $\mathcal{M}_i^1$ to $\mathcal{M}_i^3$. The ensemble is updated with the best κ performing models, choosing from the sets $\mathcal{Q} = \{\mathcal{M}_1^2, \mathcal{M}_2^1, \mathcal{M}_2^3, \ldots\}$ and $\{\mathcal{M}_i^1, \mathcal{M}_i^2, \mathcal{M}_i^3\}$. This process may replace existing models in the ensemble with newer models having better accuracy. In this example, $\mathcal{M}_i^3$ replaces $\mathcal{M}_2^1$ in $\mathcal{Q}$ after processing $Block_{i+r}$. A fixed-size ensemble ensures that the models do not get outdated soon, addressing concept drift and concept evolution in data streams.

Algorithm 1 details the setwise stream classification process. A priority queue $\mathcal{Q}$ is used to maintain the top κ performing models in the ensemble. A set $\mathbb{M}$ is used to maintain models that have not yet achieved sufficient statistics. A set of data instances is initialized in the *initializeBlock* procedure to form a block b. A set of v models $\mathcal{M}_b^i$ $(0 < i < v)$ is created by constructing k anchor points (*anchorPoints*) from randomly selected training data (obtained using the *getTrainingData* procedure), and each of these are added to $\mathbb{M}$. At every iteration, the models in the ensemble $\mathcal{Q}$ are updated with data instances in b. The *updateFingerPrint* procedure associates anchor points to data instances in b, with respect to the $\mathcal{K}_b$ space, and updates entity-fingerprint statistics for both training and test entities. The *predictLabel* procedure trains a stochastic classifier using the training entities encountered so far, predicting the class label for each test entity to provide a classification accuracy. The model's accuracy is updated with the resulting classification accuracy using the *updateAccuracy* procedure. This is used to compute the top κ models in the priority queue after processing each block. Note that the test and training entities used for this update are only those encountered so far in the stream, and having sufficient statistics.

Models having insufficient statistics are buffered in $\mathbb{M}$ for r iterations. After processing the next r blocks, the class label for its test entities are predicted, and the model is added to the priority queue with the resulting prediction accuracy. Once added in the ensemble, the model is removed from $\mathbb{M}$. Finally, the least accurate model from the ensemble is iteratively removed till the ensemble attains size κ.

In case of using class profiles for performing classification in the *predictLabel* procedure, as mentioned in section 2.3.2, we use a fixed number of class profiles throughout the stream. Training entities in each model are used to update its class profiles. We inspect all class profiles with the same label, and find the profile whose average AG has the smallest cosine distance (Lee, 1999) from the entity's fingerprint. This may change the association of entities with a class profile since entity-fingerprints are modified due to newer data instances arriving in the stream. Subsequently, the class profiles, in turn, also update their average fingerprints.

2.4 Evaluation

In this section, we describe the datasets used to evaluate the proposed approach and present our experimental results.

2.4.1 Datasets

We use a set of real-world and synthetic datasets to evaluate our approach. These are described as follows.

1. *Hub64* dataset consists of voice signals of 64 different personalities converted to an 8-dimensional GPCC format. Each data instance is a microsecond speech sample with features such as pitch. The classification problem is to predict the speaker based on a given set of speech samples. In this case, if data instances from two different speakers are considered, the feature values of these instances may not have distinguishable

patterns (e.g. silence between sentences in a speech). A distribution of a set of data instances may exhibit certain characteristic of the class label. Therefore, a setwise classification method is appropriate. In this dataset, each speaker data is divided into 10 sets of data instances, each representing an entity. Therefore, the dataset has a total of 640 entities with 394448 data instances.

2. *ForestCover* dataset is publicly available and has a total of 54 features. The classification problem is to predict the cover type of a data instance, given corresponding feature vector. However, the dataset can be converted to a form appropriate for setwise classification by discretizing the elevation feature to form a finite set of entities. This discretization into 9 points is performed using its mean and standard deviation. Each of the 10 cover-type classes is divided into 150 entities, creating a total of 1500 entities in a stream of 578029 normalized data instances.

3. *Synthetic* dataset is generated from Gaussian mixture models with 50 classes from 10 different overlapping clusters. Data points generated from each of these 50 mixture model distributions were divided into 20 entities, forming a total of 1000 entities and having 997091 data points.

These datasets are the same as used by Aggarwal (Aggarwal, 2014). Using these datasets, we compare the evaluation of their method (baseline) with our proposed approach. Further, we perform pre-processing so that data instances belonging to testing and training entities are randomly mixed to form a data stream, within each dataset. We choose the training and test entities randomly, in the ratio of 80% and 20% respectively.

In addition to the above datasets, we evaluate our approach with two other scenarios having the structure necessary for performing setwise classification, i.e., the association of class labels to entities rather than individual data instances. These are the areas of *Website Fingerprinting* and *Social Network User-Location Estimation*. We gather datasets to perform these two tasks, and evaluate our approach. We now present the details of these datasets.

Website Fingerprinting

Website Fingerprinting (WF) is a Traffic Analysis (TA) attack that threatens web navigation privacy. Users accessing certain webpages may wish to protect their identity and may use anonymous communication mechanisms to hide the content and metadata exchanged between the browser and servers hosting the webpage. This can be performed using popular methods such as Tor network (Syverson et al., 1997). A malicious attacker wishing to identity a webpage accessed by the user, captures the network packets by eavesdropping. Although packet padding, content and address encryption conceal the identity of webpages visited by a user, adversary, as a passive attacker, can still extract useful information from padded and encrypted packets using various machine learning techniques. Such WF attacks aim to target individuals, businesses and governments.

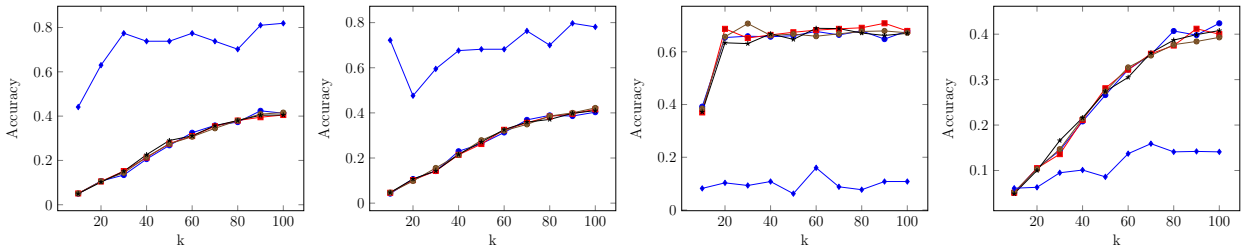
For the purpose of this study, we extract features from the Liberatore and Levine dataset (Liberatore and Levine, 2006) which has been widely used for website fingerprinting attacks research. The data set is a collection of traces for 2000 webpages spanning a two-month period. Each webpage has different traces and each trace consists of uplink and downlink packets generated when loading the webpage. Each packet contains information like time and length in bytes. A data instance was formed by combining a group of consecutive packets in a specific direction, which forms a burst B. For example, consider the following packets generated by a trace: ($P1$, $\uparrow$), ($P2$, $\uparrow$), ($P3$, $\downarrow$), ($P4$, $\downarrow$), ($P5$, $\downarrow$), ($P6$, $\uparrow$), ($P7$, $\uparrow$). Here, PX denotes packet number X, $\uparrow$ denotes an uplink packet, and $\downarrow$ denotes a downlink packet. This has three bursts: $B1$ is formed by ($P1$, $\uparrow$) and ($P2$, $\uparrow$) of uplink packets, $B2$ by ($P3$, $\downarrow$), ($P4$, $\downarrow$), and ($P5$, $\downarrow$) of downlink packets, and $B3$ by ($P6$, $\uparrow$), ($P7$, $\uparrow$) of uplink packets.

Instead of considering each trace to be a data instance, like in earlier studies (Dyer et al., 2012; Juarez et al., 2014), we consider each burst as a data instance in the data stream. Each burst is a 4-dimensional feature vector with total burst time, total burst length in uplink,

total burst length in downlink, and trace identifier as dimensions of the vector. Here, time corresponds to the total time of all packets in the burst, and total burst length corresponds to the total byte-length of packets in the burst. Each trace is considered as a unique entity, which can be associated with a webpage label. As each webpage has multiple traces and each trace is a unique entity, each webpage will have multiple entities, none of which is shared with any other webpage. MTU packet padding countermeasure, which is considered as one of the effective protective techniques against WF (Dyer et al., 2012) , has been used. With MTU padding, each packet will have the same length of MTU = 1500 bytes. A total of 128 webpages are considered for the setwise stream classification, with 16 traces of each webpage as training entities, and 4 traces as test entities. The dataset contains a total of 228252 bursts.

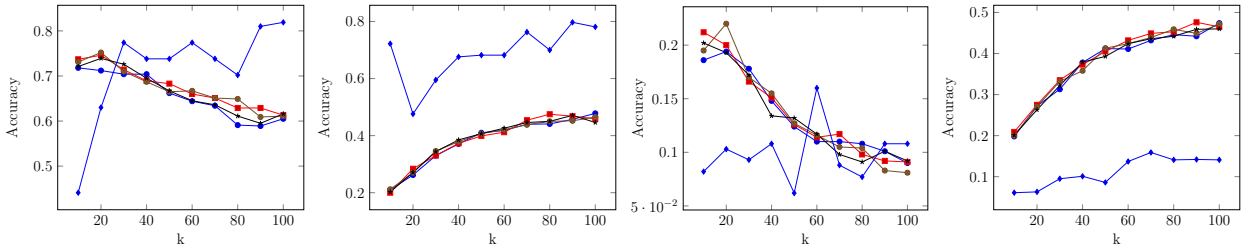
Social Network User-Location Prediction

Users in a social network such as Twitter can mention their physical location in their profile. However, this is typically a manual process, and is error prone (Chandra et al., 2011). Therefore, prediction of user location based on the content of their messages has a huge potential to impact location based services. For this study, a synthetic social network data-set containing 50 locations, that are representative of class labels is prepared. For each location, we constructed 50 users, which are modeled as entities in our framework. Each user has at least 50 short messages. Each of these short messages is modeled as a data point. The messages of different users are shuffled together when simulating the stream. As with other data-sets, 80% of users are training entities and 20% users are test entities. Each message has at least one location-related keyword. These keywords are selected from a dictionary constructed using the open-source FIPS code datasets. For example, a user with location 'Texas' may have location related keywords like 'Richardson', 'Plano', 'Dealey plaza', etc in her messages. The dataset was designed such that each message of any user u has keywords associated with a location other than u's location with a probability of 0.2.



(a) Hub64 (b) ForestCover (c) Synthetic (d) Website

Figure 2.3. Accuracy vs k on datasets using SMO classifier, with number of blocks used to generate models : 1^{st} block; First 3 blocks; First 5 blocks; First 7 blocks; Baseline accuracy.



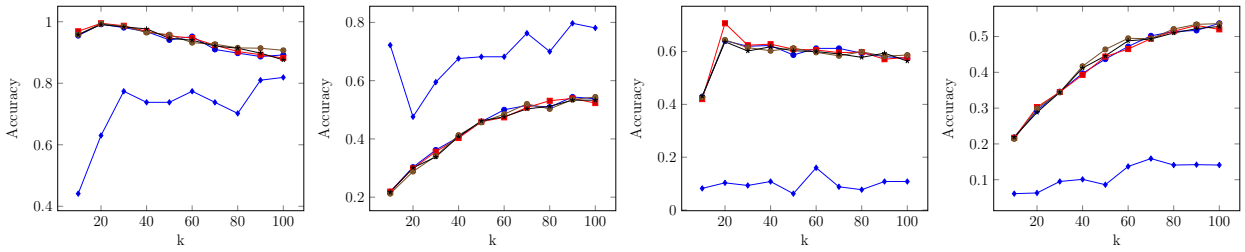
(a) Hub64 (b) ForestCover (c) Synthetic (d) Website

Figure 2.4. Accuracy vs k on datasets using J48 classifier, with number of blocks used to generate models : 1^{st} block; First 3 blocks; First 5 blocks; First 7 blocks; Baseline accuracy.

The dataset contains a total of 127500 short messages. The only features used for this are the location-related keywords appearing in user-messages. This resulted in a sparse dataset containing data-points with large number of features, but very few features with non-zero values. We modified our framework to handle this data using high-dimensional clustering. We perform subspace clustering and use iterative refinement (Vidal, 2010) to improve the quality of the clusters. A mapping of each feature f to the clusters having non-zero value for f is maintained for an efficient assignment of every data-point to its closest cluster. The fingerprint update and ensemble selection follow as usual.

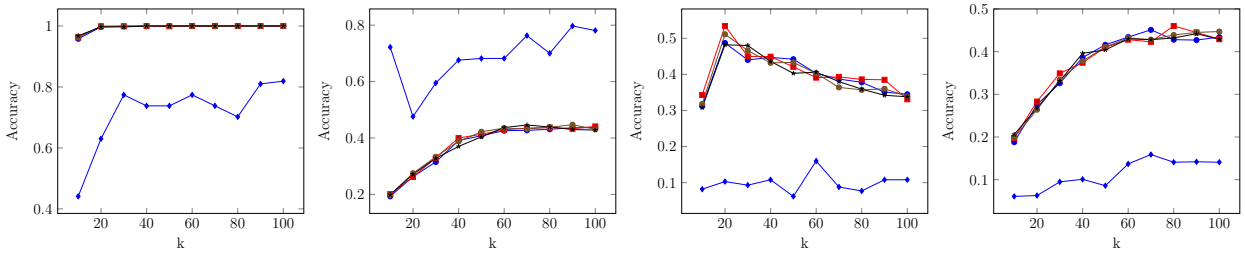
2.4.2 Experiments and Results

We now present our evaluation of the proposed approach.



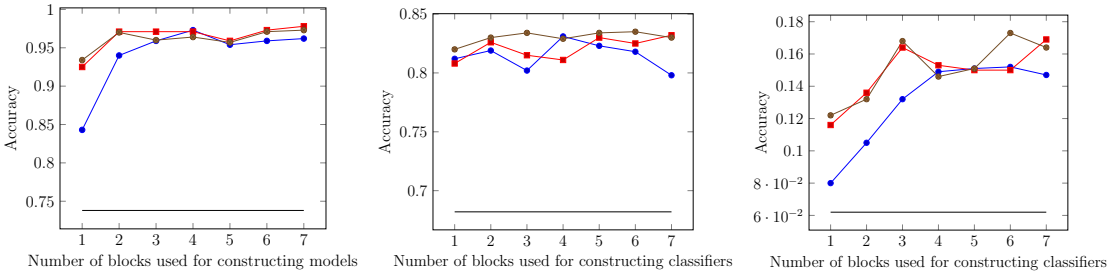
(a) Hub64 (b) ForestCover (c) Synthetic (d) Website

Figure 2.5. Accuracy vs k on datasets using NaiveBayes classifier, with number of blocks used to generate models : 1^{st} block; First 3 blocks; First 5 blocks; First 7 blocks; Baseline accuracy.



(a) Hub64 (b) ForestCover (c) Synthetic (d) Website

Figure 2.6. Accuracy vs k on datasets using 5NN classifier, with number of blocks used to generate models : 1^{st} block; First 3 blocks; First 5 blocks; First 7 blocks; Baseline accuracy.



(a) Hub64 (b) ForestCover (c) Synthetic

Figure 2.7. Accuracy of datasets using EnsembleCP with 2 models per block; 5 models per block; 10 models per block; and Baseline accuracy.

Choice of Parameters

As evident from section 2.3, ESSC algorithm has multiple parameters that can be tuned for achieving good classification performance. These parameters include the choice of classifiers,

number of anchor points in a model, number of models in the ensemble, number of models that can be constructed per block, number of blocks that can be used to construct the models along the stream, number of blocks before a model is assumed to attain sufficient statistics, minimum number of data instances needed to achieve sufficient statistics for an entity, and the number of data instances in each block.

We systematically perform various experiments on all datasets to find the optimum values for these parameters. Since the number of parameters is large, we present the results of these experiments by varying most of the parameters, which we empirically found to have a significant effect on the classification accuracy. This includes the choice of classifiers, number of anchor points in a model, and number of blocks that can be used to construct the models along the stream. We choose to use the standard Weka (Hall et al., 2009) implementation of classifiers such as *NaiveBayes*, *SMO*, *J48* and *5NN* to perform classification. We also experiment on the classifier constructed using class profiles. We refer to this classifier as *EnsembleCP*. Finally, we refer to the EnsembleCP classifier with a single set of anchor points and class profiles as *Baseline* since this is the approach presented in a previous study (Aggarwal, 2014). For parameters such as number of anchor points in a model (k), and number of blocks that can be used to construct the models along the stream, we perform a grid search over a finite discrete domain. However, we assume a specific set of values for other parameters for each of the experiments. We set the number of models in the ensemble to 10, number of classifiers per block to 5, and number of blocks before a model attains sufficient statistics to 3. These choices are based on extensive experiments, and to keep these values consistent across all experiments for each dataset. The minimum number of data instances needed for an entity to achieve sufficient statistics was set to 10 since for one of the datasets, the number of data instances available for an entity was 10. In the case of EnsembleCP and the baseline approach, we consider a fixed set of class profiles. We set the number of class profiles to 80 for Hub64, ForestCover and Synthetic datasets. For Website and Social

Network datasets, we set the maximum number of class profiles to 400 to ensure creation of at least two class profiles per class. Finally, we divide each dataset into 10 blocks for processing the data stream.

Setup

We performed various experiments to compare the accuracy of the Baseline classifier used in a single model with other classifiers used in the ensemble of models. In order to show the effectiveness of the ensemble model, especially for concept evolution, we performed an experiment where entities belonging to a new class were introduced into the data stream at a later stage.

The experiments were conducted on a Linux machine running with four Intel® Core™2 Quad 2.5 GHz processors and 7.7 GiB of memory. We now present the results of these experiments.

Results

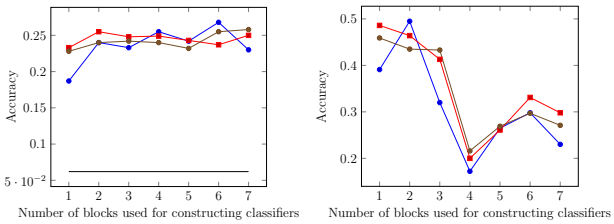
Accuracies obtained on the Hub64, ForestCover, Synthetic and Website datasets using various classifiers, and by varying the number of anchor points (k) in the range of 10 to 100, with an increment of 10, are shown in Figures 2.3 to 2.6. These figures also show the accuracy when using the baseline approach for comparison purpose. Average classification accuracies obtained across all blocks in the stream by different models in the ensemble are reported. We only show the results of these experiments on four datasets due to space constraints. The figures show that the ensemble based approach performs better than the baseline approach in most cases, and the performance improves with increase in k. For instance, accuracy of the Website dataset performs significantly better than the baseline approach on all classifiers, with the highest accuracy of 53.6% obtained when $k = 100$ using NaiveBayes classifier as

shown in Figure 2.5d. Also, it can be observed that the value of k significantly affects the classification accuracy.

In the case of Hub64 dataset, Figures 2.4a and 2.3a show that the baseline approach outperforms the ensemble method as k increases. As Hub64 data instances are samples of speeches, the data instances are close to each other in its feature space. Therefore, a decision tree (J48) or an SVM (SMO) may not be able to evaluate effective class boundaries. However, a nearest neighbor algorithm would appropriately capture such boundaries. This can be seen in Figure 2.6a where the ensemble model using 5NN outperforms the Baseline approach, and provides an accuracy of 100% for k beyond 30.

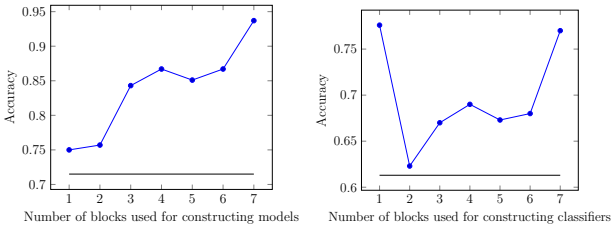
In case of the ForestCover dataset, the baseline approach outperforms the ensemble based models using SMO, J48, NaiveBayes and 5NN classifiers. This shows that the baseline classifier represents the underlying data distribution better than these other classifiers. However, a comparison of accuracies from an ensemble of models using the same type of classifier as the baseline, in Figure 2.7b, shows that the ensemble method performs significantly better than the baseline method.

Figures 2.3 to 2.6 also show the accuracies obtained when varying the number of blocks used to build a set of models in the ensemble method. For instance, Figure 2.3c shows an accuracy of 64.8% with $k = 90$ when using only the first block to generate models for the ensemble. Similarly, an accuracy of 70.8% is obtained when using the first 3 blocks, 67.9% when using the first 5 blocks, and 66.2% when using the first 7 blocks. This shows that the accuracy of the model is not significantly affected when using multiple sets of anchor points along the stream. The accuracy of each model is calculated using the test entities encountered by this model in the stream. Since models created in early stages may have encountered more training and test data instances, the accuracy may not vary with the number of blocks used to build the model. This behavior can also be observed when using the EnsembleCP method in Figures 2.7 and 2.8.



(a) Website (b) Social Network

Figure 2.8. Accuracy of datasets using EnsembleCP with 2 models per block; 5 models per block; 10 models per block; and Baseline accuracy.



(a) Hub64 (b) ForestCover

Figure 2.9. Accuracy with concept evolution. EnsembleCP; and Baseline.

Figures 2.7 and 2.8 show the accuracy obtained from the EnsembleCP classifier with all datasets, with different number of models created per block, and number of blocks used to construct these models. For instance, Figure 2.7a shows that an accuracy of 97.1% was obtained using the EnsembleCP method on the Hub64 dataset when using 5 models per block, and when the first 4 blocks are used to generate new anchor points and class profiles. These results show that more number of models per block increases the accuracy of the model. The small variations in accuracy with increase in the number of blocks used to construct the models, is due to the random data instances considered while computing corresponding anchor points. The experiments were performed using $k = 50$. Also, the figures show that the ensemble method outperforms the baseline approach when using similar classification technique.

Finally, Figure 2.9 shows the accuracy obtained when concept evolution is induced on Hub64 and ForestCover datasets, with EnsembleCP classifier. The datasets are rearranged to ensure concept evolution by introducing data instances associated with entities belonging to

5% of classes in the second half of the data stream. For the EnsembleCP method, we create 10 models per block. The results show that EnsembleCP method performs significantly better compared to the baseline method, on both these datasets, even with concept evolution.

2.5 Discussion

The distribution of data instances in a dataset can have a variety of class boundaries. Various machine learning algorithms for data classification have been developed to model these class boundaries. We leverage such classifiers in ESSC to perform setwise stream classification. When new data instances arrive, the corresponding entity fingerprint is updated. This update is performed for both testing and training entities. Therefore, we employ batch training and testing of classifiers to predict class labels of test entities. A new classifier is trained using the training entities, in the corresponding $\mathcal{K}$ space, at every block for each model in the ensemble. This method captures the concept evolution well. A set of entities belonging to a new class may arrive later in the stream. A new classifier built would incorporate the training entities of this new class, and would more suitably represent the data distribution. However, we do not perform any novel class detection (Masud et al., 2010). We leave this for future work. Further, we evaluate the proposed approach by using the same classifier type for each model in the ensemble. Instead, a heterogeneous set of classifiers may be used. We leave this for future work as well.

Entity-fingerprints summarize their data instances arriving in the stream in a k-dimensional vector. Therefore, the total amount of memory used to store these fingerprints is $O(kN)$, where N is the finite set of entities in the dataset. On the other hand, each model $\mathcal{M}$ is a $(k + N + 1)$-tuple where the $|\mathbf{A}|_{max} = k$ and $|\mathbf{E}|_{max} = N$. Therefore, the total memory required to store a model is $O(kN)$. At any point, the ensemble approach has a maximum of $Z = \kappa + |\mathbb{M}|$ models, which are formed by the union of models from the ensemble set $\mathcal{Q}$, and the models in $\mathbb{M}$. Here, $|(\cdot)|$ represents the cardinality of a set.

In ESSC, data instances in the stream are processed sequentially. The statistical summary is stored, and a classifier is built for each model in the ensemble during the testing phase. Since each data instance is processed exactly once, if B is the number of data instances in the stream, $O(B)$ time is required to gather their statistical information. Anchor points are periodically computed using a small set of training data instances ($\mathbf{T}$) from a block. This process takes $O(|\mathbf{T}|kdi)$ time, where k is the number of anchor points, d is the number of features of data instances, and i is the number of iterations needed until convergence. The time taken to update the accuracy of any classifier depends on the choice of the classification algorithm used. If this takes $O(C)$ time, building j models from the block requires $O(j(C + |\mathbf{T}|kdi))$ time.

In the EnsembleCP approach, class profiles of a class are constructed by combining a set of close fingerprints belonging to the same class, to form class profile distributions that are used for predicting the class label of test entities. The test entity fingerprint with sufficient statistics is compared with these class profile distributions for closeness. Therefore, during the construction of class profiles, the choice of fingerprints is crucial for a better quality distributional representation of the class. By generating more classifiers and using a large number of anchor point selections across the data stream, our approach shows a higher probability of achieving better quality class profiles.

2.6 Conclusion

We present ESSC, a framework for ensemble-based setwise stream classification. We show techniques to utilize various classification algorithms to predict the class label of a set of data instances. By constructing anchor points periodically along the stream, we show a superior classification performance compared to a previous approach that uses only a fixed initial set of anchor points to process the entire data stream. We perform extensive experiments to show that ESSC effectively addresses concept drift and concept evolution. We perform

a systematic study to determine suitable parameter values in ESSC. In addition, we use real-world datasets including one to perform a Website Fingerprinting attack.

Algorithm 1: Ensemble-based Setwise Stream Classification (**ESSC**).

Data: $< Y_r, \mathcal{E}_r, label_r >$ stream
Input: BlockSize β, NumBlockBuffer r, EnsembleSize κ
Result: Label predictions for test entities, $\mathcal{E}_{test}$
begin
  Initialize ensemble of models $\mathcal{Q}$;
  Initialize buffer model $\mathbb{M}$;
  while *data is streaming* **do**
    $b \leftarrow$initializeBlock(β);
    for $i \leftarrow 1$ **to** v **do**
      $trainData \leftarrow$getTrainingData(b);
      $anchorPoints \leftarrow$getAnchorPoints($trainData$,b);
      $\mathcal{M}_b^i \leftarrow$Initialize($anchorPoints$,$b$);
      $\mathbb{M} \leftarrow \{\mathcal{M}_b^i, 0\}$;
    end
    if $\mathcal{Q}$ *is not empty* **then**
      for *each* $\mathcal{M} \in \mathcal{Q}$ **do**
        $\mathcal{M} \leftarrow updateFingerPrint(b, \mathcal{M})$;
        $accuracy \leftarrow predictLabel(b, \mathcal{M})$;
        $\mathcal{Q} \leftarrow updateAccuracy(accuracy, \mathcal{M})$;
      end
    end
    if $\mathbb{M}$ *is not empty* **then**
      for *each* $\{\mathcal{M}, h\} \in \mathbb{M}$ **do**
        if $h == r$ **then**
          $\mathcal{M} \leftarrow updateFingerPrint(b, \mathcal{M})$;
          $accuracy \leftarrow predictLabel(b, \mathcal{M})$;
          $\mathcal{Q} \leftarrow updateAccuracy(accuracy, \mathcal{M})$;
          $\mathbb{M} \leftarrow \mathbb{M} \setminus \mathcal{M}$;
        end
        else
          $\{\mathcal{M}, h\} \leftarrow updateFingerPrint(b, \mathcal{M})$;
          $\mathbb{M} \leftarrow \{\mathcal{M}, (h+1)\}$;
        end
      end
    end
    while *size(* $\mathcal{Q}$*)>* κ **do**
      $\mathcal{Q} \leftarrow$ Remove least accurate $\mathcal{M}$;
    end
  end
end

CHAPTER 3

CALCULATING EDIT DISTANCE FOR LARGE SETS OF STRING PAIRS USING MAPREDUCE [1]

This chapter analyzes the parallelization of calculating edit distance for a large set of strings using MapReduce framework. Parallel versions of the dynamic programming solution for this problem are developed and empirical studies that compare their performances are presented.

3.1 Introduction

Given two strings s and t, the minimum number of edit operations required to transform s into t is called the edit distance. The edit operations commonly allowed for computing edit distance are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character. For these operations, edit distance is sometimes called Levenshtein distance (nlp.stanford.edu, 2014). For example, the edit distance between 'tea' and 'pet' is 2.

There are a number of algorithms that compute edit distances (Wagner and Fischer, 1974), (Sankoff and Kruskal, 1983a), (Masek and Patterson, 1980) and solve other related problems (Hall and Dowling, 1980), (Sellers, 1980), (Yang and Pavlidis, 1990). Edit distance has placed an important role in a variety of applications due to its computational efficiency and representational efficacy. It can be used in approximate string matching, optical character recognition, error correcting, pattern recognition (Fu, 1982), redisplay algorithms for video editors, signal processing, speech recognition, analysis of bird songs and comparing genetic sequences (Marzal and Vidal, 1993). Sankoff and Kruskal provide a comprehensive

[1] Authors: Shagun Jhaver, Latifur Khan and Bhavani Thuraisingham

compilation of papers on the problem of calculating edit distance (Sankoff and Kruskal, 1983b).

The cost of computing edit distance between any two strings is roughly proportional to the product of the two string lengths. This makes the task of computing the edit distance for a large set of strings difficult. It is computationally heavy and requires managing large data sets, thereby calling for a parallel processing implementation. MapReduce, a general-purpose programming model for processing huge amounts of data with a parallel, distributed algorithm appears to be particularly well adapted to this task. This chapter reports on the application of MapReduce, using its open source implementation Hadoop to develop a computational procedure for efficiently calculating edit distance.

The edit distance is usually computed by an elegant dynamic programming procedure (nlp.stanford.edu, 2014). Although, like the divide-and-conquer method, dynamic programming solves problems by combining the solutions to subproblems, it applies when the subproblems overlap - that is, when subproblems share subsubproblems (Cormen et al., 1990). Each subsubproblem is solved just once, and then the answer is saved, thereby avoiding the work of recomputing the answer every time it solves each subproblem. Unlike divide-and-conquer algorithms, dynamic programming procedures do not partition the problem into disjoint subproblems, therefore edit distance calculation does not lend itself naturally to parallel implementation. This chapter develops an algorithm for calculating the edit distance for MapReduce framework and demonstrates the improvement in performance over the usual dynamic programming algorithm used in parallel.

We implement the dynamic programming approach for this problem in a top-down way with memoization (Cormen et al., 1990). In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem in an associative array. The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure

computes the value in the usual manner (Cormen et al., 1990). Finding edit distance of a pair of strings (s, t) entails finding the edit distance of every pair (s', t'), where s' and t' are substrings of s and t respectively. All these distances are saved in an associative array h. Subsequently, if any new pair of strings share a pair of substrings for which the distance is already stored in h, the saved values are used, thereby saving the computation time. Pairs of strings that are likely to share common substrings are processed together, thus improving the performance over the standard dynamic programming parallel application for this problem.

The contributions of this work are as follows. First, to the best of our knowledge, this is the first work that addresses the calculation of unnormalized edit distance for a large number of string pairs in a parallel implementation. Our implementation in MapReduce improves upon the performance of usual dynamic programming implementation on a single machine. Second, our proposed approach, which uses an algorithm tailored to the MapReduce framework architecture performs better than the simple parallel implementation. Finally, this serves as an example of using the MapReduce framework for dynamic programming solutions, and paves the way for parallel implementation for other dynamic programming problems.

In particular, the requirement for calculating edit distance for a large number of pairs of strings emerged in one of our previous research projects (Parveen et al., 2013) on finding normative patterns over dynamic data streams. This project uses an unsupervised sequence learning approach to generate a dictionary which will contain any combination of possible normative patterns existing in the gathered data stream. A technique called compression method (CM) is used to keep only the longest and most frequent unique patterns according to their associated weight and length, while discarding other subsumed patterns. Here, edit distance is required to find the longest patterns.

The remainder of this chapter is organized as follows. Section II discusses the problem statement and the dynamic programming solution to the problem on a single machine.

Section III discusses our proposed approach, and the techniques used in detail. Section IV reports on the experimental setup and results. Section V then describes the related work, and Section VI concludes with directions to future work.

3.2 Background

The edit distance problem is to determine the smallest number of edit operations required for editing a source string of characters into a destination string. For any two strings $s = s_1 s_2 .... s_m$ and $t = t_1 t_2 ... t_n$ over an input alphabet of symbols $\sigma = \{a_1, a_2, ... a_r\}$, the valid operations to transform s into t are:

- Insert a character t_j appearing in string t
- Delete a character s_i appearing in string s
- Replace a character s_i appearing in string s by a character t_j in string t

For strings $s = s_1 s_2 .... s_m$ and $t = t_1 t_2 ... t_n$, and an associative array h storing the edit distance between s and t, this problem can be solved sequentially in $O(mn)$ time. The memoized dynamic programming algorithm for this, MEM_ED, is described in Algorithm 3.1:

For an input pair of strings (s, t), step 1 in MEM_ED algorithm checks whether the pair is already stored in the input associative array h. If present, the algorithm returns the stored value for (s, t) in step 2. If one of the strings is empty, MEM_ED returns the length of the other string as the output. Steps 10-11 in this algorithm divide the problem inputs into subproblem inputs of smaller size. Steps 12 - 14 calculate the edit distance recursively for these subproblems. Step 20 derives the edit distance for the problem, and step 21 stores this result in an associative array, h for further use, thereby memoizing the recursive procedure.

Fig. 3.2 shows the associative array entries for calculating the edit distance between two strings - 'levenshtein' and 'meilenstein'. For example, for calculating the edit distance

Algorithm 1 EDIT-DISTANCE($s[1, 2, ..m]$, $t[1, 2, ..., n]$, h): (**MEM_ED**)

```
1:  if pair(s, t) in h then
2:      return h[pair(s, t)]
3:  end if
4:  if len(s)==0 then
5:      return t.length
6:  end if
7:  if len(t)==0 then
8:      return s.length
9:  end if
10: s' ← s[1, 2, ...m − 1]
11: t' ← t[1, 2, ...n − 1]
12: k_a ← EDIT-DISTANCE(s', t')
13: k_b ← EDIT-DISTANCE(s', t) + 1
14: k_c ← EDIT-DISTANCE(s, t') + 1
15: if s[m]==t[n] then
16:     k_d ← k_a
17: else
18:     k_d ← k_a + 1
19: end if
20: c ← min(k_b, k_c, k_d)
21: h[pair(s, t)] ← c
22: return c
```

Figure 3.1. EDIT-DISTANCE($s[1, 2, ..m]$, $t[1, 2, ..., n]$, h): (MEM_ED).

between the string pair (‘*levens*’, ‘*meilens*’), the edit distances k_a, k_b and k_c for the pairs (‘*leven*’, ‘*meilen*’), (‘*levens*’, ‘*meilen*’) and (‘*leven*’, ‘*meilens*’) are considered respectively. By a recursive procedure in steps 12-14 of MEM_ED, these values are calculated to be 3,

4 and 4 respectively. Since the input string pair (‘*levens*’, ‘*meilens*’) have the same last character ‘*s*’, the value k_d is calculated to be equal to $k_a = 3$ in steps 15-19 of MEM_ED. Step 20 computes c, the minimum of k_b, k_c and k_d to be 3. Step 21 associates string pair (‘*levens*’, ‘*meilens*’) with value 3 in the associative array h for further use. Step 22 returns this edit distance value.

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Figure 3.2. Edit Distance between two strings.

On a single machine, we compute the edit distance for every pair of distinct strings in an input text document by repeatedly using MEM_ED for each pair of distinct strings. The SIN_ED procedure in Algorithm 3.3 describes this approach.

Step 1 in SIN_ED algorithm collects all the distinct strings in the input document. Step 3 initializes an associative array. Step 4 uses the EDIT_DISTANCE procedure of MEM_ED to calculate the edit distance for each distinct string pair. The implementation of SIN_ED takes $O(t^2n^2)$ time for t distinct strings and string length of order n. This is computationally very expensive; hence we need to implement this algorithm in parallel for faster computations.

Algorithm 2 Single Machine Implementation for calculating Edit Distance for all string pairs (**SIN_ED**)

```
1: dist_strings ← list of distinct strings in doc d
2: for all string pairs (s, t) ϵ dist_strings do
3:     H ← new ASSOCIATIVE_ARRAY
4:     c ← EDIT_DISTANCE(s, t, H)
5:     EMIT(pair(s, t), c)
6: end for
```

Figure 3.3. Single Machine Implementation for calculating Edit Distance for all string pairs (SIN_ED).

3.3 Related Work

Extensive studies have been done on edit distance calculations and related problems over the past several years. Ristad and Yianilos (Ristad and n. Yianilos, 1998) provide a stochastic model for learning string edit distance. This model allows for learning a string edit distance function from a corpus of examples. Bar-yossef, Jayram, Krauthgamer and Kumar develop algorithms that solve gap versions of the edit distance problem (Bar-Yossef et al., 2004): given two strings of length n with the promise that their edit distance is either at most k or greater than l, these algorithms decide which of the two holds.

A lot of studies have been dedicated to normalized edit distance to effect a more reasonable distance measure. Abdullah N. Arslan and Ömer Egecioglu discuss a model for computing the similarity of two strings X and Y of lengths m and n respectively where X is transformed into Y through a sequence of three types of edit operations: insertion, deletion, and substitution. The model assumes a given cost function which assigns a non-negative real weight to each edit operation. The amortized weight for a given edit sequence is the ratio of its weight to its length, and the minimum of this ratio over all edit sequences is

the normalized edit distance. Arslan and Egecioglu (Arslan, 1999) give an O(mn logn)-time algorithm for the problem of normalized edit distance computation when the cost function is uniform, i.e, the weight of each edit operation is constant within the same type, except substitutions which can have different weights depending on whether they are matching or non-matching.

Jie Wei proposes a new edit distance called Markov edit distance (Wei, 2004) within the dynamic programming framework, that takes full advantage of the local statistical dependencies in the string/pattern in order to arrive at enhanced matching performance. Higuera and Micó define a new contextual normalized distance, where each edit operation is divided by the length of the string on which the edit operation takes place. They prove that this contextual edit distance is a metric and that it can be computed through an extension of the usual dynamic programming algorithm for the edit distance (de la Higuera, 2008).

Fuad and Marteau propose an extension to the edit distance to improve the effectiveness of similarity search (Fuad, 2008). They test this proposed distance on time series data bases in classification task experiments and prove, mathematically, that this new distance is a metric.

Robles-Kelly and Hancock compute graph edit distance by converting graphs to string sequences, and using string matching techniques on them (Robles-Kelly and Hancock., 2005). They demonstrate the utility of the edit distance on a number of graph clustering problems. Bunke introduces a particular cost function for graph edit distance and shows that under this cost function, graph edit distance computation is equivalent to the maximum common subgraph problem (Bunke, 1997).

Hanada, Nakamura and Kudo discuss the issue of high computational cost of calculating edit distance of a large set of strings (Hanada et al., 2011). They contend that a potential solution for this problem is to approximate the edit distance with low computational cost. They list the edit distance approximation methods, and use the results of experiments implementing these methods to compare them. Jain and Rao present a comparative

study to evaluate experimental results for approximate string matching algorithms such as Knuth-Morris-Pratt, Boyer-Moore and Raita on the basis of edit distance (Jain and Rao, 2013).

A few studies have also been done that target a parallel implementation of calculating normalized edit distance. Instead, in this work, we address the calculation of unnormalized edit distance for large number of string pairs in a parallel implementation, and we use MapReduce for it.

3.4 Proposed Approach

We discussed in the Background section that the single machine implementation for calculating the edit distance of all distinct pairs of strings, described in SIN_ED, is computationally expensive. We propose a parallel computing approach to do this more efficiently.

MapReduce is emerging as an important programming model for expressing distributed computations in data-intensive applications (Yan et al., 2012). It was originally proposed by Google and is built on well-known principles in parallel and distributed processing dating back several decades. MapReduce has since enjoyed widespread adoption via Hadoop, a popular open-source implementation developed primarily by Yahoo and Apache. It enables easy development of scalable approaches to efficiently processing massive amounts of data on clusters of commodity machines. MapReduce systems are evolving and extending rapidly and today, Hadoop is a core part of the computing infrastructure for many web companies, such as Facebook, Amazon, Yahoo and Linkedin. Because of its high efficiency, high scalability, and high reliability, MapReduce framework is used in many fields (Yan et al., 2012), such as life science computing (Ekanakake et al., 2010), text processing, web searching, graph processing (Malewicz et al., 2010), relational data processing, data mining, machine learning (He et al., 2011) and video analysis (Pereira et al., 2010).

We use the MapReduce framework for the parallel implementation of calculating edit distance for a large set of strings. The idea is to use the associative array in SIN_ED to store the edit distances across the computations for many pairs of strings. Once the edit distance for a pair of strings (s, t) is calculated, the edit distance for all pairs (s', t'), where s' and t' are substrings of s and t respectively are stored in the associative array. Subsequent to this, for a new pair of strings (a, b), the calculations at steps 12, 13 and/or 14 in MEM_ED can be saved, if the input pairs of strings for these steps already have an entry in the associative array.

The SIM_MR algorithm (Algorithm 3.4) describes a simple Map Reduce approach to calculating edit distance in parallel using these ideas.

SIM_MR first constructs a list of distinct strings from the input document in Step 3 in the Mapper phase. The '*count*' variable initialized in Step 4 tracks the count of the string pair being processed. The '$num\_reducers$' parameter determines the number of reducers to be used in the reduce phase of the procedure. The '$reducer\_index$' variable determines the reducer that would process the current string pair. The value of '$reducer\_index$' is calculated in Step 7. This value is independent of the strings in the string pair being processed. Step 8 emits with '$reducer\_index$' as the key and the current string pair as the value.

In the reduce phase, an associative array, H is initialized in Step 13. For every input string pair, Step 15 calculates the edit distance of the current string pair using H with the EDIT_DISTANCE procedure of MEM_ED. The entries stored in H during the edit distance calculations of any string pair can be used across calculations for different string pairs. Step 16 emits the string pair with its corresponding edit distance value.

We note that the $reducer\_index$ value in SIM_MR depends just on the count of the string pair being processed. We propose a modified algorithm that uses the strings in the string pair to effect a more efficient way of determining the reducer where the current string pair gets processed.

Algorithm 3 Simple MapReduce approach to calculating Edit Distance for all string pairs (**SIM_MR**)

```
1:  class MAPPER
2:      method MAP(docid a, doc d)
3:          dist_strings ← list of distinct strings in doc d
4:          count ← 0
5:          for all string pairs (s, t) ϵ dist_strings do
6:              count ← count + 1
7:              reducer_index ← count % num_reducers
8:              EMIT(reducer_index, pair(s, t))
9:          end for
10:
11: class REDUCER
12:     method REDUCE(reducer_index, pairs [(s1, t1), (s2,
        t2),...])
13:         H ← new ASSOCIATIVE_ARRAY
14:         for all string pairs (s, t) ϵ pairs [(s1, t1), (s2, t2),...]
        do
15:             c ← EDIT_DISTANCE(s, t, H)
16:             EMIT(pair(s, t), c)
17:         end for
```

Figure 3.4. Simple MapReduce approach to calculating Edit Distance for all string pairs (SIM_MR).

The pairs of strings to be processed at a single node need to be chosen such that they are likely to have some pairs of substrings for which the edit distance has already been computed, and the computation time is saved via an associative array look-up. To accomplish this, we collect all pairs of strings with a common prefix pair at a single reducer node. This prefix pair is constructed by taking the first $prefix\_length$ characters of both strings to form a

Algorithm 4 Prefixed MapReduce approach to calculating Edit Distance for all string pairs (**PRE_MR**)

```
1:  class MAPPER
2:      method MAP(docid a, doc d)
3:          dist_strings ← list of distinct strings in doc d
4:          for all string pairs (s, t) ϵ dist_strings do
5:              s_prefix ← s[1 : prefix_length]
6:              t_prefix ← t[1 : prefix_length]
7:              EMIT(pair(s_prefix, t_prefix), pair(s, t))
8:          end for
9:
10: class REDUCER
11:     method REDUCE(prefix_pair (s', t'), pairs [(s1, t1),
    (s2, t2),...])
12:         H ← new ASSOCIATIVE_ARRAY
13:         for all string pairs (s, t) ϵ pairs [(s1, t1), (s2, t2),...]
    do
14:             c ← EDIT_DISTANCE(s, t, H)
15:             EMIT(pair(s, t), c)
16:         end for
```

Figure 3.5. Prefixed MapReduce approach to calculating Edit Distance for all string pairs (PRE_MR).

string pair. The procedure for the proposed approach, PRE_MR, is described in Algorithm 3.5.

For the current string pair (s,t), Steps 5 and 6 in PRE_MR calculate the s_prefix and t_prefix values by taking the first $prefix\_length$ characters from s and t respectively. For example, for $prefix\_length = 2$, and string pair (s,t) = ('mango', 'gate'), the s_prefix and

t_prefix values are computed to be 'ma' and 'ga' respectively. Step 7 emits with the string pair (s_prefix, t_prefix) as the key, and the string pair (s, t) as the value.

The reduce phase for PRE_MR is similar to the reduce phase in the SIM_MR procedure. An associative array H is initialized in step 12, and the edit distance of every string pair in pairs [(s_1, t_1), (s_2, t_2),...] is computed using the EDIT_DISTANCE procedure of MEM_ED. Step 15 emits the results.

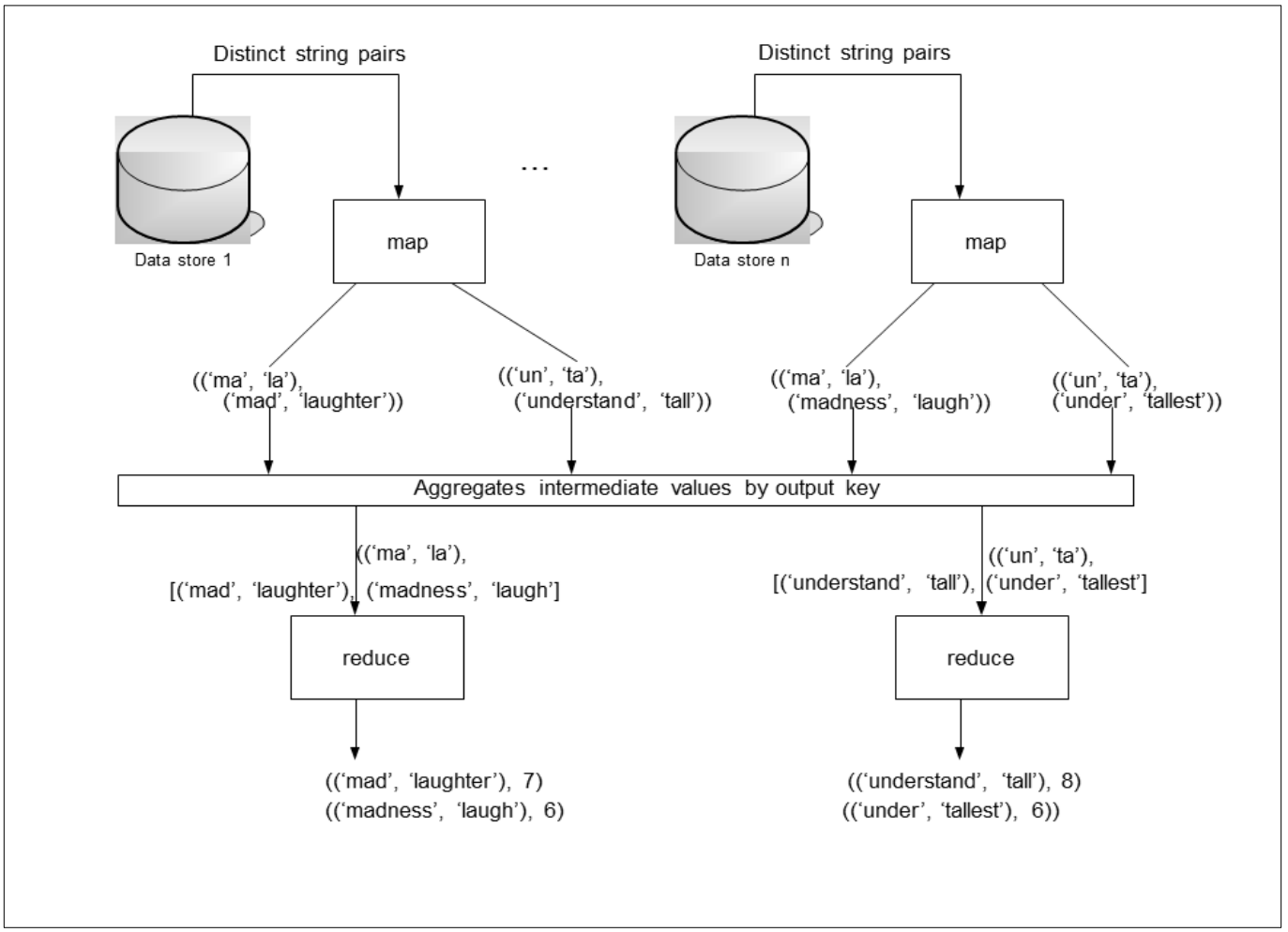


Figure 3.6. PRE_MR algorithm flow-chart.

Hadoop runs its map and reduce processes in such a way that these processes operate on independent chunks of data and have no inter process communication. We've customized

our algorithms to satisfy this constraint. For PRE_MR, the mapper sends all string pairs sharing the same pair of prefixes to a single reducer. In the reduce stage, all these string pairs are processed together. For each string pair, the associative array H saves the calculated intermediate edit distances, and a look-up in this array often saves the computations for many other pairs of strings that are input to this reducer. These savings in computations make our algorithms, especially PRE_MR more efficient than the SIN_ED approach.

Fig. 3.6 shows an example of the implementation for PRE_MR algorithm with '$prefix\_length$' = 2. Mapper constructs a prefix pair ('ma', 'la') for input pair of strings ('mad', 'laughter'), and emit with ('ma', 'la') as the key and ('mad', 'laughter') as the value. In the reduce phase, all strings pairs sharing the prefix ('ma', 'la') are processed together. Therefore, the string pairs ('mad', 'laughter') and ('madness', 'laugh') are processed at the same node. Since these string pairs share common substrings, many computations are saved, and the procedure is faster.

3.5 Experimental Setup and Results

Our hadoop cluster (cshadoop0-cshadoop9) has ten virtual machines that run in the Computer Science vmware esx cloud. Each of these VM's has 4 GB of RAM and a 256 GB virtual hard drive. These VM's are spread across three ESX hosts to balance the load. We've used one name node and nine slave nodes. For our implementation, we used Hadoop version 1.0.4 and JAVA JDK version 1.6.0.37.

The data sets were created from the ebooks for which the copyright has expired. We used the text of 'Pride and Prejudice' by Jane Austen available at `http://www.gutenberg.org/ebooks/1342`, and developed files of size 10kB, 20kB,..., 100kB from it.

We implemented a preprocessing step for each of the experiments, where all the duplicate strings in the input files were eliminated, thus all the experiments described have been conducted on unique strings.

We processed each of these files using SIN_ED, SIM_MR and PRE_MR algorithms. The results are described in Table 3.1 and Fig. 3.7. It shows the comparison of the performance of neutral baseline of SIN_ED implementation (plain sequential implementation) with our proposed algorithms. For Fig. 3.7, we've taken the input file sizes (in kB) on the x-axis and the times taken by each of the procedures (in seconds) on the y-axis. These results are obtained using 4 reducer nodes.

Table 3.1. SIN_ED vs. SIM_MR vs. PRE_MR implementation.

File Size	**SIN_ED**	**SIM_MR**	**PRE_MR**
10 kB	12 sec	72 sec	68 sec
20 kB	33 sec	73 sec	70 sec
30 kB	62 sec	82 sec	71 sec
40 kB	90 sec	94 sec	76 sec
50 kB	122 sec	147 sec	79 sec
60 kB	155 sec	120 sec	80 sec
70 kB	189 sec	125 sec	85 sec
80 kB	218 sec	140 sec	88 sec
90 kB	276 sec	145 sec	93 sec
100 kB	293 sec	209 sec	101 sec

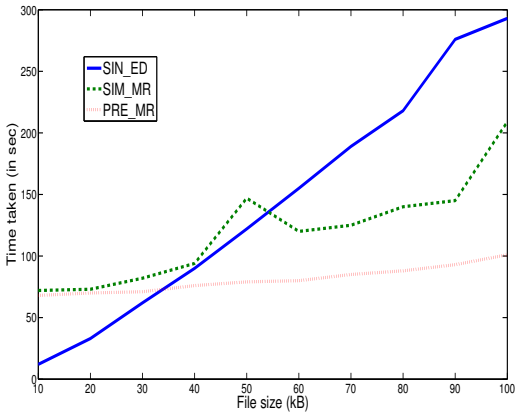


Figure 3.7. SIN_ED vs. SIM_MR vs. PRE_MR implementation.

Table I results indicate that PRE_MR algorithm gives the best results. For example, for a file of size 80 kB, SIN_ED takes 218 sec, SIM_MR takes 140 sec and PRE_MR algorithm takes 88 sec. Therefore, we conduct the rest of the experiments only for PRE_MR.

We experimented with different values of the parameter '$prefix\_length'$ used in the MAP phase for the PRE_MR implementation. The time taken for different file sizes are documented in Table 3.2, and Fig. 3.8. For Fig. 3.8, the x-axis is file size (in kB), and the y-axis is the runtime for experiments with different '$prefix\_length$' values. For this experiment, we chose to use 2 mappers and 1 reducer in each case. We see that, generally, smaller '$prefix\_length$' values tend to give better performance. For example, for a file of size 80 kB, '$prefix\_length$' = 1 case takes 100 sec, '$prefix\_length$' = 2 case takes 113 sec, '$prefix\_length$' = 3 case takes 132 sec and '$prefix\_length$' = 4 case takes 150 sec.

Table 3.2. PRE_MR performance for different $prefix\_length$ values.

File Size	***prefix_length*=1**	***prefix_length*=2**	***prefix_length*=3**	***prefix_length*=4**
10 kB	67 sec	69 sec	65 sec	66 sec
20 kB	72 sec	72 sec	75 sec	78 sec
30 kB	77 sec	79 sec	82 sec	87 sec
40 kB	79 sec	82 sec	86 sec	96 sec
50 kB	90 sec	90 sec	102 sec	112 sec
60 kB	93 sec	105 sec	115 sec	120 sec
70 kB	94 sec	108 sec	116 sec	134 sec
80 kB	100 sec	113 sec	132 sec	150 sec
90 kB	106 sec	121 sec	158 sec	155 sec
100 kB	108 sec	131 sec	134 sec	166 sec

We also experimented with different number of reducers in the PRE_MR implementation for three cases: '$prefix\_length$' = 1, '$prefix\_length$' = 2 and '$prefix\_length$' = 3. In each case, in the corresponding Fig., we take the file size as the x-axis and the runtime for the experiment as the y-axis.

Table 3.3 and Fig. 3.9 detail the times taken for this experiment when the '$prefix\_length$' parameter is set to 1. We see that the performance generally improves with increasing number

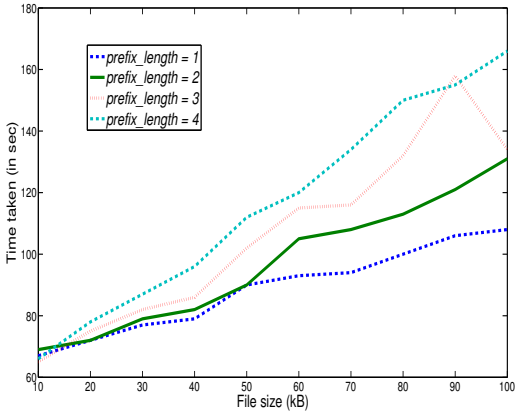


Figure 3.8. PRE_MR performance for different $prefix\_length$ values.

of reduce nodes. For example, for a file of size 80 kB, 1 reducer node takes 100 sec, 2 reducers take 91 sec and 4 reducers take 88 sec.

Table 3.3. PRE_MR performance for different number of reducers, $prefix\_length$=1.

File Size	**1 reducer**	**2 reducers**	**4 reducers**
10 kB	67 sec	65 sec	68 sec
20 kB	72 sec	68 sec	70 sec
30 kB	77 sec	69 sec	71 sec
40 kB	79 sec	75 sec	76 sec
50 kB	90 sec	86 sec	79 sec
60 kB	93 sec	95 sec	80 sec
70 kB	94 sec	90 sec	85 sec
80 kB	100 sec	91 sec	88 sec
90 kB	106 sec	96 sec	93 sec
100 kB	108 sec	112 sec	101 sec

Table 3.4 and Fig. 3.10 describe the times taken when the '$prefix\_length$' parameter in PRE_MR is set to 2. For example, for a file of size 90 kB, 1 reducer node takes 121 sec, 2 reducers take 117 sec, and 4 reducers take 102 sec.

Table 3.5 and Fig. 3.11 list the times taken for PRE_MR when the 'prefix_length' parameter is set to 3. Again, increasing the number of nodes in reduce phase tend to improve the

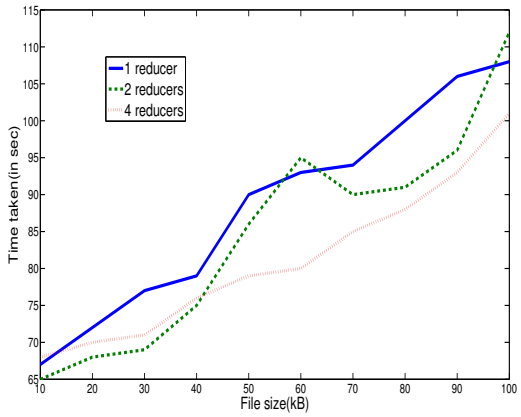


Figure 3.9. PRE_MR performance for different number of reducers, $prefix\_length$=1.

Table 3.4. PRE_MR performance for different number of reducers, $prefix\_length$=2.

File Size	**1 reducer**	**2 reducers**	**4 reducers**
10 kB	69 sec	63 sec	67 sec
20 kB	72 sec	66 sec	71 sec
30 kB	79 sec	72 sec	73 sec
40 kB	82 sec	76 sec	82 sec
50 kB	90 sec	81 sec	78 sec
60 kB	105 sec	91 sec	90 sec
70 kB	108 sec	98 sec	89 sec
80 kB	113 sec	97 sec	97 sec
90 kB	121 sec	117 sec	102 sec
100 kB	131 sec	111 sec	101 sec

performance. For example, for a file of size 80 kB, 1 reducer case takes 132 sec, 2 reducers take 124 sec and 4 reducers take 108 sec.

Table 3.6 and Fig. 3.12 describe the times taken for different number of mappers for PRE_MR with $prefix\_length$ set to 1 and 4 reducers. In Fig. 3.12, the x-axis labels the size of the input file, and the runtime for the experiment are on the y-axis. As expected, with increase in the number of mapper nodes, the performance tends to improve. For example, for a file of size 80 kB, 2 mappers take 88 sec, 4 mappers take 85 sec and 8 mappers take 82 sec.

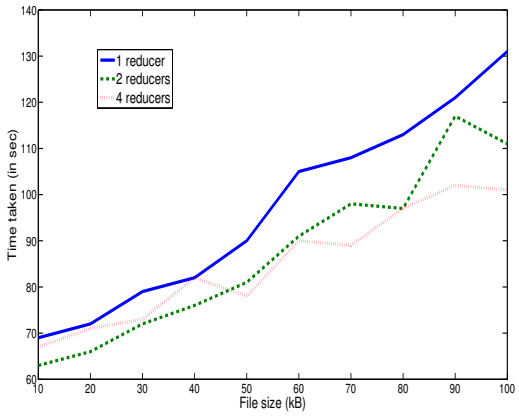


Figure 3.10. PRE_MR performance for different number of reducers, $prefix\_length$=2.

Table 3.5. PRE_MR performance for different number of reducers, $prefix\_length$=3.

File Size	**1 reducer**	**2 reducers**	**4 reducers**
10 kB	65 sec	67 sec	67 sec
20 kB	75 sec	73 sec	77 sec
30 kB	82 sec	77 sec	82 sec
40 kB	86 sec	84 sec	104 sec
50 kB	102 sec	94 sec	95 sec
60 kB	115 sec	93 sec	92 sec
70 kB	116 sec	99 sec	114 sec
80 kB	132 sec	124 sec	108 sec
90 kB	158 sec	127 sec	98 sec
100 kB	134 sec	115 sec	122 sec

We observe in the results for PRE_MR performance that the running time does not always decrease when the number of mappers or reducers increases. We believe that this is because MapReduce resources are used to split the data and send it across to different nodes, and the intermediate results need to be shuffled across the network.

For the full text of Pride and Prejudice by Jane Austen, performing PRE_MR with 2 mappers and 4 reducers and after dividing the text into chunks of 100 kB took 684 seconds, when the '$prefix\_length$' parameter is set to 2. Using SIN_ED to do this after dividing the text into chunks of 10 kB took 967 seconds. However, we note that when reducing the file

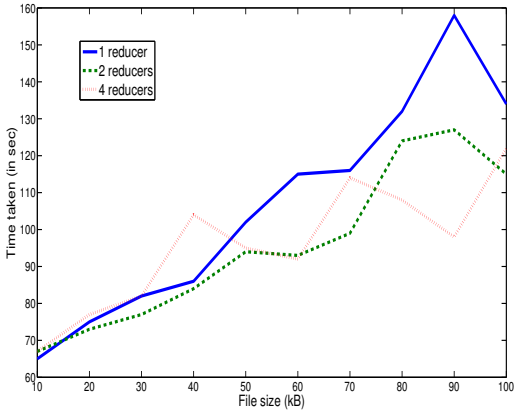


Figure 3.11. PRE_MR performance for different number of reducers, $prefix\_length$=3.

Table 3.6. PRE_MR performance for different number of mappers, $prefix\_length$=1, number of reducers=4.

File Size	**2 mappers**	**4 mappers**	**8 mappers**
10 kB	68 sec	67 sec	66 sec
20 kB	70 sec	67 sec	67 sec
30 kB	71 sec	69 sec	67 sec
40 kB	76 sec	75 sec	76 sec
50 kB	79 sec	73 sec	74 sec
60 kB	80 sec	84 sec	78 sec
70 kB	85 sec	84 sec	82 sec
80 kB	88 sec	85 sec	82 sec
90 kB	93 sec	91 sec	90 sec
100 kB	101 sec	101 sec	90 sec

chunk size, the number of distinct string pairs, p reduce drastically, as p is proportional to the square of the number of distinct strings. So, we expect that the performance improvement using PRE_MR is much more than what this result indicates.

We verified the reproducibility of the experiments by carrying out each of the experiments multiple times, and taking the average values. Besides, it was found that the results obtained had little standard deviation. In some additional experiments, for an increasing number of compute nodes, the improvement in performance was found to be quite substantial as the file

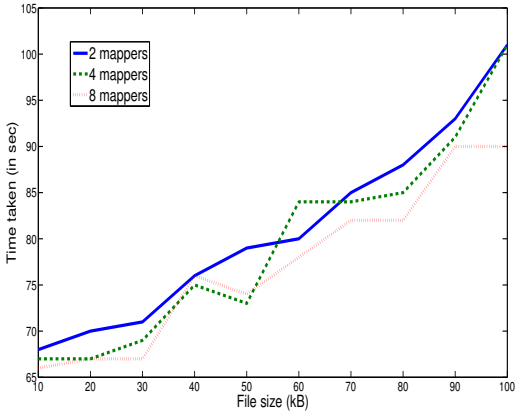


Figure 3.12. PRE_MR performance for different number of mappers.

size increased over 100 kB. For keeping the uniformity of the results across all experiments, we haven't presented the results for file sizes more than 100 kB or for larger number of map and reduce nodes, since we hadn't evaluated these cases on all experiments. The presented results are aimed to show trends in performance change with varying file sizes and different number of computation nodes.

3.6 Conclusions and Future Work

Although there are several efficient algorithms for calculating edit distance and related problems, computing edit distance for a large set of strings is expensive. We propose an efficient parallel implementation for this, using MapReduce. With support from our experimental results of Section IV, we argue that our approach is much more efficient than the usual dynamic programming method. We can also tune the '$prefix\_length$' parameter in PRE_MR, and the number of nodes used in the map phase and reduce phase to improve the performance of our algorithms for varying input file sizes.

As the number of mapper and reducer nodes are increased in MapReduce, there is greater parallelization and the number of processes increase. In Table 3.1, PRE_MR is three times

faster than the sequential procedure because it uses 4 reducers instead of 1. The speedup is not substantial when doubling the mappers and reducers because as mentioned previously, MapReduce resources are used to split the data and send it to these nodes, and the intermediate results are shuffled across the network. However, we expect this to get more than compensated for with larger files, where each prefix pair would be expected to have a larger number of corresponding string pairs, and thus each reduce process initiated would produce more results.

The optimal number of mappers, reducers and the '$prefix\_length$' parameter value vary with the file size and file content. It is hoped that the results on varied experiments presented can help guide towards a good initial guess for these parameters.

The field of dynamic programming problems is far from exhausted when it concerns creating scalable, effective, parallel algorithms. We argue, however, that our algorithms are a step in the right direction. Future research includes further testing to explore their efficiency in different datasets. In addition, further analysis of dynamic programming algorithms can lead to more effective MapReduce solutions, especially for problems that require ad-hoc data analysis.

CHAPTER 4

COMPARATIVE ANALYSIS OF CLASSIFIERS PREDICTING POLITENESS AND APPLICATION IN WEB-LOGS [1]

This chapter develops a computational framework for identifying and characterizing politeness markings in text documents. A number of classifiers are constructed for this task and a comparison of their results is presented. An application is also designed where this framework has been used to study the politeness levels in a variety of web-logs.

4.1 Introduction

Politeness, deference and tact have a sociological significance altogether beyond the level of table manners and etiquette books (Goffman, 1971). Politeness, introduced into linguistics more than forty years ago, has emerged as a vital and rapidly developing area of study. Brown and Levinson's (1978, 1987) classic treatment of linguistic politeness show that politeness strategies are a basis for social order. The concepts inherent to their model have been invoked in much subsequent literature which has focused on linguistic carriers of politeness (e.g., speech acts, syntactic constructions, lexical items, etc.), seeking to quantify them, to compare them across cultures and genders, and to identify universals (Meier, 1999).

Danescu-Niculescu-Mizil, Sudhof, Jurafsky, Leskovec and Potts (Danescu-Niculescu-Mizil et al., 2013) develop a computational framework for identifying and characterizing the linguistic aspects of politeness. Their investigation is guided by a new corpus of requests annotated for politeness, that they constructed and released. This corpus consists of a large

[1] Authors: Shagun Jhaver and Latifur Khan

collection of requests from two different sources - Wikipedia and Stack Exchange. Both of these are large online communities in which users frequently make requests of other members.

In this Project, We use this richly labeled data for politeness to construct politeness classifiers using different supervised and unsupervised machine learning algorithms, and present a comparative analysis of the performance of these classifiers. We also study the improvement in classifiers' performance after they use a wide range of lexical, sentiment and dependency features operationalizing key components of politeness theory.

We observe that some of the classifiers achieve near human-level accuracy across different test-sets, which demonstrates the consistent nature of politeness strategies, and We use these classifiers with new data for further analysis of the relation of politeness to social factors. We select the web-log (blog) entries from a variety of blogs, assign these entries a politeness score on a scale of 0 to 1 using the classifiers we build, and compare these scores.

4.2 Background

The meaning of politeness and concomitant concepts, and the claims for universals have shown considerable divergence and lack of clarity as they have received increased attention since Brown and Lewinson's proposed framework (Meier, 1999), (Brown and Levinson, 1978), (Brown and Levinson, 1987). Scholars use a variety of approaches to account for politeness: the social-norm view, the conversational-maxim view; the face-saving view; and the conversational-contract view (Fraser, 1990). While none of these views is considered adequate, the face-saving view by Brown and Levinson is seen as the most clearly articulated and is the most popular.

Brown and Levinson contend that linguistic politeness must be communicated, that it constitutes a message. They assert that the failure to communicate the intention to be polite may be taken as absence of the required polite attitude. They propose a framework

to explain politeness in which their rational Model Person has 'face', the individual's self-esteem. This face is a culturally elborated public self-image that every member of a society wants to claim for himself (Fraser, 1990). They characterize two types of face in terms of participant wants rather than social norms:

Negative Face: "the want of every 'competent adult member' that his action be unimpeded by others"

Positive Face: "the want of every member that his wants be desirable to at least some others"

The organizing principle for their politeness theory is the idea that "some acts are intrinsically threatening to face and thus require softening ..." To this end, each group of language users develops politeness principles from which they derive certain linguistic strategies. It is by the use of these so-called politeness strategies that speakers succeed in communicating both their primary message(s) as well as their intention to be polite in doing so. And in doing so, they reduce the face loss that results from the interaction.

The choice of a specific linguistic form is to be viewed as a specific realization of one of the politeness strategies in light of the speaker's assessment of the utterance context. Brown and Levinson outline four main types of politeness strategies: bald on-record, negative politeness, positive politeness, and off-record (indirect). The speaker must choose a linguistic means that will satisfy the strategic end. Since each strategy embraces a range of degrees of politeness, the speaker will be required to consider the specific linguistic forms used and their overall effect when used in conjunction with one another.

We try to identify such strategies and use them to construct the classifiers. A brief description of the classifiers We used is given below.

1) *Naive Bayes*: Naive Bayes is a highly practical learning method whose performance is shown to be comparable to that of neural network and decision tree learning in some domains. It applies to the learning tasks where each instance x is described by a conjunction

of attribute values and where the target function $f(x)$ can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $<a_l, a_2, ..., a_n>$. The learner is asked to predict the target value, or classification, for this new instance. The Bayesian approach to classifying the new instance is to assign the most probable target value, V_{MAP}, given the attribute values $<a_l, a_2, ..., a_n>$ that describe the instance (Mitchell, 1997).

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction $a_l, a_2, ..., a_n$ is just the product of the probabilities for the individual attributes.

2) *Naive Bayes Multinomial*: In the multinomial model, a document is an ordered sequence of word events, drawn from the same vocabulary V. It is assumed that the lengths of documents are independent of class. Again, an assumption similar to naive Bayes is made: that the probability of each word event in a document is independent of the word's context and position in the document. Thus, each document d_i is drawn from a multinomial distribution of words with as many independent trials as the length of d_i. This yields the familiar "bag of words" representation for documents (McCallum and Nigam, 1998). Whereas simple naive Bayes would model a document as the presence and absence of particular words, multinomial naive bayes explicitly models the word counts and adjusts the underlying calculations to deal with in.

3) *J48*: J48 is an open source Java implementation of the C4.5 algorithm in the weka data mining tool. C4.5 is an algorithm used to generate a decision tree developed by Ross Quinlan. C4.5 is an extension of Quinlan's earlier ID3 algorithm. The decision trees generated by C4.5 can be used for classification, and for this reason, C4.5 is often referred to as a statistical classifier. At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion

is the normalized information gain (difference in entropy). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurses on the smaller sublists (j48, 2014).

4) *Random Forest*: Random Forests grow many classification trees. To classify a new object from an input vector, the input vector is put down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Each tree is grown as follows:

1. If the number of cases in the training set is N, sample N cases at random - but with replacement, from the original data. This sample will be the training set for growing the tree.

2. If there are M input variables, a number m<<M is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.

3. Each tree is grown to the largest extent possible. There is no pruning.

5) *IBk*: This is an implementation of the k-nearest neighbors algorithm. This basic instance-based algorithm assumes all instances correspond to points in the n-dimensional space. The nearest neighbors of an instance are defined in terms of the standard Euclidean or Manhattan distance. The value of the classification label for an input x returned by this algorithm is just the most common value of label among the k training examples nearest to x (Mitchell, 1997).

6) *SMO*: Sequential minimal optimization (SMO) is an algorithm for solving the optimization problem which arises during the training of support vector machines. It was invented by John Platt in 1998 at Microsoft Research. SMO is widely used for training

support vector machines and is implemented by the popular LIBSVM tool. SMO breaks the optimization problem in SVM into a series of smallest possible sub-problems, which are then solved analytically (smo, 2014).

4.3 Experiments

Similar to Danescu-Niculescu-Mizil et al (Danescu-Niculescu-Mizil et al., 2013), the training data is from two different domains:

1. Wikipedia
2. Stack Exchange

The experiments are on two different types of classifiers:

1. Bag of Words classifier (BOW)
2. Linguistically Informed classifer (Ling.)

For Linguistically Informed classifer (Ling.), We use the features described in (Danescu-Niculescu-Mizil et al., 2013) and reproduced here in Table 4.1:

We use Weka (Waikato Environment for Knowledge Analysis), a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand for all my experiments.

We run experiments of two types:

1. In-domain: We use 5-fold cross-validations for these experiments. The experiments are:
    - Training on Wikipedia, Testing on Wikipedia
    - Training on Stack-Exchange, Testing on Stack-Exchange

Table 4.1. Politeness Strategies used by Danescu-Niculescu-Mizil et al (Danescu-Niculescu-Mizil et al., 2013) for features in Linguistically Informed Classifiers.

Strategy	Description
Gratitude	Contains words like “appreciate”, “thankful”, “grateful”, “recognize”, “indebted”
Deference	Contains words like “Nice work”, “respect”, “polite”
Greeting	Use of words like “Hey”, “Hi”, “Hello”, “take care”, “bye”, “Good morning”, “Dear”, “what’s up”, “welcome”
Positive lexicon	Contains words in positive lexicon
Negative lexicon	Contains words in negative lexicon
Apologizing	Contains words like “sorry”, “pardon”, “regret”, “apologize”, “ashamed”, “regretful”, “penitent”
Please	Contains “please”
Please start	Starts with “please”
Indirect (btw)	Contains phrases like “by the way”, “btw”
Direct question	Contains sentences beginning with “wh” and ending with “?”
Direct start	Contains sentences beginning with “So”, “Well”, etc
Counterfactual modal (Could/Would)	Contains sentences beginning with “could”, “would”, etc
Indicative modal (Can/Will)	Contains sentences beginning with “can”, “will”, etc
First Person Start	Contains sentences beginning with “I”, “We”, etc.
First Person plural	Use of words like “I”, etc.
First Person	Use of words like “me”, etc.
Second Person	Contains words like “you”, etc.
Second Person Start	Contains sentences beginning with “you”, “your”, etc.
Hedges	Contains phrases like “I suggest”
Factuality	Contains phrases like “In fact”

2. Cross-domain:

    - Training on Wikipedia, Testing on Stack-Exchange
    - Training on Stack-Exchange, Testing on Wikipedia

The training and 5-fold cross-validation (In-domain) is done as follows:

1. Sort the training requests by their politeness scores.
2. Get top 25% of requests, and label them as positive.
3. Get bottom 25% of requests, and label them as negative.
4. Divide the data into 80% for training and 20% for testing.
5. Run classifier training procedure on training data.
6. Test the classifier on testing data.
7. Go back to Step 4 to repeat the procedure for different sets of training and testing data, and then take the average performance.

For cross-domain experiments, We train the classifiers again using Steps 1-3 above. We use the alternate domain data for testing.

For each experiment type and classifier type, We have four sets of experiments:

1. Using String-to-word unsupervised filter with alphabetic tokenizer (pre_alpha)
2. Using String-to-word unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha_with_attribute_selection)
3. Using String-to-word unsupervised filter with word tokenizer (pre_word)
4. Using String-to-word unsupervised filter with word tokenizer followed by attribute selection (pre_word_with_attribute_selection)

We use the following settings with String-to-word unsupervised filter:

- IDFTransform: True
- TFTransform: True
- attributeIndices: first-last
- doNotOperateOnFirstClassBasis: False
- invertSelection: False
- lowerCaseTokens: False
- minTermFrequency: 10
- normalizeDocLength: No Normalization
- outputWordCounts: True
- periodicPruning: -1.0
- stemmer: NullStemmer
- stopwords: weka-3-6-10
- useStoplist: False
- wordsToKeep: 1000

For attribute selection, We use:

- evaluator: InfoGainAttributeEval and
- search: Ranker with threshold 0.0

In each experiment set, We collect experiment results on these classifiers:

1. Naive Bayes
2. Naive Bayes Multinomial
3. J48
4. Random Forest with:
    - 10 trees
    - 100 trees
5. IBk (Instance-based k), the K-nearest neighbours classifier with:
    - K=1 and using Euclidean distance
    - K=10 and using Euclidean distance
    - K=1 and using Manhattan distance
    - K=10 and using Manhattan distance
6. SMO (Support vector classifier)

We observe that linguistically informed classifiers (Ling.) using String-to-word unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha_with_attribute_selection) generally give the best in-domain and cross-domain results. We use the classifiers to determine politeness in some blogs. We've used the blog entries from the blogs described in Table 4.2.

4.4 Experimental Results

This section describes the results for the experiments. The percentage figures in the tables for In-domain and Cross-domain experiments denote the percentage of correctly classified instances.

Table 4.2. Blogs used in testing.

Blog no.	url	Description
blog 1	http://blogs.wsj.com/peggynoonan/	A Wall Street Journal Columnist
blog 2	http://www.thefashionpolice.net/	A blog about shopping and style
blog 3	http://www.rogerebert.com/reviews/tyler-perrys-a-madea-christmas-2013	A blog for Movie reviews
blog 4	http://www.thedailybeast.com/	A blog dedicated to breaking news and sharp commentary
blog 5	http://www.blogcatalog.com/blogs/haters-be-hatin	A satirical humour blog
blog 6	http://www.tmz.com/	A celebrity news blog
blog 7	http://www.samizdata.net/	An individualistic perspective blog
blog 8	http://waiterrant.net/	A waiter's rant blog
blog 9	http://www.hecklerspray.com/	A gossip and reviews blog
blog 10	http://wow.joystiq.com/	A gaming blog

4.4.1 In-domain Experiments

Four sets of experiments are done for in-domain analysis using a 5-fold cross-validation.

The correctly classified instances (by %) for In-domain analysis on Wikipedia requests using Bag of Words classifiers are shown in Table 4.3.

The correctly classified instances (by %) for In-domain analysis on Wikipedia requests using Linguistic classifiers are shown in Table 4.4.

The correctly classified instances (by %) for In-domain analysis on Stack Exchange requests using Bag of Words classifiers are shown in Table 4.5.

The correctly classified instances (by %) for In-domain analysis on Stack Exchange requests using Linguistic classifiers are shown in Table 4.6.

4.4.2 Cross-domain Experiments

Four sets of experiments are done for cross-domain analysis using a 5-fold cross-validation.

Table 4.3. In-domain analysis on Wikipedia requests using Bag of Words classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	74.5864%	74.4945%	77.2518%	77.068%
Naive Bayes Multinomial	78.9063%	79.6415%	80.5147%	80.1471%
J48	70.864%	71.2776%	73.7132%	74.3107%
Random Forest (10 trees)	74.5404%	73.6673%	76.7004%	76.6085%
Random Forest (100 trees)	80.7445%	80.193%	80.3309%	79.8254%
iBK (k=1, using Euclidean Distance)	64.8897%	64.1544%	71.2316%	70.6342%
iBK (k=10, using Euclidean Distance)	59.1912%	58.9154%	76.7463%	76.7923%
iBK (k=1, using Manhattan Distance)	63.2813%	63.1434%	71.2316%	69.1636%
iBK (k=1, using Manhattan Distance)	56.4338%	56.1581%	74.6783%	73.4835%
SMO	80.193%	79.8713%	82.307%	82.2151%

The correctly classified instances (by %) for Cross-domain analysis with Wikipedia requests for training and Stack Exchange requests for testing and using Bag of Words classifiers are shown in Table 4.7.

The correctly classified instances (by %) for Cross-domain analysis with Wikipedia requests for training and Stack Exchange requests for testing and using Linguistic classifiers are shown in Table 4.8.

Table 4.4. In-domain analysis on Wikipedia requests using Linguistic classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	74.4485%	74.7702%	77.0221%	76.3327%
Naive Bayes Multinomial	80.7904%	80.239%	80.4688%	80.3309%
J48	72.7022%	72.4265%	75%	73.3456%
Random Forest (10 trees)	72.932%	74.7702%	76.7004%	77.4357%
Random Forest (100 trees)	79.9173%	80.6066%	80.1011%	80.4228%
iBK (k=1, using Euclidean Distance)	64.6599%	64.8438%	71.4614%	71.829%
iBK (k=10, using Euclidean Distance)	60.2022%	59.6967%	76.5165%	76.7923%
iBK (k=1, using Manhattan Distance)	64.6599%	64.8897%	70.5423%	70.5423%
iBK (k=1, using Manhattan Distance)	59.4669%	59.5129%	74.6783%	74.9081%
SMO	81.3879%	80.3768%	82.2151%	81.0202%

The correctly classified instances (by %) for Cross-domain analysis with Stack Exchange requests for training and Wikipedia requests for testing and using Bag of Words classifiers are shown in Table 4.9.

The correctly classified instances (by %) for Cross-domain analysis with Stack Exchange requests for training and Wikipedia requests for testing and using Linguistic classifiers are shown in Table 4.10.

Table 4.5. In-domain analysis on Stack Exchange requests using Bag of Words classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	68.4%	67.7%	71.8%	71.2%
Naive Bayes Multinomial	71.8%	71.55%	72.35%	71.4%
J48	67.15%	65.9%	69.05%	69.75%
Random Forest (10 trees)	69.5%	68.75%	70.15%	69.95%
Random Forest (100 trees)	73.6%	73.45%	72.35%	72.45%
iBK (k=1, using Euclidean Distance)	57.7%	58.85%	65.75%	65.6%
iBK (k=10, using Euclidean Distance)	53.2%	53.15%	66.95%	66.35%
iBK (k=1, using Manhattan Distance)	56.9%	56.85%	65.9%	63.55%
iBK (k=1, using Manhattan Distance)	51.35%	51.95%	64.6%	63.1%
SMO	74.55%	73.5%	74.8%	75.05%

4.4.3 Experiments on web logs

We now use some of the best classifiers We observed in the previous experiments to determine the politeness for blog entries of some popular blogs. The sources for these blogs are discussed in the 'Experiments' section in Table 4.2. For each experiment, We show the probability that the classifier assigns to the blog entries of being 'polite'.

The web-logs are not as ideal as requests for characterizing politeness as requests involve an imposition on the addressee, making them optimal for exploring politeness. However,

Table 4.6. In-domain analysis on Stack Exchange requests using Linguistic classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	68.45%	68.55%	72.4%	72.4%
Naive Bayes Multinomial	72.85%	72.7%	74.6%	74.4%
J48	67.7%	67.25%	71.1%	69.65%
Random Forest (10 trees)	69.15%	67.35%	71.6%	70.25%
Random Forest (100 trees)	74.2%	74.15%	73.35%	72.75%
iBK (k=1, using Euclidean Distance)	58.4%	59.2%	64.9%	63.5%
iBK (k=10, using Euclidean Distance)	55.25%	57.65%	71.2%	71.15%
iBK (k=1, using Manhattan Distance)	58.3%	58.6%	63.8%	63.15%
iBK (k=1, using Manhattan Distance)	53.55%	54.45%	68.8%	67.85%
SMO	72.95%	73.95%	75%	75.95%

some of the politeness strategies used by Danescu-Niculescu-Mizil et al. (Table 4.1) like Greeting, Deference, Hedges, can be applied to blogs too. We use this subset of relevant features for constructing linguistic classifiers for this task. Although the politeness corpus provided by (Danescu-Niculescu-Mizil et al., 2013) characterize the politeness of requests, this is the largest training data that We could find for characterizing politeness, therefore we used this data for this task.

The classification results for blog 1 - blog 5 using Wikipedia requests for training are shown in Table 4.11. For this, we use linguistic classifiers (Ling.) applying String-to-word

Table 4.7. Cross-domain analysis with Wikipedia requests for training and Stack Exchange requests for testing and using Bag of Words classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	63.1%	62.7%	65.35%	64.85%
Naive Bayes Multinomial	66.45%	66%	66.55%	66.5%
J48	62.55%	61.6%	60.4%	61.1%
Random Forest (10 trees)	64.85%	64.95%	64.05%	64.35%
Random Forest (100 trees)	66.2%	66.25%	64.65%	64.65%
iBK (k=1, using Euclidean Distance)	55.05%	55%	62.6%	63.25%
iBK (k=10, using Euclidean Distance)	50.45%	50.25%	61.15%	61.35%
iBK (k=1, using Manhattan Distance)	54.7%	54.3%	61.05%	60.55%
iBK (k=1, using Manhattan Distance)	50.5%	50.35%	58.35%	58.75%
SMO	64.65%	65.55%	65.35%	64.4%

Table 4.8. Cross-domain analysis with Wikipedia requests for training and Stack Exchange requests for testing and using Linguistic classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	64.4%	64.35%	65.55%	65.55%
Naive Bayes Multinomial	66.3%	66%	66.3%	66.5%
J48	61.45%	61.2%	61.05%	60.85%
Random Forest (10 trees)	63.95%	62.3%	65.1%	63.45%
Random Forest (100 trees)	62.85%	63.65%	64.65%	64.65%
iBK (k=1, using Euclidean Distance)	56.75%	56.75%	63.35%	62.5%
iBK (k=10, using Euclidean Distance)	51.85%	51.9%	60.4%	60.7%
iBK (k=1, using Manhattan Distance)	55.15%	55.65%	60.1%	59.6%
iBK (k=1, using Manhattan Distance)	51.35%	50.95%	58.05%	59.35%
SMO	64.9%	64.95%	65.8%	65.45%

Table 4.9. Cross-domain analysis with Stack Exchange requests for training and Wikipedia requests for testing and using Bag of Words classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	60.9375%	61.1213%	66.0386%	65.3493%
Naive Bayes Multinomial	68.9338%	68.75%	65.4412%	65.579%
J48	62.1783%	62.546%	64.0625%	64.7518%
Random Forest (10 trees)	66.9577%	64.568%	66.682%	67.6471%
Random Forest (100 trees)	70.5423%	68.9338%	68.1985%	68.4743%
iBK (k=1, using Euclidean Distance)	57.1691%	57.5827%	62.5919%	62.9596%
iBK (k=10, using Euclidean Distance)	53.3088%	52.8952%	66.5441%	66.4522%
iBK (k=1, using Manhattan Distance)	56.5257%	56.296%	61.2592%	61.6728%
iBK (k=1, using Manhattan Distance)	52.0221%	52.0221%	64.9816%	64.8897%
SMO	71.0938%	71.3235%	68.704%	68.75%

Table 4.10. Cross-domain analysis with Stack Exchange requests for training and Wikipedia requests for testing and using Linguistic classifiers.

Classifier	**pre_alpha**	**pre_word**	**pre_alpha_with attribute selection**	**pre_word_with attribute selection**
Naive Bayes	60.8915%	60.9835%	65.3952%	64.568%
Naive Bayes Multinomial	69.761%	69.6691%	68.0147%	68.2904%
J48	62.9136%	60.6618%	65.7169%	65.2114%
Random Forest (10 trees)	64.0625%	61.9945%	65.3952%	64.0625%
Random Forest (100 trees)	70.0368%	69.0257%	67.4632%	66.9577%
iBK (k=1, using Euclidean Distance)	58.1801%	57.8125%	57.5827%	58.1342%
iBK (k=10, using Euclidean Distance)	55.239%	53.3548%	63.1434%	62.8676%
iBK (k=1, using Manhattan Distance)	58.7776%	57.307%	57.5368%	57.9963%
iBK (k=1, using Manhattan Distance)	53.7224%	53.171%	64.1544%	64.0165%
SMO	70.9099%	71.6452%	69.761%	69.4393%

unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha with attribute_selection).

Table 4.11. Classification results using Wikipedia requests for training for blog 1 - blog 5.

Classifier	**Blog 1**		**Blog 2**		**Blog 3**		**Blog 4**		**Blog 5**	
	polite	impol.	polite	impol.	polite	impol.	polite	impol.	polite	impol.
Naive Bayes	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0
Naive Bayes Multinomial	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
J48	0.0	1.0	0.25	0.75	0.034	0.966	0.25	0.75	0.034	0.966
Random Forest (10 trees)	0.3	0.7	0.3	0.7	0.4	0.6	0.4	0.6	0.3	0.7
Random Forest (100 trees)	0.33	0.67	0.46	0.54	0.47	0.53	0.54	0.46	0.42	0.58
iBK (k=1, using Euclidean Distance)	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0
iBK (k=10, using Euclidean Distance)	0.5	0.5	0.5	0.5	0.5	0.5	0.7	0.3	0.6	0.4
iBK (k=1, using Manhattan Distance)	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
iBK (k=10, using Manhattan Distance)	0.4	0.6	0.5	0.5	0.5	0.5	0.7	0.3	0.6	0.4
SMO	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

The classification results for blog 1 - blog 5 using Stack Exchange requests for training are shown in Table 4.12. For this, we again use linguistic classifiers (Ling.) applying String-to-word unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha with attribute_selection).

The classification results for blog 6 - blog 10 using Wikipedia requests for training are shown in Table 4.13. For this, we use linguistic classifiers (Ling.) applying String-to-word

Table 4.12. Classification results using Stack Exchange requests for training for blog 1 - blog 5.

Classifier	Blog 1		Blog 2		Blog 3		Blog 4		Blog 5	
	polite	impol.	polite	impol.	polite	impol.	polite	impol.	polite	impol.
Naive Bayes	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0
Naive Bayes Multinomial	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
J48	0.0	1.0	0.0	1.0	0.049	0.951	0.0	1.0	0.049	0.951
Random Forest (10 trees)	0.8	0.2	0.8	0.2	0.7	0.3	0.6	0.4	0.8	0.2
Random Forest (100 trees)	0.7	0.3	0.67	0.33	0.49	0.51	0.68	0.32	0.45	0.55
iBK (k=1, using Euclidean Distance)	0.0	1.0	1.0	0.0	1.0	0.0	0.0	1.0	1.0	0.0
iBK (k=10, using Euclidean Distance)	0.7	0.3	0.8	0.2	0.6	0.4	0.4	0.6	0.9	0.1
iBK (k=1, using Manhattan Distance)	1.0	0.0	1.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0
iBK (k=10, using Manhattan Distance)	0.7	0.3	0.8	0.2	0.6	0.4	0.6	0.4	0.4	0.6
SMO	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha with attribute_selection).

The classification results for blog 6 - blog 10 using Stack Exchange requests for training are shown in Table 4.14. For this, we again use linguistic classifiers (Ling.) applying String-to-word unsupervised filter with alphabetic tokenizer followed by attribute selection (pre_alpha with attribute_selection).

Table 4.13. Classification results using Wikipedia requests for training for blog 6 - blog 10.

Classifier	Blog 6		Blog 7		Blog 8		Blog 9		Blog 10	
	polite	impol.	polite	impol.	polite	impol.	polite	impol.	polite	impol.
Naive Bayes	0.001	0.999	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0
Naive Bayes Multinomial	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
J48	0.0	1.0	0.667	0.333	0.961	0.039	0.081	0.919	0.667	0.333
Random Forest (10 trees)	0.5	0.5	0.5	0.5	0.6	0.4	0.4	0.6	0.5	0.5
Random Forest (100 trees)	0.4	0.6	0.35	0.65	0.53	0.47	0.37	0.63	0.56	0.44
iBK (k=1, using Euclidean Distance)	1.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0
iBK (k=10, using Euclidean Distance)	0.3	0.7	0.4	0.6	0.5	0.5	0.0	1.0	0.6	0.4
iBK (k=1, using Manhattan Distance)	1.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0
iBK (k=10, using Manhattan Distance)	0.3	0.7	0.5	0.5	0.4	0.6	0.4	0.6	0.7	0.3
SMO	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

4.5 Related Work

Politeness is a source of pragmatic enrichment, social meaning, and cultural variation (Danescu-Niculescu-Mizil et al., 2013). The social-norm view of politeness reflects the historical understanding of politeness generally embraced by the public within the English-speaking world. It assumes that each society has a particular set of social norms consisting of more or less explicit rules that prescribe a certain behavior, a state of affairs, or a way of thinking in a context. A positive evaluation (politeness) arises when an action is in congruence with

Table 4.14. Classification results using Stack Exchange requests for training for blog 6 - blog 10.

Classifier	**Blog 6**		**Blog 7**		**Blog 8**		**Blog 9**		**Blog 10**	
	polite	impol.	polite	impol.	polite	impol.	polite	impol.	polite	impol.
Naive Bayes	1.0	0.0	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0
Naive Bayes Multinomial	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0
J48	0.958	0.042	0.0	1.0	0.0	1.0	0.75	0.25	0.0	1.0
Random Forest (10 trees)	0.7	0.3	0.7	0.3	0.8	0.2	0.5	0.5	0.5	0.5
Random Forest (100 trees)	0.73	0.27	0.65	0.35	0.73	0.27	0.47	0.53	0.71	0.29
iBK (k=1, using Euclidean Distance)	1.0	0.0	0.0	1.0	1.0	0.0	0.0	1.0	1.0	0.0
iBK (k=10, using Euclidean Distance)	0.3	0.7	0.2	0.8	0.5	0.5	0.273	0.727	0.5	0.5
iBK (k=1, using Manhattan Distance)	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	1.0	0.0
iBK (k=10, using Manhattan Distance)	0.5	0.5	0.3	0.7	0.6	0.4	0.273	0.727	0.4	0.6
SMO	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0	1.0	0.0

the norm, a negative evaluation (impoliteness = rudeness) when action is to the contrary (Fraser, 1990).

Manuals of etiquette contain aphorisms that reveal quickly this underlying assumption. The 1872 version of *Ladies' Book of Etiquette and Manual of Politeness* ((Hartley, 1872)) offers a variety of rules intended to govern polite discourse.

The conversational-maxim perspective on politeness relies principally on the work of Grice (1967, published 1975 (Grice, 1975)) in his now-classic paper 'Logic and conversation'.

Grice argued that conversationalists are rational individuals who are, all other things being equal, primarily interested in the efficient conveying of messages (Fraser, 1990).

By far, the most popular view of politeness is the face-saving view by Brown and Levinson. The aspects of their theory have have been explored from game-theoretic perspectives and implemented in language generation systems for interactive narratives, cooking instructions, translation, spoken dialog and subjectivity analysis, among others.

In recent years, politeness has been studied in web environments. Politeness variations across different social groups and different media types have been researched. Danescu-Niculescu-Mizil, Sudhof, Jurafsky, Leskovec and Potts (Danescu-Niculescu-Mizil et al., 2013) pursue similar goals and construct annotated data orders of magnitude larger for a more reliable study of politeness strategies. My project uses this annotated data for performing classifications.

Some researchers have also focused on domain-specific textual cues to study how language relates to power and status in the context of social networking and workplace discourse .

4.6 Conclusions and Future Work

We train classifiers employing different machine learning algorithms and using unsupervised and supervised filters and perform experiments in in-domain and cross-domain environments. We use linguistic features and find an improvement in the performance of the classifiers. We observe that in general, SMO classifiers tend to give the best performance for text classifications. The performance of SMO algorithms improve when we apply String-to-word unsupervised filter with alphabetic tokenizer followed by attribute selection.

For in-domain experiments, in classifiers that don't use attribute selection, using linguistic features for training improves the performance by 2-3%. This improvement reduces to 0-1% when attribute selection is also used while training the classifiers. In almost every case, using linguistic features registers an improvement in the performance.

These classifers perform well in cross-domain settings too. When training on Wikipedia and tested on Stack Exchange, the best performance is 65.8%. When training on Stack Exchange and tested on Wikipedia, the best performance is 71.64%. SMO classifiers using linguistic features show these best results.

We use the better performing classifiers to determine the politeness for blog entries of some popular blogs. We observe that a majority of classifiers classify Blogs 1, 2, 3, 4, 6, 7, 8 and 10 as polite, and Blogs 5 and 9 as impolite.

In future, we plan to employ AdaBoosting techniques and infer if the performance improves for these experiments. We also plan to use domain adaptation techniques and use these classifiers to determine politeness in domains other than web-logs.

CHAPTER 5

COMPARATIVE ANALYSIS OF DIFFERENT APPROACHES TO SENTIMENT ANALYSIS OF TWEETS [1]

This chapter develops a variety of techniques to process data from the social network website Twitter, and generate useful information from it. An application of twitter data processing is designed in which the tweets for the year 2012 are processed to predict the rating (on a scale of 0 - 10) of movies released in 2012. An outline of the approaches used for this task is presented, and the conclusions drawn from the results are described.

5.1 Motivation

Social networks have changed the way we consume media, and interact with it. Widespread use of the social networks like Facebook, Twitter, Google+, Weibo and RenRen has made it possible to analyze any subject of popular interest with the data collected from these networks. Twitter passed the 500 million users mark back in July 2012, and churns out 750 tweets per second from its collective user base (Hermens, 2014). It has become unavoidable in our current cultural landscape. Unlike the traditional media that does not possess the proper infrastructure to support interaction with mass society, this microblogging service has emerged as an active way to discuss things on a massive scale, and it allows to get a sense of mass sentiment about almost any popular subject. In particular, movies and television programs are heavily discussed, and commented upon on Twitter. In this article, we describe an application of how we can mine tweets related to movies to generate useful predictions about the quality and performance of movies.

[1] Authors: Shagun Jhaver, Ranjitha Shadakshari and Shweta Menon

5.2 Introduction

We've a collection of more than 350 GB of tweets from the year 2012. We parse these tweets to look for the tweets related to the movies released in 2012. We then conduct a sentiment - analysis on these relevant tweets to predict the rating of each movie on a scale of 1 to 10. As the last step, we compare these ratings with those from popular movie rating databases like IMDB.

We've used a stand-alone hadoop System to test the code. Later, we'll run the code on HDFS to be obtained from the lab. The input files contain the tweets in JSON format. We have 12 .tar files for each month of the year. Each .tar file has .json.bz2 files (zipped) containing JSON files with tweets information. We use a file 'Movies.txt' to store the keywords to be used via DistributedCache. These keywords are the list of movies along with their IMDB ratings released in 2012. We've retrieved this list from IMDB website.

5.3 Filtering Tweets

We use Map Reduce jobs to filter the tweets related to movies released in 2012. Our input files are in .bz2 format, and Map Reduce works very efficiently for such files. These compressed files are automatically extracted, data is split and sent across the different slave nodes for processing. The first Map Reduce job parses the tweets using the json-simple library and cleans the tweets, so that only the tweets which are relevant for further processing are filtered.

The raw tweets data are very noisy. There are a large number of irregular words and non-English characters. We reduce the feature space by observing that each tweet is associated with many columns like username, timestamp, location etc., but we only require the text message for our analysis. All nonEnglish tweets, and irregular character symbols

are removed. For example, ?@!#$#%$%$@... MH3G becomes MH3G after filtering. We also reduce the number of letters that are repeated more than twice in all words. For example the word looooove becomes love after reduction.

The second Map Reduce job uses the movies obtained from the movies.txt file (sent to the slave nodes using DistributedCache) to create a HashMap of movie titles. The Mapper checks whether the input tweet references any of the movie titles in the hashmap. In case a tweet is related to a movie, the movie name is removed from the tweet, and the output of the form <Movie, Modified Tweet> is sent to the reducer. Removing movie title from the tweet ensures that the sentiment analysis for the movie is not disturbed when the movie title has positive or negative connotations, eg, the Fantastic Mr. Fox.

5.4 Classifying Tweets

We now have a collection of relevant tweets against their corresponding movies. We use three different approaches to classify each tweet. This section describes each of these approaches

5.4.1 Classifying using list of positive and negative words

Tweets are assigned a positive or negative polarity based on the occurrence of positive or negative words in the tweet. We have a collection of positive words and negative words list stored in distributed cache. Each tweet is compared against the list. If a positive word is detected in the tweet, the corresponding tweet will be given a positive polarity with a value of 10. Similarly, a negative polarity with a value of 0 will be assigned to tweets containing any term in the list of negative words.

5.4.2 Using Distant Supervision (Sentiment140 api)

In this approach, we utilize the sentiment140 API (sen, 2014) to classify twitter messages. Sentiment140 is an open source Application Programming Interface developed by Stanford

University, which automatically classifies the sentiment of Twitter messages. It classifies a given query term as either positive, negative or neutral.

Sentiment140 algorithm uses distant supervision, in which the training data consists of tweets with emoticons. The emoticons serve as noisy labels. For example, :) in a tweet indicates that the tweet contains positive sentiment and :(indicates that the tweet contains negative sentiment. With the help of the Twitter API, it is easy to extract large amounts of tweets with emoticons in them. Classifiers trained on emoticon data are run against a test set of tweets.

We use sentiment140 algorithm (Sentiment140 API) to find the sentiment of the tweets based on polarity obtained in the response file. We generate a bulk request using the json-lib library, and then send this request to http://www.sentiment140.com/api/bulkClassifyJson. We parse the response file to collect rating corresponding to each tweet (sen, 2014).

5.4.3 Using customized Mahout Classifier

Classification algorithms can be used to automatically classify documents, images, implement spam filters and in many other domains. In this case, we used Mahout to classify tweets using the Naive Bayes Classifier. The algorithm works by using a training set which is a set of documents already associated to a category. Using this set, the classifier determines for each word, the probability that it makes a document belong to each of the considered categories. To compute the probability that a document belongs to a category, it multiplies together the individual probability of each of its word in this category. The category with the highest probability is the one the document is most likely to belong to.

We consider two categories - positive and negative. We build a classication model based on the training set. The new set of tweets are classified using this model as positive or

negative category. This classified set of tweets are used in the next stage to obtain the average rating of the movie.

5.5 Calculating Average rating

In this stage, we have as input <movie title, rating> as a result of the classification described above. We write a MapReduce code that sends a list of ratings corresponding to each movie to the reducer. The reducer averages over these ratings to calculate the average rating. This is concatenated with the standard IMDB rating stored in Distributed cache, and the final output is of the form:

<Movie-Name | Tweet Based Rating | IMDB Rating>

5.6 Conclusion

Analyzing the results of this project using different test input-sets, we have observed that out of the 3 approaches for classification, Machine learning technique i.e. Naive Bayes classification to determine the movie rating is more accurate compared to other 2 approaches. We should be able to improve the efficiency of this approach by tweaking the classifying model. We find that the second approach of sending the bulk tweets over the network to the sentiment140 API is not efficient for standalone systems. The first approach is a basic approach and can implement efficiently with good algorithm and combines unigrams and bigrams.

However, we'd like to avoid giving a conclusive verdict on the relative efficiency of these approaches because we've only analyzed small-sized inputs. Using larger data-sets on these different approaches should help us comment more concretely on the performances of the three approaches.

5.7 Looking Ahead

We'd test each of these approaches on the same, larger data sets to obtain figures describing their relative efficiencies. Right now, the classifier is being run on the data on a single machine. We plan to modify this code so that the classifier can work on data on the distributed file system. We'd also look into whether the classification job should also be distributed on to the slave nodes. We'd tweak the algorithm building the classifying model, and then run tests to see which features give the best results. We'd also look into applications other than movie ratings that we can use these methods for. Besides, we'd consider getting the tweets indexed using Apache SOLR for efficient processing.

REFERENCES

(2014). *Sentiment 140 api*. `http://help.sentiment140.com/api`.

(2014). Wikipedia: C4.5/ algorithm. `http://en.wikipedia.org/wiki/C4.5_algorithm`.

(2014). Wikipedia: Sequential minimal optimization. `http://en.wikipedia.org/wiki/Sequential_minimal_optimization`.

Aggarwal, C. C. (2012). The multi-set stream clustering problem. In *SDM*, pp. 59–69. SIAM.

Aggarwal, C. C. (2014). The setwise stream classification problem. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 432–441. ACM.

Arslan, A. N. E. O. (1999). An efficient uniform-cost normalized edit distance algorithm. *String Processing and Information Retrieval Symposium and International Workshop on Groupware*, 8,15.

Bar-Yossef, Z. J., T. S. Krauthgamer, and R. Kumar (2004, October). R. In *Approximating edit distance efficiently*, 2004. Proceedings. 45th Annual IEEE Symposium on , vol., no., pp.550,559,Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on , vol., no., pp.550,559, pp. 17–19. Foundations of Computer Science.

Brown, P. and S. Levinson (1978). Universals in language usage: Politeness phenomena. In E. Goody (Ed.), *Questions and Politeness*, pp. 56–289. Cambridge: Cambridge University Press.

Brown, P. and S. Levinson (1987). *Politeness: Some Universals in Language Usage*. Cambridge: Cambridge University Press.

Bunke, H. (1997). On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters 18*(8), 689–694.

Chandra, S., L. Khan, and F. B. Muhaya (2011). Estimating twitter user location using social interactions–a content based approach. In *IEEE Third International conference on Social Computing (SocialCom)*, pp. 838–843. IEEE.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein: (1990). *Introduction to Algorithms*. Cambridge, MA: MIT Print.

Danescu-Niculescu-Mizil, C., M. Sudhof, D. Jurafsky, J. Leskovec, and C. Potts (2013). A computational approach to politeness with application to social factors. *ACL*.

de la Higuera, C.; Mico, L. (2008, April). A contextual normalised edit distance. *Data Engineering Workshop*, 354,361.

Dietterich, T. G. (2000). Ensemble methods in machine learning. In *Multiple classifier systems*, pp. 1–15. Springer.

Domingos, P. and G. Hulten (2000). Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 71–80. ACM.

Dyer, K. P., S. E. Coull, T. Ristenpart, and T. Shrimpton (2012). Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 332–346. IEEE.

Ekanakake, J., H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox (2010). Twister: a runtime for iterative mapreduce. In *19th ACM International Symposium on High Performance Distributed*. Computing (HPDC.

Fraser, B. (1990). *Perspectives on Politeness*. Journal of Pragmatics.

Fu, K. S. . (1982). *Syntactic Pattern Recognition and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Fuad, M. M. M.; Marteau, P.-F. (2008, June). The extended edit distance metric. *Content-Based Multimedia Indexing*.

Gaber, M. M., A. Zaslavsky, and S. Krishnaswamy (2005). Mining data streams: a review. *ACM Sigmod Record 34*(2), 18–26.

Glymour, C., D. Madigan, D. Pregibon, and P. Smyth (1997). Statistical themes and lessons for data mining. *Data mining and knowledge discovery 1*(1), 11–28.

Goffman, E. (1971). *Relations in Public: Microstudies of the Public Order*. New York Harper and Row.

Grice, H. P. (1975). *Logic and Conversation*.

Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten (2009). The weka data mining software: an update. *ACM SIGKDD explorations newsletter 11*(1), 10–18.

Hall, P. A. V. and G. R. Dowling (1980, December). Approximate string matching. *ACM Comput Surveys 12*, 381–402.

Hanada, H., A. Nakamura, and M. Kudo (2011). A practical comparison of edit distance approximation algorithms. *Granular Computing (GrC)*, 231–236.

Hansen, L. K. and P. Salamon (1990). Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence 12*(10), 993–1001.

Hartley, F. (1872). *The Ladies' Book of Etiquette: And Manual of Politeness: A Complete Hand Book for the Use of the Lady in Polite Society: Containing Full Directions for Correct Manners, Dress, Deportment, and Conversation ... and Also Useful Receipts for the Complexion, Hair, and with Hints and Directions for the Care of the Wardrobe*.

He, Q., C. Du, Q. Wang, F. Zhuang, and Z. Shi (2011). A parallel incremental extreme svm classifier. *Neurocomputing 74*(16), 2532–2540.

Hermens, C. (2014). Tweet-a-new-way.

Jain, S. and A. L. N. Rao (2013, October). A comparative performance analysis of approximate string matching. *International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 3*, 2278–3075.

Juarez, M., S. Afroz, G. Acar, C. Diaz, and R. Greenstadt (2014). A critical evaluation of website fingerprinting attacks. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS 2014)*.

Klinkenberg, R. (2003). Concept drift and the importance of examples. In *Text mining–theoretical aspects and applications*. Citeseer.

Kolter, J. Z. and M. A. Maloof (2007). Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research 8*, 2755–2790.

Lee, L. (1999). Measures of distributional similarity. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pp. 25–32. Association for Computational Linguistics.

Lee, W. and S. J. Stolfo (1998). Data mining approaches for intrusion detection. In *Usenix Security*.

Liberatore, M. and B. N. Levine (2006). Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 255–263. ACM.

Malewicz, G., M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and C. G. (2010). Pregel: a system for large-scale graph processing. In *International Conference on Management of.* data (SIGMOD.

Marzal, A. and E. Vidal (1993). Computation of normalized edit distance and applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence 15*(9), 926–32.

Masek, W. J. and M. S. Patterson (1980, February). A faster algorithm computing string edit distances. *J. Comput. Syst. Sci. 20*, 18–31.

Masud, M. M., Q. Chen, L. Khan, C. Aggarwal, J. Gao, J. Han, and B. Thuraisingham (2010). Addressing concept-evolution in concept-drifting data streams. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pp. 929–934. IEEE.

Masud, M. M., J. Gao, L. Khan, J. Han, and B. Thuraisingham (2010). Classification and novel class detection in data streams with active mining. In *Advances in Knowledge Discovery and Data Mining*, pp. 311–324. Springer.

McCallum, A. and K. Nigam (1998),). A comparison of event models for naive Bayes text classification. *AAAI/ICML-98 Workshop on Learning for Text Categorization*.

Meier, A. J. (1999). Defining politeness: Universality in appropriateness.

Mitchell, T. (1997). *Machine Learning* (1 ed.). Science/Engineering/Math; (March 1: McGraw-Hill.

nlp.stanford.edu (2014).

Parveen, P., P. Desai, B. M. Thuraisingham, and L. Khan: (2013). Mapreduce-guided scalable compressed dictionary construction for evolving repetitive sequence streams. *CollaborateCom*, 345–352.

Pereira, R., M. Azambuja, K. Breitman, and M. Endler (2010). An architecture for distributed high performance video processing in the cloud. In *3rd International Conference on Cloud*. computing (Cloud.

Ristad, E. and P. n. Yianilos (1998). Learning string-edit distance. *Yianilos IEEE Transactions on Pattern Analysis and Machine Intelligence 20*(5), 522–32.

Robles-Kelly, A. and E. Hancock. (2005). Graph edit distance from spectral seriation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 27*(3), 365–78.

Sankoff, D. and J. B. Kruskal (1983a). *Time Warps, String Edits, and Macro-molecules: The Theory and Practice of Sequence Comparison*. Reading, MA: Addison-Wesley.

Sankoff, D. and J. B. Kruskal (1983b). *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Inc: Addison-Wesley Publishing Company.

Sellers, P. H. (1980). The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms 1*, 359–373.

Syed, N. A., H. Liu, and K. K. Sung (1999). Handling concept drifts in incremental learning with support vector machines. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 317–321. ACM.

Syverson, P. F., D. M. Goldschlag, and M. G. Reed (1997). Anonymous connections and onion routing. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pp. 44–54. IEEE.

Vidal, R. (2010). A tutorial on subspace clustering. *IEEE Signal Processing Magazine 28*(2), 52–68.

Wagner, R. A. and M. J. Fischer (1974, January). The string-to-string correction problem. *J. Assoc 21*(1), 168–173.

Wang, H., W. Fan, P. S. Yu, and J. Han (2003). Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 226–235. ACM.

Wei, J. (2004, March). Markov edit distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on , vol 26*(3).

Yan, C., X. Yang, Z. Yu, M. Li, and X. Li. (2012). Incmr: Incremental data processing based on mapreduce. *IEEE CLOUD, page*, 534–541.

Yang, Y. P. and T. Pavlidis (1990, November). Optimal correspondence of string subsequences. *IEEE Trans Patt. Anal. Machine Intell 12*(11), 1080–1087.

VITA

Shagun Jhaver was born in Neemuch, India. After completing his schoolwork at Modi Public School, Kota in Rajasthan, India, Shagun entered Indian Institute of Technology Bombay in Mumbai, India in 2006. During the summer of 2009, he worked as a researcher at Institut national de recherche en informatique et en automatique (INRIA) Paris-Rocquencourt, France. He received a Bachelor of Technology with a major in Electrical Engineering from Indian Institute of Technology Bombay in May 2010. During the following two years, he was employed as a software developer at Future Group in Mumbai, India. In August, 2012, he entered the Graduate School at The University of Texas at Dallas.