ADAPTIVE DOCUMENT DISCOVERY FOR VERTICAL SEARCH ENGINES

by

Wojciech Piaseczny
B.A.Sc., Simon Fraser University, 2005

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in the
School of Engineering Science
Faculty of Applied Sciences

APPROVAL

Name:	**Wojciech Piaseczny**
Degree:	**Master of Applied Science**
Title of Thesis:	**Adaptive Document Discovery for Vertical Search Engines**

Examining Committee:

Dr. Marek Syrzycki
Chair
Professor of Engineering Science, SFU

Dr. Bozena Kaminska
Senior Supervisor
Professor of Engineering Science, SFU

Dr. Lesley Shannon
Supervisor
Assistant Professor of Engineering Science, SFU

Dr. Anoop Sarkar
Examiner
Assistant Professor of Computing Science, SFU

Date Defended/Approved: April 20, 2011

ABSTRACT

Vertical search engines attempt to aggregate all available online data for a specific vertical into a normalized and structured data model. There are two common strategies for aggregating data: 1) data feeds, and 2) web crawling. Data feeds use source-specific translation rules to collect structured data, but require the source to specifically expose the data. Web crawling collects data through the same interface that users view it, which requires additional work to identify and extract the relevant content from unstructured or semi-structured text. Generalizing these tasks across many websites is difficult because each website presents content in its own arbitrary way. This thesis proposes a strategy for identifying relevant content across many websites with improved accuracy.

Many well known statistical document classification algorithms can distinguish between classes of documents with high accuracy. These algorithms fail when test data is significantly different than training data, as is often the case in the vertical search context. This thesis adaptively builds website-specific document classifiers to avoid common classification failure conditions. Training data is selected dynamically by exploiting common user interface patterns. The results obtained here demonstrate that using adaptive document classifiers improves accuracy with minimal performance costs.

Keywords: Adaptive document classification; supervised learning; clustering; vertical search; entity extraction.

DEDICATION

This thesis is dedicated to my wife, Sheena. I love you.

ACKNOWLEDGEMENTS

I would like to thank Prof. Bozena Kaminska for her guidance, encouragement and patience during my studies. I would like to thank the members of my thesis committee: Prof. Anoop Sarkar, Prof. Lesley Shannon, and Prof. Marek Syrzycki.

I would like to thank my co-workers Sean Whitley, John Schaub, Tristen Georgiou, and Jake Pan for their feedback on my research. I would also like to thank Move, Inc. for its support of my work.

Finally, I would like to thank my family. My beautiful, kind, patient, and loving wife Sheena gave me the time and encouragement I needed to complete this thesis. My son, Mikolaj, gave me the motivation to wrap things up. My parents gave me inspiration to start this thesis and helped me persevere along the way, qualities that I hope to pass on to my son.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

GLOSSARY

Document Classification	Assigning a document to one or more categories using supervised or unsupervised learning.
Entity Extraction	Identifying atomic elements in text into a predefined data model.
Indexed Data	A representation of any data that improves search or lookup performance.
Rich Document	The content that users target in a vertical search engine. E.g., real estate listings on a real estate listing search website.
Structured Data	Data organized into structure so that it can be identified.
Supervised Learning	Machine learning algorithms that infer a function from labelled input training data.
Unstructured Data	Data with no identifiable structure.
Unsupervised Learning	Machine learning algorithms that determine how data is organized without the use of labelled training examples.
Vertical	Specific industry, topic, or type of content.

1: INTRODUCTION

The internet has grown into a vast collection of information on most imaginable subjects (Murray et al., 2000). Finding information relevant to one's specific needs is manageable using tools like Google's search engine when the relevance of the data is the same for all web users. Data publishers gain trust within a community by producing high quality data and search engines recognize this trust and use it to influence relevance. For highly structured data, relevance is user-specific rather than community based. For example, a student and a retiree would likely purchase very different vehicles even though they may go to the same dealership to buy them. Much of the internet's richest data is highly structured and is not indexed effectively by large-scale search engines for several reasons relating to the data's structure:

1) the structure is obvious to a human user but not to a program that indexes the web,

2) the structure is vertical-specific, and

3) the structure is website specific, even when dealing with a single vertical.

Vertical search engines expose subsets of these rich documents in useful ways for their chosen domain. Unlike general purpose search engines, most vertical search engines do not crawl the web to find their data. Even with the simplifications of being within a single vertical, it is very difficult to find and then index rich documents using static algorithms. Each website presents its documents with varying amounts of verboseness and unique terminology. However, simplifying this problem to a single website presents an obvious solution to finding these rich documents. Document classification is a well studied and understood application of machine learning algorithms. Good training conditions are the key to building high precision and high recall classifiers. Within a single website, there is significant similarity between one rich document and the next. The structure (order of elements, terminology, etc.) will be consistent because:

1) human users have to use the website and understand what they are looking at, and

2) for sites with a large number of documents, each document is templated.

This scenario is ideal for building a high precision, high recall classifier to identify rich documents.

Generalizing this solution requires some observations about modern websites. Every website with a large number of documents provides some search interface to find documents. For example, a real estate listings search interface typically allows a user to specify their location, price range, home size, and some home features. The result of a search provides a summary view of the data, listing only key elements for several documents at a time, as well as a link to deeper information for each document. Note that the summary view will have a high frequency of key data fields (one for each document). Continuing the real estate listings example, the summary view will likely include the address, price, and size for each home that matches the search criteria. The summary view would likely not include information about each home's garage because this is not considered a key data field. If a website does contain summary views, then the links to the deep document views will share significant structural similarities. Again using the real estate listings example, one home's detail page may have a URL like: example.com/properties/details?listing_id=1234, while another could be: example.com/properties/details?listing_id=4321. Just as deep documents within a single website share significant structural similarity, summary documents across all websites (within the vertical) share a small set of high frequency data elements and labels. This creates another scenario that is ideal for building high accuracy classifiers.

The objective of this work is to develop a high accuracy document classifier for rich documents across all websites within a vertical. Certainty about a document's classification simplifies the task of indexing the document, both when the data is structured and unstructured. In the case of wanting structured data, entity extraction also benefits by knowing a document's classification and can make otherwise unsafe assumptions about data elements. For example, a document classified as a real estate listing likely includes a price. If there is only one price in the document, it can safely be assumed to be the listing's price. Without knowing the classification, this assumption cannot be made.

Traditional static document classifiers work well when the training context is consistent with the “real world” the classifier later lives in. This is not the case for rich documents due to their variability between across websites. This thesis proposes a novel strategy named adaptive document classification for identifying domain-specific structured data by exploiting consistency requirements both within a single website, and across all websites within a vertical. Adaptive document classification uses two levels of document classifiers: 1) a general-purpose summary view classifier (whose training examples are manually collected by a domain expert), and 2) an adaptive, website-specific rich document classifier trained with dynamically collected examples. Once the data is identified, it can be indexed using traditional strategies or more intelligent domain-specific entity extraction. The strategy is demonstrated using real estate as the example domain.

The remainder of this thesis is organized into five chapters. Chapter 2 presents prior work related to this thesis, starting with an introduction to supervised, unsupervised, and active machine learning algorithms. Document classification is introduced next as an application of machine learning. Finally, works similar to this one are discussed. Chapter 3 details this thesis’ system design for a vertical search engine powered by crawled data. The focus of the design discussion is on adaptive document classification, which is the novel piece introduced here. Chapter 4 presents the subset of the design that is included in this thesis’ sample implementation. The sample implementation demonstrates the novelty introduced by this thesis, adaptive document classification, but does not implement entity extraction or search/indexing. While the design chapter includes general items like “machine learning algorithm”, the implementation chapter lists the specific algorithms used. Chapter 5 discusses the set of experiments used to test the implementation, including all results. Chapter 6 summarizes this thesis, including its successes, limitations and future work.

2: RELATED WORK

The goal of this thesis is to produce high accuracy, context-sensitive document classifiers for rich documents where traditional classifiers do not perform well. This chapter introduces the necessary background and prior research relevant to this thesis.

2.1 Statistical Learning

Statistical learning commonly refers to learning from data (Nilsson, 1998; Hastie et al., 2001). In the computing science domain, this is also called machine learning (ML). An ML algorithm is given a set of input data from which it derives relationships that it applies to future input. The opposite of ML (in the context of computing science) is an explicit encoding of a problem and instructions for how to solve it. This opposite is impossible (or at least highly impractical) to implement in many real world applications, resulting in the need for ML. Some tasks cannot be defined well except by example. Some tasks involve data that has deep correlations that are difficult for a human to extract, understand, and/or describe. Some tasks change over time due to changes in their environments, new information becoming available, new expertise being acquired, and/or changes to the inputs. A statistical approach assumes that these difficulties are all encoded in a set of training data, or can be included in a future set. Current research indicates that this is a very good assumption (Cortes and Vapnik, 1995; Joachims, 1998).

ML accuracy is often measured with two metrics: precision and recall (van Rijsbergen, 1979). Precision is defined as:

$$precision = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + number\ of\ false\ positives}$$

And recall:

$$recall = \frac{number\ of\ true\ positives}{number\ of\ true\ positives + number\ of\ false\ negatives}$$

These metrics require the number of true and false positives and negatives to be known, so are only applicable in training situations. Unfortunately, training is always done with a subset of the data, so the results are only as representative as the data itself. Subsets are used for two reasons:

1) subsets are usually all that is known, and

2) there are practical performance implications to using too much training data.

2.1.1 Learning Context

ML has a simple task: given some input, predict the output. This can be represented as learning a function $f.$ The learner proposes a solution $g: X \to Y$ from all possible solutions G, which maps $x \in X \to y \in Y$, where X is the set of all possible inputs and Y is the set of all possible outputs. The difference between g and f results in some loss: $loss(x, g(x), y),$ where $y = f(x)$. The ultimate goal is to minimize this loss function for all inputs. For some algorithms it is more convenient to represent the learning problem in the form of a conditional probability model $g(x) = P(y|x)$, or f as the joint probability model $f(x,y) = P(x,y)$.

2.1.1.1 Supervised Learning

Learning based on training data is called supervised learning (Vapnik, 2000; Hastie et al., 2001). The training data is a set of pairs of the form $\{(x_1, y_1), \dots, (x_n, y_n)\}$, where both x and y can be vectors, and each (x_i, y_i) pair is an example of the desired input-output mappings. The learning algorithm attempts to generalize the relationship between input-output pairs and infers some function, h, as a result. When y is discrete the learning task is called classification, and when continuous it is called regression. The strict distinguishing factor of a supervised learning algorithm is that it requires training data. Training data has several important implications:

1) all classes of outputs must be known ahead of time, and

2) unseen classes of inputs must be handled in reasonable ways.

The size and quality of training data has significant effects on the accuracy of an algorithm. Given a reasonably good algorithm with some training data, the fastest (and often easiest) way to improve accuracy is to choose a better training set. Some algorithms have tuning parameters that can be optimized. These optimizations usually affect the variance and bias of an algorithm. Given an algorithm with several sets of training data, it is said to be biased against some input x if it systematically predicts the wrong output value for x regardless of which training set was used. Conversely, an algorithm that makes a different prediction for x using different training sets is said to have high variance. The ideal algorithm has low bias and low variance, and fittingly, the error rate of an algorithm is related to the sum of these two values.

There is a trade off between flexible (low bias, high variance) and inflexible (high bias, low variance) either set by parameters or built into each algorithm. Flexible algorithms can adapt to data well, but sometimes so well that the underlying concept is not generalized. This is called over-fitting. Inflexible algorithms avoid the over-fitting problem, but risk missing good information from the training set. Simple problems can be solved by inflexible algorithms with even a small amount of data. More complex problems need flexible algorithms and a lot more examples to extract relationships from the training data.

Some problems can appear to be more complex than they really are. Input training data is a collection of samples, where each sample, x, can be a vector. High dimensionality of x can introduce a lot of data that does not contribute to f, the function we are trying to discover. As a domain expert preparing the training data one might know this, but to an algorithm, all dimensions of the input are equally likely to contribute to the output.

2.1.1.1.1 Semi-Supervised Learning

Semi-supervised learning learns how to act given an observation (Zhu and Goldberg, 2009). Every action has some impact, and a supervisor provides feedback about the impact that guides the learning algorithm. The learner is allowed to choose which data to use for training, which usually results in needing fewer labelled examples overall at the risk of focusing on the wrong, irrelevant, or even invalid examples. This type of learning is useful when there is an abundance of unlabelled data that is expensive to label.

A popular (and effective) class of semi-supervised learning algorithms uses Minimum Marginal Hyperplane methods, including Transductive Support Vector Machines (SVMs). The distance between each unlabeled example and a separating hyperplane is measured, and the examples closest to the hyperplane are assumed to be the strongest indicators so are chosen for labelling. This creates a new hyperplane and the process repeats iteratively. SVM is discussed in depth later in this chapter.

2.1.1.2 Unsupervised Learning

Learning without training data is called unsupervised learning (Hofmann et al, 1999). Unlike supervised learning, the class of outputs is unknown at training time. From a theoretical point of view, supervised learning defines the effect a set of inputs has on a set of outputs. Unsupervised learning assumes that all observations are caused by a set of latent variables. This distinction is shown in Figure 1.

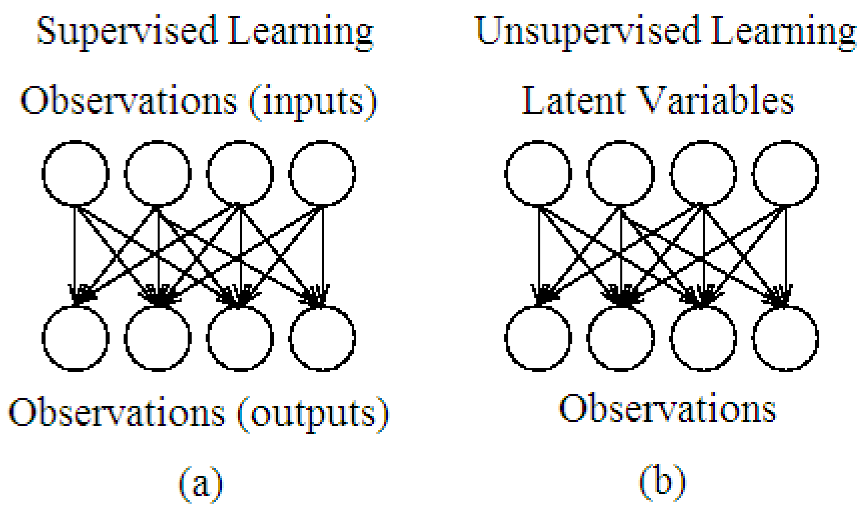


Figure 1: (a) In Supervised Learning all outputs are assumed to be caused by a set of inputs. (b) In Unsupervised Learning all outputs are caused by a set of latent variables. (Valpola, 2000)

As with supervised learning, the quality of input data is essential to learn useful relationships. A practical advantage of unsupervised learning over supervised learning is the complexity of relationships that it can learn. Supervised learning tries to find the relationship between two sets of data (input and output), and the difficulty of this task increases exponentially with each hierarchical step between the sets. In unsupervised learning, an algorithm can learn one hierarchical step at a time, making the cost per step linear rather than exponential. Unsupervised learning can be used to solve supervised problems by assuming that a set of latent variables causes both the input and the output, as shown in Figure 2.

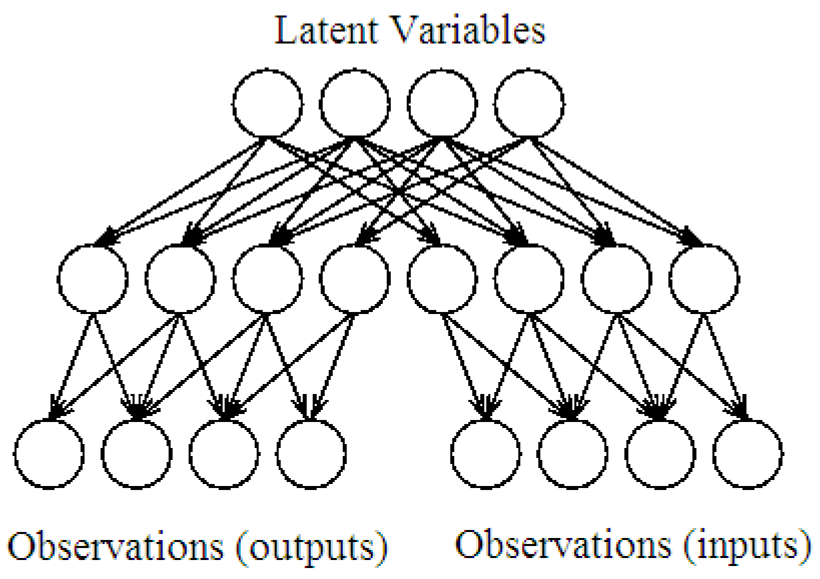


Figure 2: Unsupervised learning can discover hierarchical relationships (Valpola, 2000)

Data mining is among the practical applications of unsupervised learning (Chakrabarti, 2000; Berkhin, 2002). When used to 'discover' new relationships within some data set, it is difficult to judge an algorithm's effectiveness (other than 'good' or 'bad'). Many applications of unsupervised learning are equally difficult to evaluate, which has led to the introduction of many algorithms based on many different heuristics, with none provably better than others.

2.1.2 Learning Algorithms

This section introduces a selection of supervised and unsupervised learning algorithms that lead up to the academic community's current standard. Much of the current research focuses on optimizing and improving existing methods rather than introducing entirely new methods. Both classes of algorithms are used in this thesis' system design.

2.1.2.1 Supervised Learning Algorithms

2.1.2.1.1 Linear Least Squares Fit

Linear classifiers create a model to fit a given vector of inputs $X = (X_1, X_2, \dots, X_p)$ to predict the output Y based on a linear combination of the input (Lawson, 1974). The model takes the form

$$\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$$

where $\hat{\beta}_0$ is known as the bias. Combining the bias (and $X = 1$) into vector of coefficients $\hat{\beta}$, we can rewrite the model as

$$\hat{Y} = X^T \hat{\beta}.$$

There are two broad classes of algorithms for determining the parameter $\hat{\beta}$ of a linear classifier: 1) discriminative models and 2) generative models. Discriminative models attempt to maximize the quality of the output of a training set, and generative models use probability density functions to estimate a given input's likelihood of belonging to a particular output class. Linear least squares fit (LLSF) is a discriminative model that picks $\hat{\beta}$, which characterizes the entire fitted surface, to minimize the sum of squares

$$SS(\beta) = \sum_{i=1}^{N} (y_i - x_i^T \beta)^2$$

This quadratic always has at least one minimum that can be demonstrated by writing the sum in matrix notation, then differentiating with respect to β:

$$SS(\beta) = (\boldsymbol{y} - \boldsymbol{X}\beta)^T(\boldsymbol{y} - \boldsymbol{X}\beta)$$

Setting the derivative to zero, we get:

$$\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\beta) = 0$$

If a unique solution exists, it is given by

$$\hat{\beta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

where $\boldsymbol{X}$ is an $N \times p$ input vector and $\boldsymbol{y}$ is N-vector of outputs from the training data. The prediction of an unknown input x_0 is $\hat{y}(x_0) = x_0^T\hat{\beta}$. Linear decision boundaries produced by LLSF assume that such boundaries are appropriate for the problem, making it a low variance and potentially high bias learning algorithm. Despite its relative simplicity, LLSF has been shown to perform comparably to some of today's best classifiers (Yang, 1999).

2.1.2.1.2 k-Nearest Neighbour

The idea of k Nearest Neighbour (k-NN) classification in based on simple intuition: new input that is 'close' to some training input is likely to have the same output as said training input (Yang, 1994). A single input may have several similar training inputs, so an average of these training outputs can be used:

$$\hat{Y}(x) = \frac{1}{k} \sum_{x_i \epsilon N_k(x)} y_i$$

Where $N_k(x)$ is the set of similar training documents, or neighbourhood. A common definition of this neighbourhood for inputs represented as vectors is cosine similarity:

$$cos(x_i, x_j) = \frac{\sum_p \omega_{pi} \cdot \omega_{pj}}{\sqrt{\sum_p \omega_{pi}^2}\sqrt{\sum_p \omega_{pj}^2}}$$

where x_i is a vector and ω_{pi} is the weight of the p^{th} element in input x_i. Unlike LLSF, k-NN does not build a fitted model at training time. Instead, every new input is compared to the training data at classification time. There are two main parameters for k-NN:

1) the size of the neighbourhood, k, and

2) the definition of distance to collect neighbouring items.

Optimizing the selection of k is meaningless at training time, since using $k = 1$ will always minimize the error for the training set. The error on the training set to roughly an increasing function of k. The decision boundaries produced by k-NN can be non-linear and are independent of assumptions about their appropriateness for the problem. Any subset of the decision boundary depends only on k inputs, making k-NN a high variance, low bias learning algorithm.

Current k-NN research focuses on runtime performance optimizations. Since this algorithm has no training, runtime depends entirely on speed of discovering and evaluating neighbours. There are two classes of optimizations:

1) better search strategies, and

2) reducing search space by combining neighbours.

There is no decrease in classification accuracy associated with 1), but start-up and runtime memory usage increases. Search space reduction methods trade classification accuracy for runtime performance. Runtime savings can be significant (up to a factor of 10) at an accuracy cost of about 1%.

2.1.2.1.3 Naïve Bayes

Naïve Bayes is a generative classifier that models the conditional probability density function $P(\boldsymbol{X}|class)$, and is based on Bayes' Theorem (Ng, 2002):

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

X is a collection of characteristics about the input, so the conditional probability can be rewritten as:

$$P(C|X_1, \ldots, X_n) = \frac{P(C)P(X_1, \ldots, X_n|C)}{P(X_1, \ldots, X_n)}$$

The denominator $P(X_1, \ldots, X_n)$ is effectively a constant because it does not depend on C and the X_i values are given. The numerator portion of the probability can be rewritten using the definition of conditional probability as follow:

$$P(C)P(X_1, \ldots, X_n|C)$$

$$= P(C)P(X_1|C)P(X_2, \ldots, X_n|C, X_1)$$

$$= P(C)P(X_1|C)P(X_2|C, X_1)P(X_3, \ldots, X_n|C, X_1, X_2)$$

which generalizes to:

$$= P(C)P(X_1|C)P(X_2|C, X_1) \ldots P(X_n|C, X_1, X_2, \ldots, X_{n-1})$$

At this point the ‘naive’ assumption is made. The distinguishing idea behind Naïve Bayes classification is that presence or absence of one input is independent of the presence or absence of any other input. Mathematically speaking, this means:

$$P(X_i|C,X_j) = P(X_i|C)$$

for all inputs where $i \neq j$. This simplifies our previous generalization to:

$$= P(C)P(X_1|C)P(X_2|C)\dots P(X_n|C)$$

The entire conditional distribution simplifies to:

$$P(C|X_1,\dots,X_n) = \frac{1}{Z}P(C)\prod_{i=1}^{n}P(X_i|C)$$

where Z is a constant. To classify an input X we can calculate the probability that it belongs to each possible class:

$$C(x_1,\dots,x_n) = \arg\max_{c} P(C=c)\prod_{i=1}^{n}P(X_i = x_i|C=c)$$

The desired output is a classification, not a probability, so this algorithm will be correct if the actual classification is the most likely, regardless of how accurate the class probabilities turn out to be. By assuming independence between all inputs, Naïve Bayes remains a one-dimensional problem. This simplifies computation, implementation, and analysis while working well for many real world problems.

Naïve Bayes classifiers are commonly used in commercial applications and in benchmarking state-of-the-art research, but are no longer a popular topic among the latest research. The methods introduced next are more widely researched.

2.1.2.1.4 Support Vector Machines

A support vector machine (SVM) creates a hyperplane in an n-dimensional space to separate a set of training points, as shown in Figure 3 (a) (Joachims, 1998; Brown, 2010). More precisely, SVM chooses the hyperplane whose sum of distances to the nearest data points on each side is maximized, as shown by the solid line in Figure 3 (b).

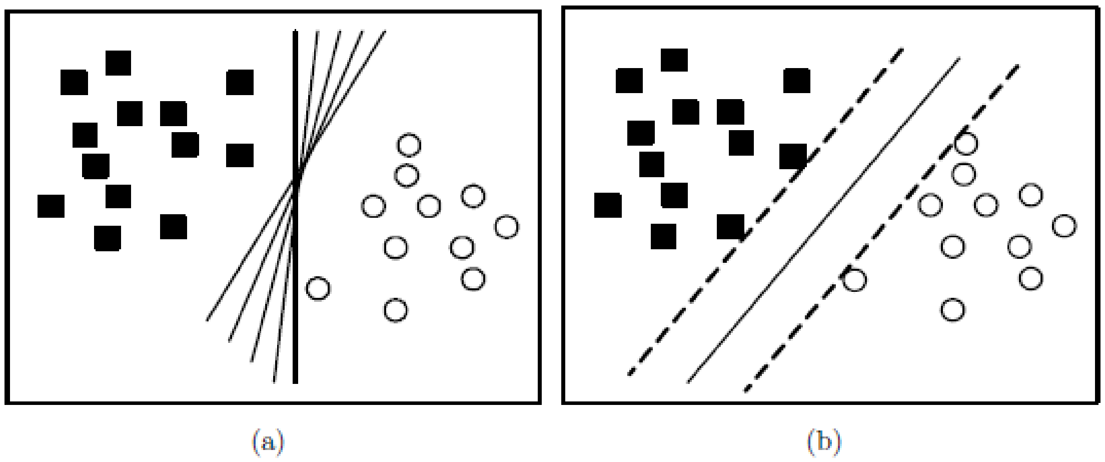


Figure 3: (a) Many possible hyperplanes can separate this data. (b) SVM chooses the maximum margin line of separation.

This section shows the formalization of SVM for the simplest case in which the data is linearly separable. SVM training data is of the form

$$\mathcal{D} = \left\{(\boldsymbol{x}_i, c_i) | \boldsymbol{x}_i \in \mathbb{R}^p, c_i \in \{-1,1\}\right\}_{i=1}^{n}$$

where $\boldsymbol{x}_i$ is a p-dimensional data point, and c_i is the classification for $\boldsymbol{x}_i$ which can be either -1 or 1. A hyperplane dividing the points $c_i = 1$ and $c_i = -1$ can be written as the set of points $\boldsymbol{x}$ that satisfy the equation $\boldsymbol{w} \cdot \boldsymbol{x} - b = 0$, where $\boldsymbol{w}$ is the normal vector perpendicular to the hyperplane and $\frac{b}{\|\boldsymbol{w}\|}$ is the hyperplane's offset along $\boldsymbol{w}$. SVM chooses $\boldsymbol{w}$ and b to maximize the distance between the hyperplane and each class' nearest points through which parallel hyperplanes run. These parallel hyperplanes are characterized by $\boldsymbol{w} \cdot \boldsymbol{x} - b = 1$ and $\boldsymbol{w} \cdot \boldsymbol{x} - b = -1$, and are represented by the dotted lines in Figure 3(b). The distance between these two hyperplanes is $\frac{2}{\|\boldsymbol{w}\|}$. The goal is to minimize $\|\boldsymbol{w}\|$ while keeping all data points outside of the margin, formalized as $c_i(\boldsymbol{w} \cdot \boldsymbol{x_i} - b) \geq -1$ for all data points. This optimization problem can be solved with a linear combination of training vectors:

$$\boldsymbol{w} = \sum_{i=1}^{n} \alpha_i c_i \boldsymbol{x_i}$$

where α_i is the set of Lagrange multipliers, most of which will be zero to satisfy the KKT conditions (Kuhn et al., 1951). Each non-zero α_i corresponds to a support vector $\boldsymbol{x_i}$ which lies on the margin.

This formalization makes several simplifying assumptions. The first assumption is that the training set is linearly separable. There are several methods for dealing with non-separable data, including:

1) the use of slack variables to maximize the boundary while allowing some data to be on the wrong side of the margin, and

2) transforming the data into a linearly separable set using kernel methods, which will be discussed in the next section.

A second assumption is two-class classification. A single hyperplane separates data into two classes. Multi-class SVM classification generally reduces to several binary classifiers.

Current SVM research focuses on reducing runtime computational requirements (memory and processing time), and on improving the efficiency of multi-class classification (Hsu et al., 2002; Shao et al., 2008; Dong et al., 2008). Computational gains are generally achieved by using approximations that do not affect the relative output of an SVM. The most common multi-class SVM methods are:

1) one vs. one, and

2) one vs. all.

In 1), every unique combination of classes is used to build an SVM. Similar to Naïve Bayesian methods, an input document is tested against every SVM and the classification is based on which class has the most votes. In 2), only one SVM is built for every class, in which the positive training documents are examples from the target class and the negative training documents are examples from all other classes. Similar to the previous method, a new document is tested against each SVM and the highest scoring class indicates the classification.

2.1.2.1.5 Kernel Methods

Linear SVMs are powerful but their applications are limited. Many real world problems are not linearly separable so SVM cannot be applied directly. However, transforming the input data to a higher dimensional space might make it linearly separable, as shown in Figure 3 (Rodriguez, 2004).

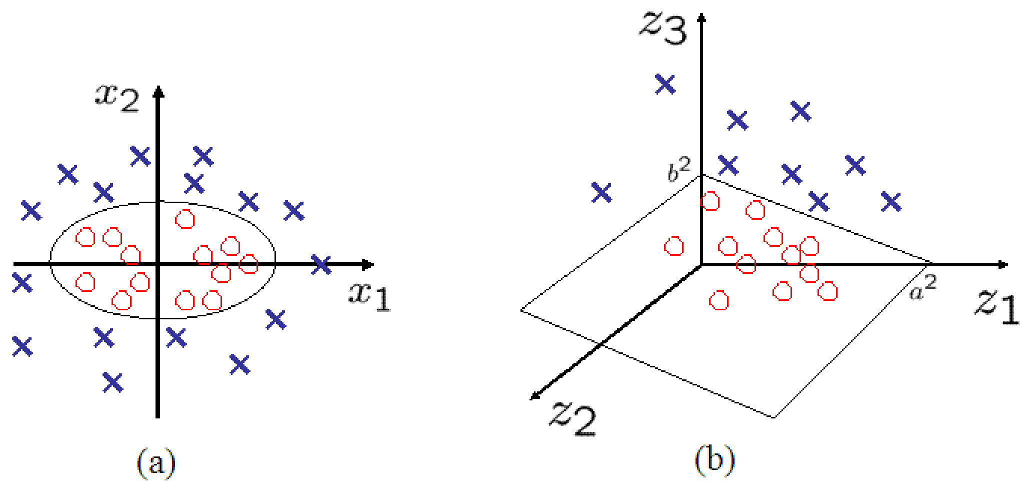


Figure 4: (a) Non-linearly separable data in two-dimensional space. (b) Transformation to three-dimensional space makes it linearly separable.

Using a linear classifier on the transformed input space will result in linear classification in the new space, but non-linear classification in the original space. SVM has a convenient application of this method: replacing the dot product with a kernel function K results in a boundary between classes: $K(\boldsymbol{w}, \boldsymbol{x}) - b = 0$. The resulting formalization is similar and enables non-linear classification.

2.1.2.2 Unsupervised Learning Algorithms

Clustering is a common form of unsupervised learning that groups items based on some measure of similarity. There are three classes of clustering algorithms: hierarchical, partitional, and spectral. This section focuses on clustering because it is the only type of unsupervised learning algorithm used in this thesis' design.

2.1.2.2.1 Hierarchical Clustering

Hierarchical clustering is an iterative approach that, given some set of clusters, refines them to a *better* set of clusters (Hastie et al., 2001). This is repeated until some convergence (or termination) criterion is met. Hierarchical clustering algorithms are iterative and either divisive or agglomerative. Divisive algorithms initially assume that all examples form a single cluster and clusters are divided at each iteration. Conversely, agglomerative algorithms initially assume that each example is in its own unique cluster and clusters are merged at each iteration. A similarity function measures the distance between any two given examples. Merging or dividing sets is determined by the pair-wise distance between examples. Common forms of the similarity function include Euclidean, Manhattan, and Hamming distance.

2.1.2.2.2 Partitional Clustering

Hierarchical algorithms require several iterations to reach a useful state. Partitional algorithms attempt to separate the examples all at once, although they can also iterate to improve results (MacQueen, 1967). Partitional clustering starts with k randomly generated clusters from the training set. Next, the centroid of each cluster is calculated and all examples are re-clustered by being place in the cluster whose centroid each is nearest to. Repeating the previous two steps can refine the choice of clusters, and as with hierarchical clustering, any similarity metric can be used. The initial randomization plays a significant role in determining the final clusters. Including a heuristic into the randomization scheme can help guide clustering in a desired direction.

2.1.2.2.3 Spectral Clustering

Accurate pair-wise similarity between examples is the key to good clustering. Spectral clustering represents the similarity between all examples in a matrix S, where S_{ij} is the similarity between examples i and j (Bach et al., 2003). Dimensionality reduction simplifies S, then another method (a partitional clustering algorithm for example) is applied to cluster the examples. Similar to previously discussed non-linear classification implementation, dimensionality reduction can introduce non-linear transformations to produce non-linear clustering.

2.2 Document Classification

A machine learning algorithm applied to the contents of a document is called document classification. Learning algorithms take a collection of vectors as input. In document classification, each vector represents a single document, and each element of the vector represents some word, or more generally, feature in the document (Joachims, 1998). Machine learning algorithms have been introduced in 2.1.2, so this section focuses on methods for converting input documents to vectors.

2.2.1 Pre-processing

A feature set is the collection of all possible features that a document can contain. In the simplest case, each document in the training set is converted to a set of features, and the superset of all document features makes up the feature set. A problem with this approach is the high dimensionality caused by considering every possible feature in each document (Yang, 2002). The goal of pre-processing is to reduce the size of the feature set without reducing the amount of information contained in it. Language lends itself well to this goal because the same information can be communicated in many different ways. Three common strategies are:

1) stemming,

2) synonyms, and

3) grouping.

Each strategy benefits from heuristics. A more domain-specific heuristic will lead to better results.

2.2.1.1 Stemming

Stemming is the process of reducing inflected words to their base form (van Rijsbergen et al., 1980). For example, the base form of the word *'swimming'* is *'swim'*. In document classification, the form of the word usually does not matter. A document that contains any of the words *'swim'*, *'swimming'*, *'swims'*, or *'swimmer'* might be about swimming. The form of the word it contains does not change this probability, so reducing all forms to the base form reduces the size of the feature set. The base form is chosen out of convenience rather than necessity. There are two classes of stemming algorithms: one that stems based on a lookup table, and one that stems based on patterns. As the name implies, lookup table methods must know the relationships between all root forms and inflected forms ahead of time. This has the benefit of always producing the correct result, but is a difficult requirement. Pattern-based methods are language specific, and attempt to generalize how inflected forms are allowed to relate to base forms. The key benefit is that no lookup table is required, but there is no guarantee that output will be valid. An inflected might produce the wrong base form (e.g. ‘*ran*’ reducing to itself instead of the correct ‘*run*’), or the base form might not be a valid word (e.g. ‘*friendliest*’ reducing to ‘*friendl*’). The first form of error limits the effectiveness of stemming for document classification, but the second form of error is a negligible concern. Document classification requires consistent features, not necessarily grammatically correct features.

2.2.1.2 Synonyms and Grouping

Synonyms are different words that have the same or similar meanings. Their application to document classification is similar to stemming: reducing the size of the feature set. Synonyms also help highlight important dimensions in the feature set. For example, if *n* training documents each contained a different version of the same synonym, a machine learning algorithm will see *n* different dimension, each with very little data. A synonym list reduces these to a single dimension, making it a stronger indicator for learning algorithms. Note that stemming provides a similar benefit.

Grouping takes similar features and reduces them to a common form. For example, the presence of a phone number might be an important indicator for a class of documents. However, the exact phone number is unlikely to matter. Grouping recognizes patterns, like phone numbers, and represents them with a common symbol. Another practical use of grouping is for prices. As with phone numbers, the precise value rarely matters, but unlike phone numbers, orders of magnitude may be good indicators. For example, $0.29 might be an insignificant indicator for documents about cars for sale, but $13,299 could be a very significant indicator.

2.2.2 Feature Extraction

Converting a document into a list of features (or at least potential features) is called feature extraction, and is done using tokenization (Weston et al., 2000). Tokenization is the process of breaking up a collection of words into tokens, where each token can be any subset of the original set, including a word, phrase, sentence, paragraph, or even the original set. The simplest tokenization algorithm uses whitespace to split a document into multiple tokens, where each word becomes a token. This works reasonably well for some languages (e.g. English) but fails for others (e.g. Japanese). Language-specific techniques such as part-of-speech tagging and parsing can function on any language, but are considerably more complicated. The order of tokens is often ignored; the set is treated as an unordered bag of words. A grouping pre-processor can handle order-dependant tokens by converting them into a single token. Another way to impose token order is with *n*-grams, where each token t_i is combined with up to *n* trailing tokens to form the following tokens: “t_i”, “$t_i\ t_{i+1}$”, ..., “$t_i\ t_{i+1} \dots\ t_{i+n}$”.

2.2.3 Feature Selection

Feature extraction converts a document to a collection of features. Feature selection chooses a subset of these features to form the feature set based on training documents. At runtime, a document is converted to a collection of features in exactly the same way, but only its features that are in the feature set are used for classification. Feature selection as a dimensionality reduction technique is optional. Using the superset of all training features is possible, but has some limitations including: 1) high dimensionality, 2) potential over-training to the training set, and 3) obscuring intuitive interpretation of a classifier’s performance. Feature selection algorithms assign a predictive weight to each feature, and then choose the top features based on some threshold.

Document frequency (DF) counts the number of training documents that contains each feature (Salton et al., 1988). The features with the highest counts form the feature set. Similarly, term frequency (TF) counts each feature’s occurrence in the training set (can occur multiple times per document), normalizes against the size of each document, and then applies the same threshold method for choosing the feature set. TF incorrectly emphasizes common words (e.g. *‘the’*). Term frequency-inverse document frequency (TF-IDF) attempts to overcome this emphasis by penalizing frequently occurring features’ weight. More formally, the TF of term t_i in document d_j is

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

where $n_{i,j}$ is the number of times t_i occurs in d_j and $\sum_k n_{k,j}$ is the number of features in d_j. The inverse document frequency is

$$idf_i = log\frac{|D|}{|\{d: t_i \in d\}|}$$

where $|D|$ is the size of the training set and $|\{d: t_i \in d\}|$ is the number of training documents that contain t_i. TF-IDF is

$$(tf - idf)_{i,j} = tf_{i,j} \times idf_i$$

Frequency alone is not always a good indicator the predictive power of a feature. Chi-squared (χ^2) identifies features that are highly dependent and assumes these features are the best classification indicators (Rogati et al., 2002). For a feature f_i in class c_j, χ^2 is:

$$\chi^2_{f_i,c_j} = \frac{\left(n(f_i,c_j) - E(f_i,c_j)\right)^2}{E(f_i,c_j)} + \frac{\left(n(\overline{f_\imath},\overline{c_j}) - E(\overline{f_\imath},\overline{c_j})\right)^2}{E(\overline{f_\imath},\overline{c_j})} + \frac{\left(n(\overline{f_\imath},c_j) - E(\overline{f_\imath},c_j)\right)^2}{E(\overline{f_\imath},c_j)} + \frac{\left(n(f_i,\overline{c_j}) - E(f_i,\overline{c_j})\right)^2}{E(f_i,\overline{c_j})}$$

where $n(f_i, c_j)$ is the number of training documents of category c_j in which contain f_i, $n(\overline{f_\imath},\overline{c_j})$ is the number of training documents not of category c_j which do not contain f_i, $n(\overline{f_\imath}, c_j)$ is the number of training documents of category c_j in which do not contain f_i, and $n(f_i,\overline{c_j})$ is the number of training documents not of category c_j which contain f_i. E is the expected value of similarly defined pair assuming that the feature and category are independent. For independent feature-category pairs χ^2 is zero. χ^2 is averaged over all categories for every feature to find each feature's marginal χ^2 value. The feature set is formed of the k features with largest marginal χ^2 value, where k is the desired size of the feature set.

Information theory measures the amount of new knowledge provided by some evidence as a probability that this evidence will alter the current result (Mitchell, 1997). In a binary classification case, evidence that is equally likely to predict either class contains no information. Information gain (IG) measures the number of bits of new knowledge toward predicting a class:

$$IG(f_i, c_j) = P(f_i, c_j) \cdot log\left(\frac{P(f_i, c_j)}{P(c_j) \cdot P(f_i)}\right) + P(\overline{f_\imath}, c_j) \cdot log\left(\frac{P(\overline{f_\imath}, c_j)}{P(c_j) \cdot P(\overline{f_\imath})}\right)$$

where $P(f_i, c_j) = {}^{n(f_i, c_j)}\!/_{|D|}$ is the probability that a document D contains feature f_i and belongs to class c_j. The $^{-}$ notation is consistent with the explanation of Chi-squared. The probabilities are computed based on occurrences in the training set and, as in Chi-squared, each feature's information gain is averaged over all classes to calculate a marginal value. The feature set is made up of the k most informative features.

2.3 Adaptive Document Classification

As document collections evolve, the context of the classification problem also evolves. One goal of automatic document classification is to classify a document as a human would. If a human's preferences change over time then their classification decision can also change. Terminology and content are perpetually changing aspects of language (Kroch, 1989), so using linguistic features for classification is time-sensitive. Classifiers need to evolve with the language. Domain changes can make the original document classes obsolete. As the logical lines between document classes begin to blur, new classes should be created. The document classification techniques described in 2.2 work well with consistent data, but do not adapt to changes. Using non-adaptive methods on a changing data set leads to accuracy degradation over time until new training sets are built (to reflect changes in the data), and new classifiers are trained against said sets. Many real-world document classification problems have evolving data. For example, an English language spam classifier would have to understand new slang and nomenclature that is added to the language every year. Rather than waiting for performance to drop below some threshold, there are classification methods that absorb new information as it becomes available. The methods presented in the remainder of this chapter will be used to evaluate the motivations for this thesis rather than for functional or performance comparison.

2.3.1 Adapting to User Interests

(Potamias, 2001) proposes an adaptive classification system for web documents that tailors results to user preferences. Web document classification is a specialization of document classification that takes advantage of structured data in web documents. Web documents contain tags that indicate the relative important of some words or phrases in a document, allowing feature selection to treat these features preferentially. Potamias's adaptive approach is described in the following steps.

1. A non-adaptive, supervised classification model is built using techniques described in 2.2. All output classes are known ahead of time.
2. Users create profiles that indicate *classes* and *keywords of interest*. Profiles do not contribute to classifier training therefore can be updated dynamically by users.
3. Incoming documents are classified against the generated models using non-adaptive techniques.
4. Classification output is compared against user preferences, and any document that is *un-interesting* based on the user profile is removed from the result set.
5. Users can also specify which web document tags they trust most, and if specified, the output set is refined further based on these tags.

Steps 4 and 5 implement this algorithm's adaptiveness, allowing each user to see results specific to their own preferences. As user preferences change, their profiles are updated and their result sets immediately reflect the change. However, changes to relevant features in the taxonomy are not detected or consumed.

2.3.2 Incremental Context Mining

Every class of documents has a key set of distinguishing features. Incremental context mining maintains a set of features per class, which the authors call a class' *contextual requirement* (Liu et al., 2002). This feature set is updated by every incoming training document, as shown in Algorithm 1.

Algorithm 1: Incremental Context Mining

Inputs	A text hierarchy $\boldsymbol{T}$. A training document $\boldsymbol{d}$ and the most specific category $\boldsymbol{c}$ that $\boldsymbol{d}$ belongs to.
Output	Update the feature set of $\boldsymbol{c}$ and its related categories in $\boldsymbol{T}$

Algorithm

```
1. $W \leftarrow \{w | w\ is\ a\ feature\ in\ d\ that\ occurs\ \geq \delta\ times\}$
2. While ($c$ is not the root of $\boldsymbol{T}$) do
   2.1. For each feature $w$ in $W$ do
        2.1.1. If $w$ is not in feature set of $c$, add to feature set
   2.2. For each of $c$'s features $w$ do
        2.2.1. Update strength of $w$
        2.2.2. For each sibling category $b$ of $c$ do
               2.2.2.1. If $w$ is in $b$'s feature set, update strength
   2.3. $c \leftarrow father\ of\ c$
```

Dynamic feature sets require a classification algorithm that does not regenerate its model as feature sets change. k-NN, which was introduced earlier, is an example of such an algorithm. The authors propose a similar algorithm whose distance measure is called *degree of acceptance*. Incremental context mining addresses the problem of vocabulary and context change with a class of documents assuming that it is presented with new training documents.

2.3.3 Dynamic Class Creation

The growth of the internet, including most subsets of documents within it, makes it impossible for a fixed class set to provide accurate classification of its documents. To maintain accuracy, an algorithm either needs to be re-trained, or it needs to be able to expand the class set. Dynamic class creation represents the category set as a hierarchy, and new categories can be added as leaf nodes in the tree (Choi et al., 2004).

Dynamic class creation consists of two stages: the first stage is to create a category tree that defines the current state of knowledge, and the second stage is to classify documents and update the category tree when needed. The choice of categories is dependent on the domain and is up to the user. As with other classification assumptions, the better the input, the better the generated class structure will be. Each category is used to generate a set of features that best predict its members. Next, the features are propagated from child nodes to parent nodes recursively, so that any node includes the union of all of its own features and all features of all its children. The importance of the features decreases as they move up the hierarchy to maintain each node's core relevance. Unique features among sibling nodes are considered key features, and are the basis of the search algorithm used for classification in the second stage.

Each node in the classification tree has a classifier that answers the question: "what is the probability that the input document belong in this class". Starting at the root node, an input document is classified against the root. If it passes, the same process it then classified against each of the root's children, and at most one child is chosen to continue the path. *Passing* means exceeding some probabilistic threshold based on statistics about each class' members. This process is repeated until classification does not pass, or it passes on a leaf node. Since each leaf node's feature were propagated upwards, a document belonging anywhere in the tree will have sufficient features along the path from the root to ensure correct classification, similar to how decision trees propagate conditions. There are two special cases where action is taken even when classification does not pass. When a document fails at any level in the tree, a new subclass is created under the class that the document had the highest probability of belonging to. This is called a deeper expansion. The second expansion occurs when a document is much more likely to belong to a parent node than to any of this parent's children. A new sibling node to the parent is created, called wider expansion. Once a document is assigned to a class, the features that most distinguish the document are incorporated into the class, and propagated upwards, allowing the class itself to further distinguish itself from other classes.

Creating dynamic classes ensures that classes remain distinct. The challenge becomes doing something useful with the generated classes. Similar to unsupervised learning, the output is statistically valid but highly subjective.

2.4 Summary

This chapter has introduced the current state of research in both static and adaptive document classification. The stated problem is to develop a classification system that works well across all web documents, where some subsets of documents are related by belonging to the same website. Data diversity and change over time makes this problem impossible to solve with a single document classifier, so an adaptive approach is required.

Adapting to user preferences is analogous to adapting to each website. The presented algorithm requires user preferences to be manually entered, so in the context of the stated problem, would require manual input for each website, which is an unreasonable requirement. Incremental context mining is a similar problem because just as two documents can represent the same information using different vernacular, so can two websites present the same documents using different templates. This algorithm requires labelled incremental training data, which would again require manual work per website in context of the stated problem. Creating dynamic classes theoretically allows each website to consist of a subset of classes. However, it is more likely that subsets across websites would collide, making it difficult to distinguish which classes contain the sought-after rich data (assuming that this data was correctly clustered, an unlikely assumption). The prior works do not address the problem of identifying rich data across web documents. Rich data is constantly changing, but summary data is concise and therefore much more consistent across all corpora. The solution proposed by this thesis is the first to take advantage of this consistency to solve the stated problem.

3: SYSTEM DESIGN

Vertical search engines provide a search service focused on a specific subset of internet content, such as research articles or classified ads. Their goal is to discover all relevant information available online, understand it, and make it searchable to their users. To achieve this goal there are three functional components that all search engines implement:

1) web crawling,

2) document classification, and

3) data extraction and indexing,

as shown in Figure 5 (Bialecki, 2009).

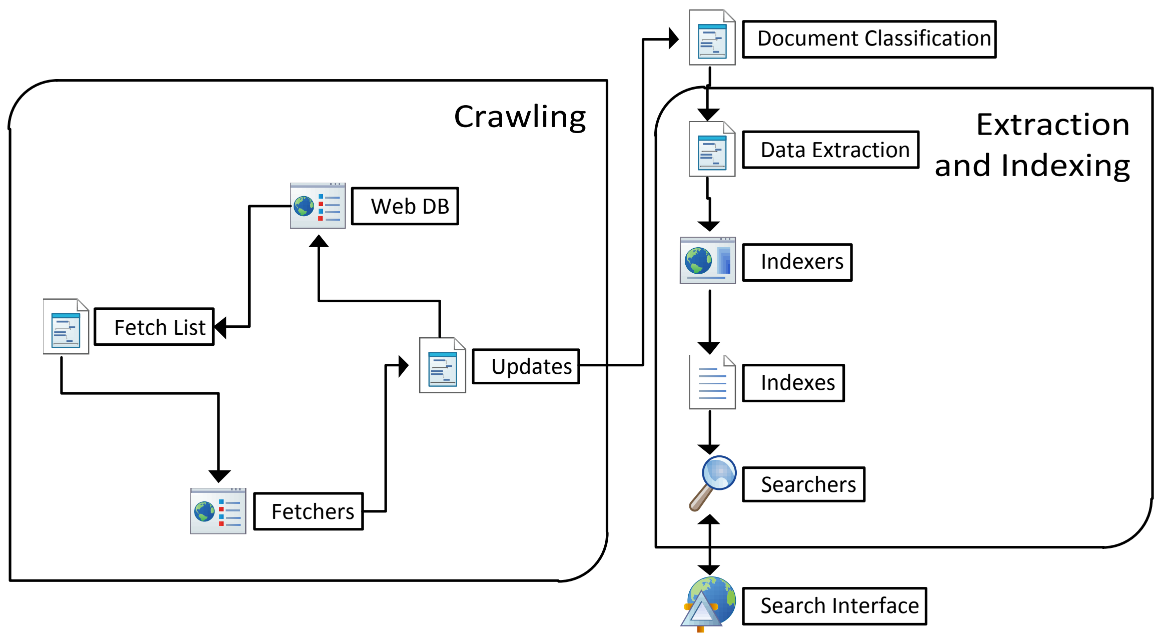


Figure 5: Vertical Search Engine System Design

Adaptive document classification uses the same general system design as shown in Figure 5 with several important distinctions to allow for feedback between the crawling and classification components to create site-specific classifiers. This chapter explains each of the core components involved, with special focus given to designs unique to this thesis.

3.1 Crawling

Crawling is the time consuming process of scouring the internet for content. While general-purpose (or whole-web) crawling might take several weeks or more on many computers, vertical crawling focuses on a subset of the internet so is much quicker (although it may still take days or weeks). A single cycle of crawling is shown below in Figure 6, which includes more details than the summary version presented earlier.

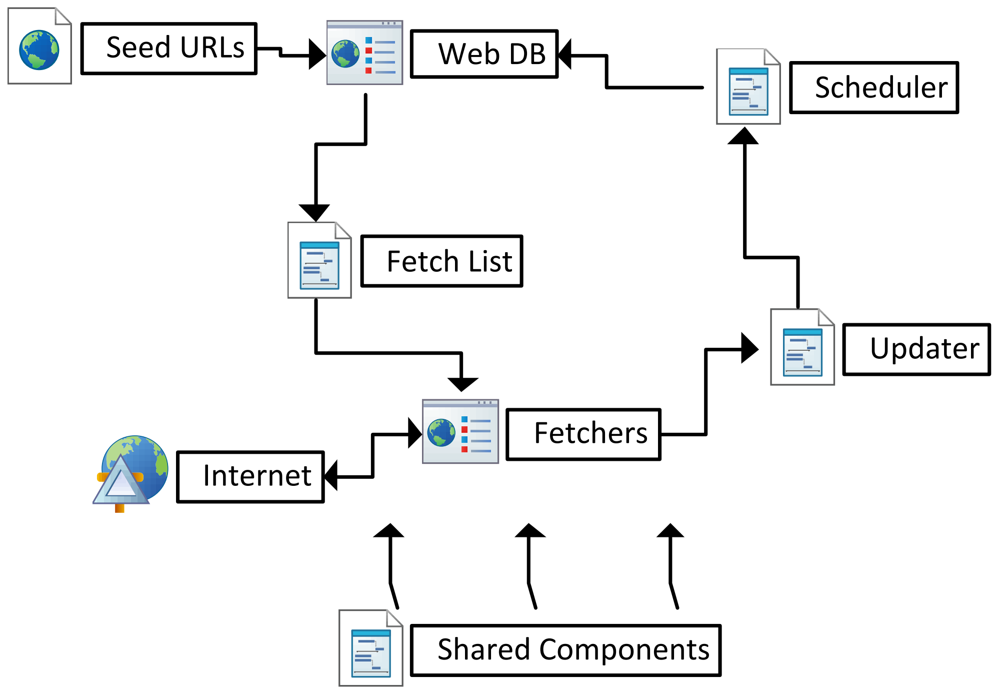


Figure 6: Crawling Design

A crawl starts with a seed list of Universal Resource Locators (URLs) that tell the crawler where to start. These URLs are stored in Web DB, which contains just URLs for pages to be visited, and complete information after a page has been visited including timestamps of when each document was downloaded and link structure of the document space. Each URL is then placed into a queue of items to be downloaded. A collection of Fetchers is responsible for monitoring this queue and fulfilling its requests. A Fetcher downloads the contents of a URL, and then passes it on to an Updater that is responsible for determining future URLs to visit and updating Web DB's content and context relating to the downloaded URL. This cycle continues until all desirable URLs have been visited.

Future URLs are based on outgoing links from a page, and for vertical search applications, are usually restricted to links within the current domain. Such crawlers are bounded by the set of domains provided in the seed set. The content of a page is used in a downstream collection of processes that will convert it into something searchable for a user. The context of a page is used to indicate relevance, or trust-worthiness, of its content. A page that has many incoming links means that many internet users find its content interesting and link to it from their blog, social networking medium or other form of website. In general, a web page with more incoming links has better information than one with fewer incoming links, so in cases where their content disagrees, the more popular page should be trusted. Google calls this trust system PageRank.

There are several important considerations to make crawling practical. Some websites want to be crawled so their content is more widely available, but others want to remain hidden (with more certainty than using voluntary crawl restrictions such as robots.txt). Such sites may set crawler traps that makes static content across an infinite number of pages appear unique. This will cause an infinite crawling loop. Some sites create crawler traps accidently by exposing the same content through several different URLs. The simplest way around such a trap is to restrict the depth that a crawler can visit, where depth is measured as the length of the link path away from a seed URL. The Updater is responsible for tracking depth and ignoring links that exceed that maximum allowed.

As soon as a URL is visited, its downloaded version becomes stale. Its author will not notify all crawlers when the URL content changes, so it is up to the crawler to determine how stale a URL is allowed to become. The rate of content change is highly variable from one website to another, but often consistent within a single website. Measuring this rate over several crawls of a single website gives a good indication of how frequently the website needs to be visited. The re-crawl frequency of a website can be controlled by a feedback system (e.g. PID controller) to maintain some acceptable level of staleness. As shown in Figure 6, the Scheduler is logically between the Updater and Web DB because in addition to scheduling re-crawl tasks, it is also responsible for scheduling real-time crawling tasks.

Crawlers access a website through the same interface as the websites users. Every website has some finite set of resources powering it, meaning that its concurrency rates are also finite. Aggressive crawlers can eat up a significant portion of these resources to the detriment of users visiting the same website. Polite crawling restricts where on a website a crawler is allowed to go, and the maximum bandwidth per unit time that a crawler can consume. The robots.txt standard allows webmasters to broadcast how they want their website crawled.

The most time consuming process in web crawling is downloading content from the internet. After the content is retrieved, the required series of tasks to process the content executes relatively quickly. Crawl parallelization allows multiple crawls and disjoint parts of the same crawl to run simultaneously across multiple machines. A key goal of crawl parallelization is to minimize the amount of overhead introduced from parallelization control while maximizing the download rate. Parallel crawling requires coordination to ensure the disparate crawlers do not download the same page, and that polite rules are obeyed across the superset of crawlers, not just within each individual crawler. A singleton model is used per domain to ensure politeness to that domain's resources by running all processing for a single domain on one logical machine. This simplifies de-duplication of newly discovered URLs by keeping all relevant data per domain in a single memory space.

A common set of utilities are used by several functional blocks, shown in Figure 6 as *Shared Components*. These components include URL content-type parsers, which translate specific content types like HTML and PDF into text, link parsers, which extract links from specific content types, URL filters, which have rules for acceptability of URLs, URL normalizers, which translates relative or partial URLs into a common form, and rendering engines, which render content and styles the same way a user would see them.

3.2 Adaptive Document Classification

This thesis' proposed adaptive document classification algorithm distinguishes it from other data gathering strategies for vertical search engines. Most websites that have many documents related to a single vertical have the same user flow for finding these documents:

1) a user enters some search criteria (either in a form or by selecting a pre-defined set),

2) the website presents the user many results showing summary information for each result, and

3) selecting a summary view takes the user to a detailed document about the entry.

Adaptive document classification creates a details page (DP) classifier per website. After the classifier is created, each page found by the crawler is classified using the classifier. Creating the classifier is described here, and shown in Figure 7. Each page found by the crawler is checked for usefulness as a training example. For each crawled page, the general purpose search result page (SRP) classifier is used to check if the page contains summary information about multiple detail pages. Note that this classifier is assumed to be an input to the algorithm. If the page is not an SRP then it is collected as a negative DP example. If the page is an SRP then it is mined for links to detail pages using a clustering algorithm on all the links on the page. If a cluster of links to detail pages cannot be found then the page is not used as a training example. Otherwise, each DP link is downloaded and its content is added to the collection of positive training examples, and the SRP is added to the list of negative examples. If a sufficient number of training examples are collected, then the site-specific DP classifier is trained. Otherwise, the crawler continues to its next location. After the DP classifier is trained, all pages visited within the website up to that point are classified. This thesis empirically found that five positive and five negative examples are sufficient to create a high accuracy classifier.

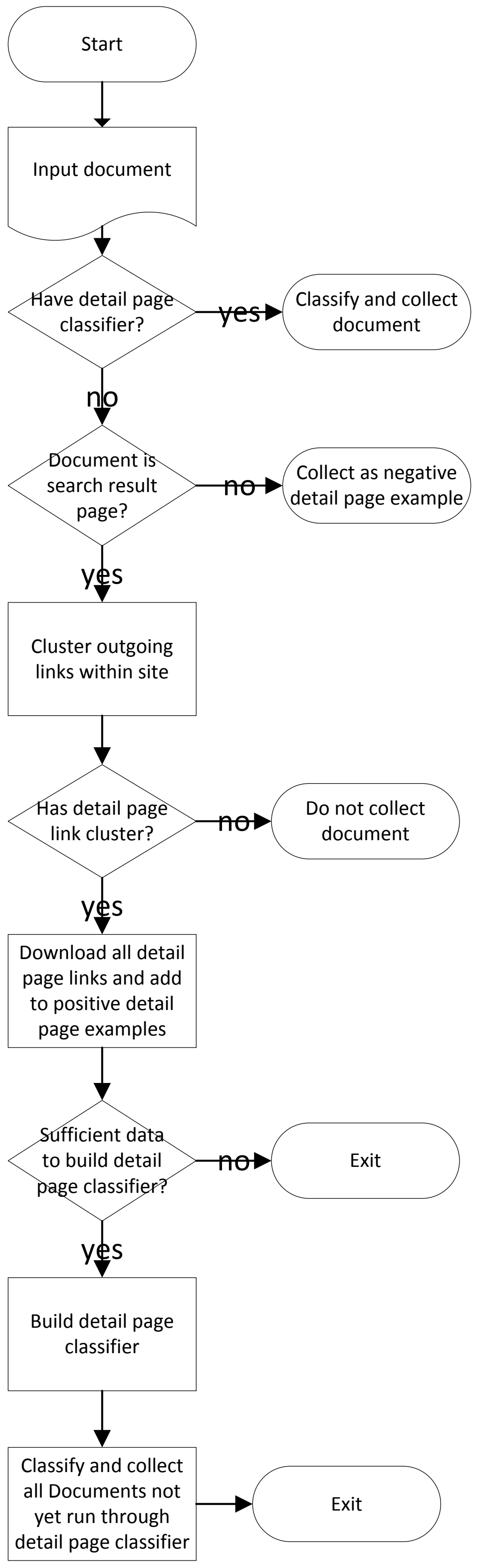

Start
Input document
Have detail page classifier?
yes
Classify and collect document
no
Document is search result page?
no
Collect as negative detail page example
yes
Cluster outgoing links within site
Has detail page link cluster?
no
Do not collect document
yes
Download all detail page links and add to positive detail page examples
Sufficient data to build detail page classifier?
no
Exit
yes
Build detail page classifier
Classify and collect all Documents not yet run through detail page classifier
Exit

Figure 7: Adaptive Document Classification Flowchart

As an example of how this works, assume a very simple website with a search box on its home page which takes users to a search result page, which contains links to detail pages. The crawler would first see the home page, and ADC would classify it as not being an SRP, so it would be collected as a negative DP training example. Next, the crawler would get to an SRP page, and ADC would recognize this fact (using the input SRP classifier). Next, ADC would find a cluster of links pointing to the DPs, download each link, and then add each to the collection of positive training examples. The SRP would be collected as a negative training example. The crawler would continue until at least five positive and five negative training examples were collected, at which point ADC would train a site-specific DP classifier, classify all pages visited within the website up to that point, and then continue crawling and classifying each new page using the DP classifier. A more realistic example would include pages within the website that are not in the search path. ADC would treat these similarly to the home page in the example.

The URL structure of all DP is assumed to be consistent within a single website. For example, the real estate listings website *realtor.ca* uses URLs like http://www.realtor.ca/propertyDetails.aspx?propertyId=10067266 and http://www.realtor.ca/propertyDetails.aspx?propertyId=9915829 to distinguish between two specific real estate listing detail pages. The only difference between these URLs is an identifier specific to each real estate property. When a document is classified as an SRP, a clustering algorithm is used to group all links from the document by similarity to each other. If the website fulfils the similarity assumption, then one of the resulting clusters will be a set of URLs that point to detail pages. The size of each cluster, statistics about the URL similarities, and domain-specific heuristics allow ADC to select the cluster containing the DP URLs. Each link from this cluster is downloaded, and its content is blindly added to the list of positive training examples for the DP classifier. Once a minimum threshold of training document size is met, a DP classifier is created for the website. All past documents are run through this classifier in a batch to find any DPs that have already been missed, and future documents are all classified in real time. The flowchart presented above includes several uses of the same logical components, but this may not be clear from the labels used. A similar flowchart is shown in Figure 8, but using common component names wherever applicable.

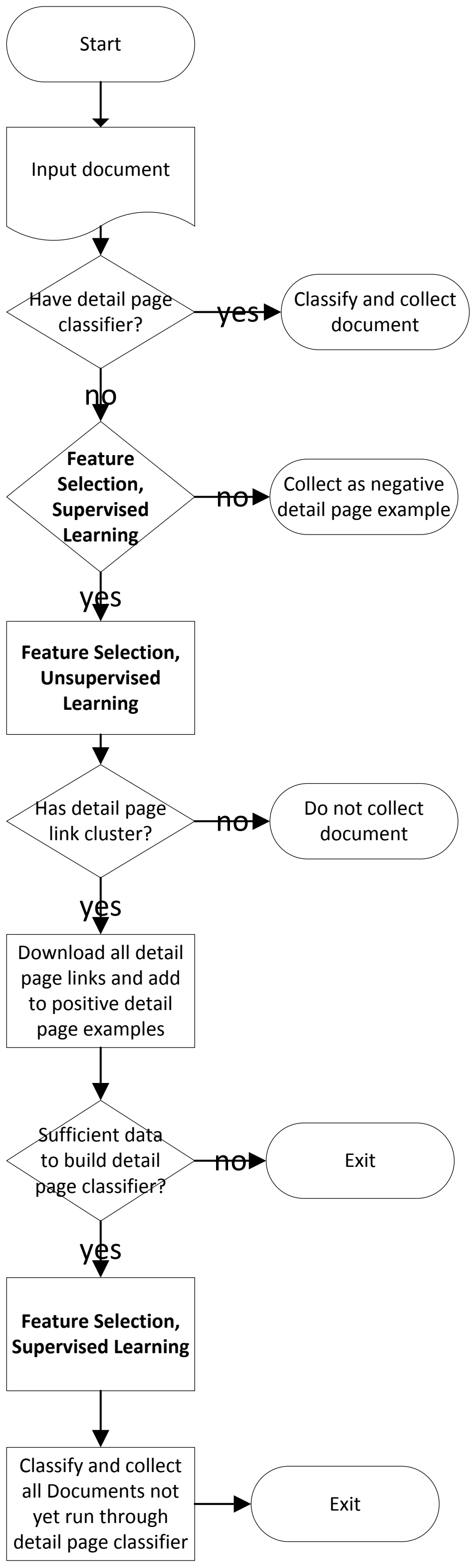

Start
Input document
Have detail page classifier?
yes
Classify and collect document
no
Feature Selection, Supervised Learning
no
Collect as negative detail page example
yes
Feature Selection, Unsupervised Learning
Has detail page link cluster?
no
Do not collect document
yes
Download all detail page links and add to positive detail page examples
Sufficient data to build detail page classifier?
no
Exit
yes
Feature Selection, Supervised Learning
Classify and collect all Documents not yet run through detail page classifier
Exit

Figure 8: Adaptive Document Classification with Functional Components

Feature selection, supervised learning, and unsupervised learning are the core reusable components of ADC. The remainder of this section will describe the design of these components, regardless of where in the system they are used.

3.2.1 Pre-Processing

Pre-processing is intended to reduce dimensionality of the feature-space by normalizing input. At both training and classification time pre-processing is applied before anything else is done to a document. At training time this means before generating the feature set, and at classification time this means before converting an input document to a feature vector. The same pre-processing steps must be applied to preserve the relevance of the machine learning model. There are three common cases that normalization is intended to handle: 1) inconsistent word or phrase usage, 2) *rogue* documents skewing feature counts, and 3) mark-up overwhelming the content.

3.2.2 Feature Selection

Feature selection is the process of converting a document into a vector in a feature space. In the case of supervised learning, a feature set is calculated at training time, and defines the feature space. In the case of unsupervised learning, the feature set is infinite. This section focuses on the supervised case, as it will become clear that the unsupervised case is a simplified form of the supervised case.

Feature selection has two distinct phases. The training-time phase determines the feature space and the classification-time phase determines the feature space vector for each input document. At training-time a feature extraction method (like those presented in 2.2.3) is applied to select the strongest indicators of each document class. The features are aggregated into a feature set, and then refined by retaining only the *top* positive and negative indicators. This sequence is shown in Figure 9.

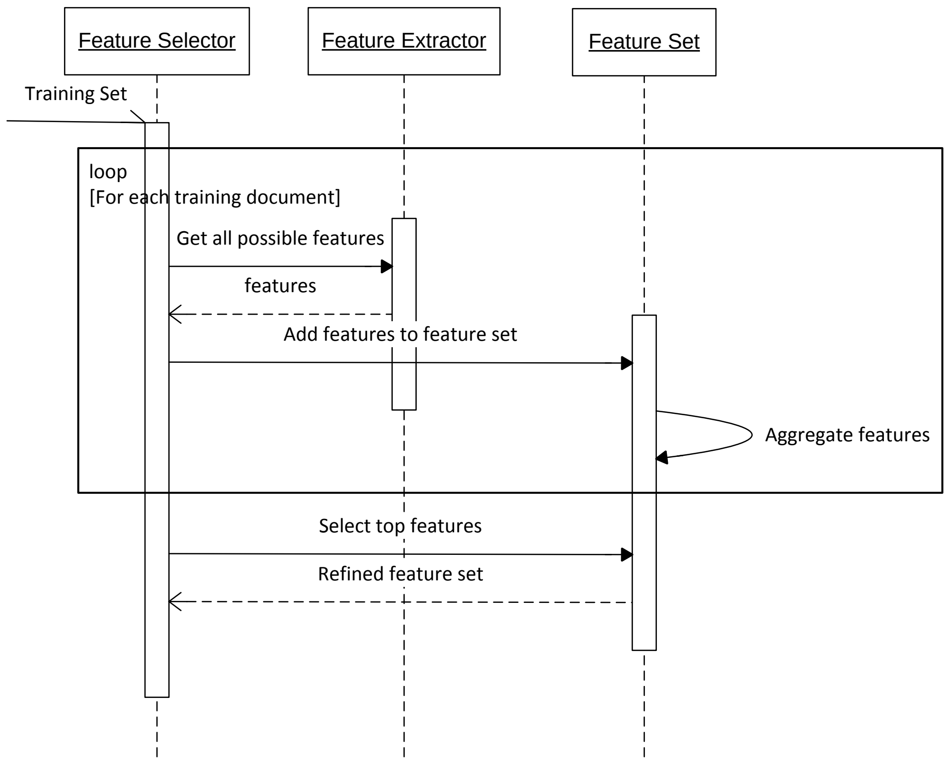


Figure 9: Training Time Feature Selection

Classification-time feature selection uses the feature set as a filter for input documents, as shown in Figure 10. In this context, *classification-time* means any time the system needs to convert a document into a feature vector. In the context of supervised learning, this needs to happen at both training-time and classification time.

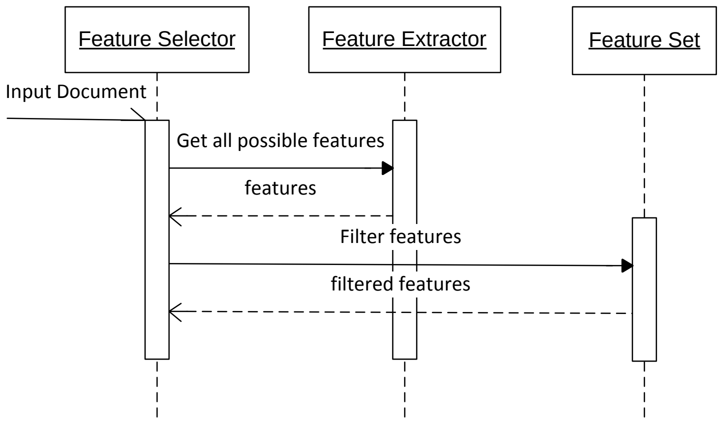


Figure 10: Classification Time Feature Selection

3.2.3 Supervised Learning

Supervised learning is used for both SRP and DP classification, with the SRP classifier provided as an input to the system, and the DP classifier trained dynamically per website. In both cases, the sequence of tasks performed is identical. The training-time tasks are shown in Figure 11.

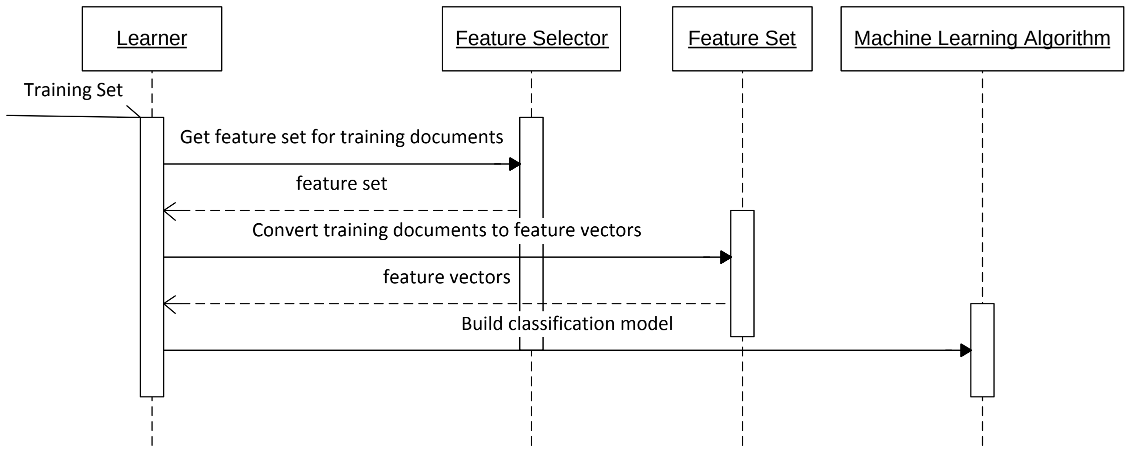

Learner
Feature Selector
Feature Set
Machine Learning Algorithm
Training Set
Get feature set for training documents
feature set
Convert training documents to feature vectors
feature vectors
Build classification model

Figure 11: Training-Time Supervised Learning

The learner receives a training set as input, consisting of positive and negative training examples. The training set is passed to the feature selector to generate an appropriate feature set. Next, each training example is converted to a feature-space vector using the feature set, and finally, the vector representation of all training data is passed to the supervised machine-learning algorithm to generate a classification model. Generating this model is the purpose of training. At this point, the classifier is ready to handle new (unclassified) data. The sequence of tasks at classification-time is shown in Figure 12.

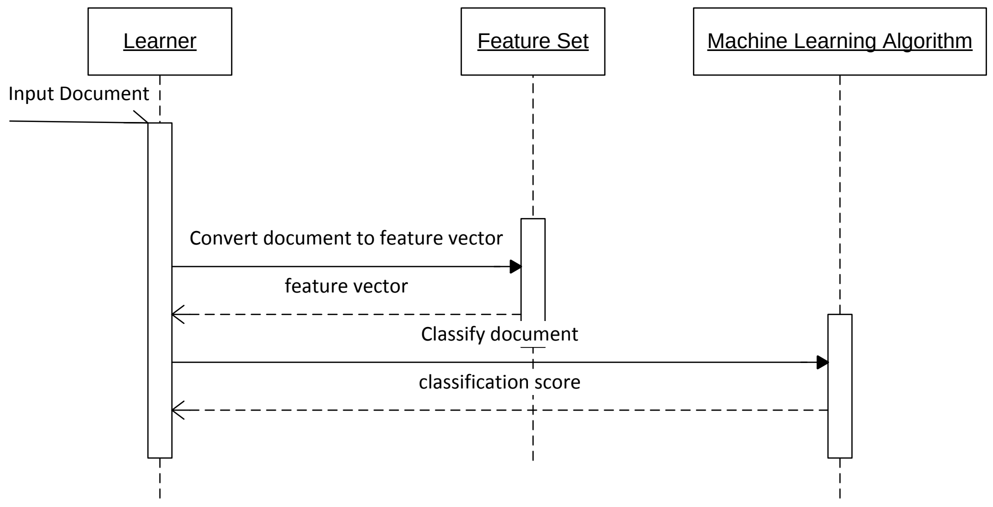


Figure 12: Classification-Time Supervised Learning

The same feature set must be used at both training and classification time to generate consistent feature-space vectors. At classification-time, a classifier logically deals with one document at time. The document is converted to a feature-space vector using the feature set, then the vector is compared against the classification model to generate a score. Depending on the algorithm used, the score can either be a discrete value indicating membership in some class, or a probability of inclusion in some class.

3.2.4 Unsupervised Learning

Unsupervised learning is used to cluster URLs on Search Result Pages to find a set of Detail Page documents. Detail Page URLs share patterns that allow them to be clustered with high accuracy. Unsupervised learning algorithms are not trained using labelled data so the only computation is at runtime, which is shown in Figure 13.

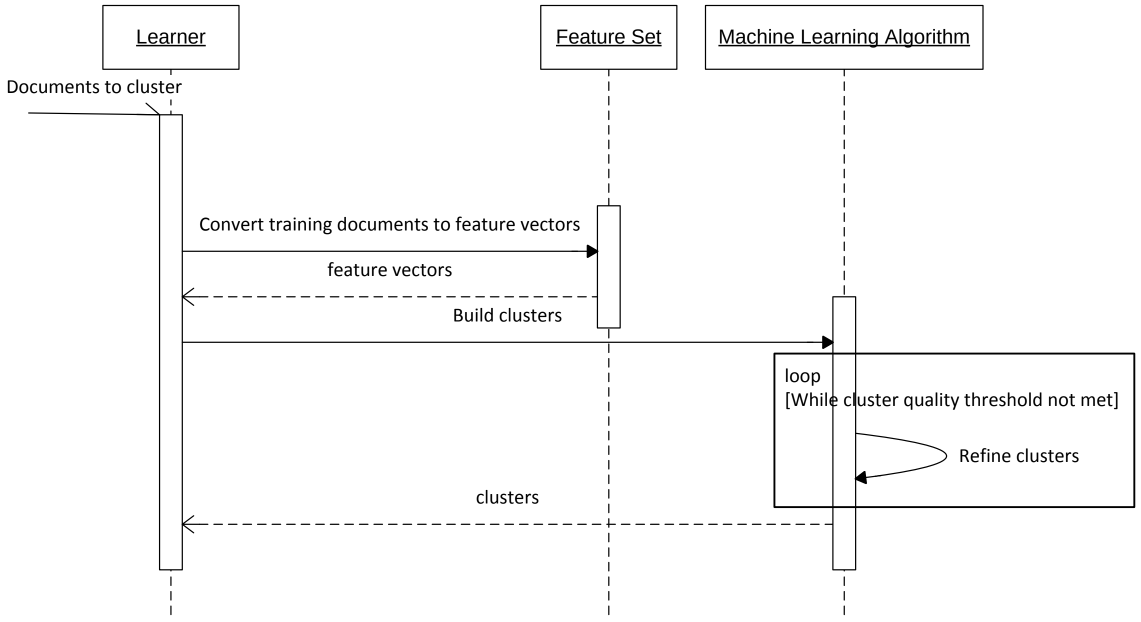

Learner
Feature Set
Machine Learning Algorithm
Documents to cluster
Convert training documents to feature vectors
feature vectors
Build clusters
loop
[While cluster quality threshold not met]
Refine clusters
clusters

Figure 13: Unsupervised Learning

Unsupervised learning deals with documents, and in this solution each document is a URL. A set of documents is passed to the learner to be clustered. A feature set is used to convert each document into a feature space vector. In this case the feature space is set of all possible URLs, so each URL is converted to a vector containing itself. In other solutions, the feature set can be optimized using the approach described in 3.2.2. The collection of URL vectors is passed to the unsupervised machine learning algorithm for clustering. The beginning state of unsupervised algorithms is implementation-dependant, but all algorithms share the flow of iteratively improving the existing cluster set until some exit criteria is achieved.

This thesis includes two uses of unsupervised learning. URL clustering is consistent with traditional unsupervised learning applications, and is the obvious first example. The second example is more subtle. The flowchart in Figure 7 shows that for every website a site-specific DP classifier is created. At a high level, the input to this algorithm is:

1) a target website (while simplifies to a collection of documents), and

2) an SRP classifier (a domain-specific heuristic).

Note that the input does not include DP training data. The output of this algorithm is two sets, or clusters, of documents:

1) a set of DP documents, and

2) a set of non-DP documents.

The implementation details include supervised learning to generate these output sets, but the supervised learner's training data is generated at runtime rather than being passed as input. This differs from traditional unsupervised learning algorithms because the classes of the output clusters are known ahead of time (DPs and non-DPs). Traditional algorithms cluster the input documents based on how the data itself is most obviously separable, and the classes of the output clusters are unknown.

3.3 Data Extraction and Search

The content of online documents can be categorized in two ways: unstructured and structured. Unstructured data is a collection of text with no logical distinction between any subsets of the text. The core content of an article about importing cars from the United States to Canada is an example of unstructured data. Structured data is a collection of name – value pairs, where both names and values are arbitrary strings. The career statistics of a professional athlete are an example of structured data. There is a middle ground in data categorization called semi-structured data, which contains the same information as structured data, but the presentation of the data does not conform to any formal standard. An HTML page showing the career statistics of a professional athlete in a table is an example of semi-structured data. For vertical search, structured data is needed but generally not available to crawlers. However, DPs contain semi-structured data that can be converted to structured data. This is called data extraction.

The end goal of data extraction is to obtain structured data to present is some domain-specific product, so data extraction is inherently a domain-specific task and implementations rely heavily on rules or heuristics. One of the key challenges in data extraction is ambiguity. A word or phrase can have different meaning and relevance depending on its context. Continuing a previous example, a document with an athlete's statistics may also include a date. This date might be the current date, the date on which the statistics were compiled, or a date indicating statistics of a subset of the athlete's career. In the first two cases the date would be irrelevant, and in the third case the date would be relevant, and its meaning would be necessary to understand. Accurate document classification helps alleviate (but not eliminate) the ambiguity problem. The richest semi-structured data on a website will reside on DPs. Other pages within the site will also contain semi-structured data, sometimes relevant to the desired subject, but also irrelevant data that looks indistinguishable from the relevant data. Document classification eliminates the irrelevant non-DP documents from ever getting to data extraction. Ambiguity within a single DP is handled within the extraction implementation. The sequence diagram of such an implementation is shown below in Figure 14.

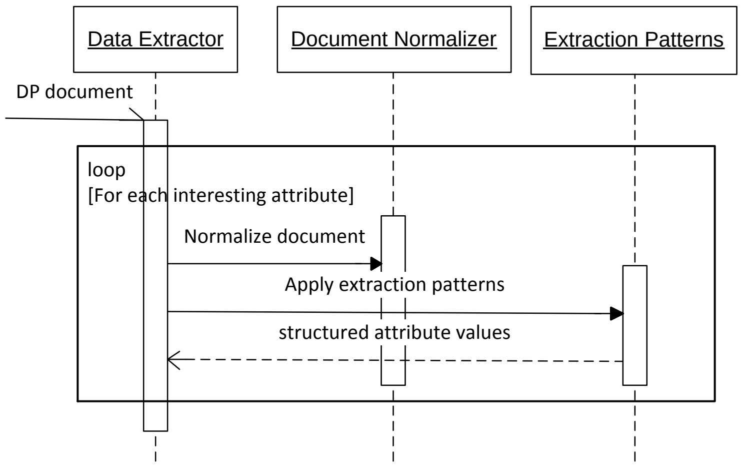


Figure 14: Data Extraction

Each *interesting* attribute has two sets of inputs: 1) normalization rules, and 2) extraction patterns. Normalization rules eliminate the variation of attribute names and values between sources. Using the athlete example again, an athlete’s point total may be indicated by any of the following strings: “pts”, “points”, “pt”, etc. Extraction patterns identify relationships between normalized name-value pairs in text, and extract the values. Extraction patterns can sometimes be generalized, and sometimes need to be website specific.

Structured data needs to be exposed after it has been collected. A search interface (similar to the one whose structured is taken advantage of to build the adaptive document classifier) is built to allow users to browse whatever subset of the data interests them. A search service is the empowering component of such an interface, the design of which is shown in Figure 15.

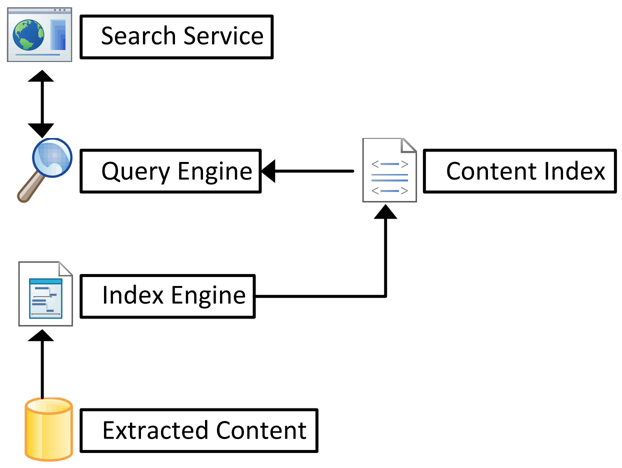


Figure 15: Search Design

The extracted data is fed to an index engine to build an index. Search indexes store data in a way that makes search fast. Traditional data stores persist entries as key-value pairs, where the key is the document’s identifier (for example, title) and the value is the content of the document. An index engine inverts this relationship and stores the content as the key because user queries are run against content, not identifiers. A single document will have many entries in an index, one for each *token* in the document, where a token is some subset of the document. This structure allows query engines to accept query trees of Boolean logic and perform set operations on document identifiers to fulfill each query.

4: IMPLEMENTATION

The sample implementation focuses on the novel algorithms proposed in this thesis, including relevant comparisons to existing methods. Adaptive document classification proposes a method for creating high-accuracy, website-specific detail page classifiers to capture rich and relevant data. Of the three main components introduced earlier, only crawling and classification are implemented. Crawling is required to find and download web pages, and classification includes the novel solution. Data extraction and search are excluded from this implementation. There are well known patterns for their implementations, and their exclusion here does not change this thesis' ability to evaluate the proposed methods. A revised system design is shown in Figure 16 that strikes out the excluded components. Real estate listings are chosen as the target data, and a Search Result Page classifier for real estate listings is built and passed to the system as input.

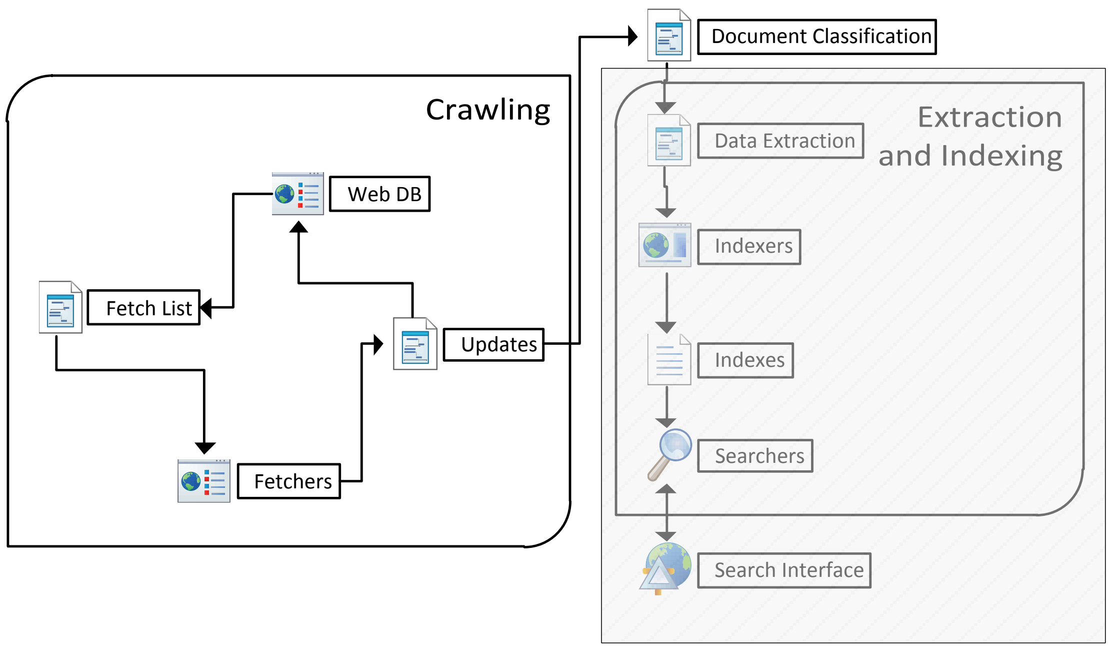


Figure 16: Implemented System Design. Shaded area is not implemented.

4.1 Crawling

This implementation's crawler uses the open source framework called the *Smart and Simple Web Crawler* (SSWC) (Torunski, 2009). This crawler is written in Java and implements most of the design shown in Figure 16. This section presents key features of this crawler, its limitations, and implementation details of this project's customizations.

SSWC controls the lifespan of a crawl in two ways: 1) maximum depth of a crawl and 2) maximum crawl iterations. Crawling generates a link (or page) graph by following links from one page to the next. Maximum depth restrictions prevent the crawler from traversing more than *n* links away from a seed URL, where *n* is the specified limit. The link graph's depth is restricted to this same maximum, but the total size of the link graph is unbounded. The memory footprint of maximum depth restrictions is considerable since the full link graph must be maintained. Maximum crawl iteration restrictions limit the total size of the link graph, but do not constrain its depth. The memory footprint of crawl iteration restrictions is minimal since a link graph is not necessary.

Some links are undesirable to crawl for a variety of reasons including unknown content types, irrelevant data, and recognized crawler traps. SSWC provides a filter interface that allows the implementer to specify acceptable URL patterns. Each filter is constructed with a user-specified list of acceptable links, but this list has a different meaning in the context of each filter. The *Beginning Path Filter* accepts links that begin with the same substring as any one of acceptable links. This restricts a crawler to a sub-tree of the link structure within a website. The *File Extension Filter* accepts links that have the same extensions as any of the acceptable links. This eliminates unknown content types and can protect against some crawler traps. The *Regular Expression Filter* treats each acceptable link as a pattern, and incoming links must match one of these patterns to be accepted. These filters implement heuristics when targeting a subset of URLs on a specific website or when avoiding specific crawler traps, and do not generalize well. The *Server Filter* limits a crawl to the collection of domains present in the acceptable links list. These filters can be combined using arbitrary Boolean logic. ADC uses the server and regular expression filters to restrict crawls to individual domains, and to exclude known *bad* content types like Portable Document Format (PDF) and Flash (SWF), where *bad* content simply means that this crawler does not know how to treat the content as plain text.

SSWC supports single and multi-threaded modes. Network bandwidth is a single-threaded crawler's bottle neck, so multi-threaded crawling allows concurrent downloading of several web pages. Starting from an input seed list (of an arbitrary number of URLs), multi-threaded begins by downloading each seed, then waiting for events to continue the crawl. SSWC exposes two events: 1) download complete, and 2) link graph parser events. The first event is well described by its name. The link graph parsing event is triggered after basic parsing and processing has been applied to a downloaded web page, including updating the link graph to include new page and parsing its links to trigger subsequent downloads. SSWC uses the same event-based model for single-threaded, but limits the total number of threads to one.

The event-driven design allows for simple integration of ADC-specific components into the crawling flow. ADC uses the link graph parsing event to include itself in the data flow. Every downloaded document triggers the sequence of events shown in Figure 7. If a Site-Specific Classifier (SSC) already exists then new documents are treated strictly as candidates to be Detail Pages, and are classified using the SSC. If a SSC does not yet exist then new documents go through several processes. First the document is classified using the Search Result Page Classifier to check if it contains training data for the SSC. Next, the document is marked as needing to be classified by the SSC when it is eventually built. Finally, if sufficient training data exists after including new training data from this document then the SSC is built, and all documents marked as needing to be classified are classified in a one-time batch process. Once the SSC is built, new documents are classified immediately.

SSWC does not implement the full crawling design described earlier. Specifically, it excludes scheduling and only optionally tracks link graphs. The exclusion of these components does not affect this thesis' ability to demonstrate ADC and measure its effectiveness.

4.2 Document Classification

Document classification involves three distinct functional components: 1) document pre-processing, 2) feature selection, and 3) machine learning. Pre-processing applies normalization rules to generate consistent documents from inconsistent sources. Feature selection computes a feature set, and then converts text to feature-space vectors.

4.2.1 Pre-Processing

Document classification benefits from pre-processing by producing a smaller and more relevant feature set. This section lists the specific pre-processors used.

4.2.1.1 HTML Decoding and Filtering

Web pages contain mark-up (i.e. HTML), advertising, navigation, and other irrelevant content that does not contribute to the core content of the document. The visible content of a web page is typically a small subset of its underlying HTML code. The additional HTML code includes structural and meta-data for the page. For example, displaying content in tabular form requires the use of the *table*, *tr*, and *td* HTML tags. Structural HTML code is quite consistent across all types of web pages, so in general it is not useful for web page classification. Meta-data is supposed to represent a page's content, but because the author has complete freedom to add arbitrary meta-data, the containers are frequently abused for the purpose of enhancing a page's rank in a search index. It has been empirically shown that classifying a page using its meta-data actually reduces accuracy (Chekuri et al., 1997).

HTML decoding and filtering has two objectives: first to translate any encoded data into its symbolic representation, and second to keep just the relevant portion of the marked-up web page. Most mark-up (including meta-data and other invisible elements) should be removed from the document, but some may be desirable when it corresponds to visual elements that are emphasized to the user, such as a document's title. This implementation uses the Jericho HTML Parser (Jericho, 2009) to convert web pages into object model instances, and allows ADC's start-up parameters to specify which HTML elements are desirable.

4.2.1.2 Part of Speech Selection

The English language has many words that are ambiguous without context. Words can have different meanings depending on whether they are acting as nouns or verbs, preceded by adjectives or adverbs, or separated by an indicator like a quote or comma. This implementation uses a Part of Speech tagger called OpenNLP (Baldridge et al., 2003) to tag every word in each input document, allowing ADC's start-up parameters to choose which tags are important. The implementation demonstrates several different choices and compares their performance.

4.2.1.3 Stemming

As described in 2.2.1.1, stemming reduces all forms of a single word to its base form. This implementation uses the Porter stemming algorithm (van Rijsbergen, 1980), a pattern based algorithm originally introduced more than 20 years ago. Porter stemming does not produce the *paradigm* form of every word, but does produce consistent stems.

4.2.1.4 Synonym Lists

Section 2.2.1.2 describes two classes of synonyms:

1) several different words (or abbreviations) with the same meaning, and

2) a specific value where only the *type* of the value matters.

This implementation includes many synonyms of each type. For example, the United States Postal Service's official list of abbreviations is used to equate state/provinces with their abbreviations, street suffixes with their common and abbreviated forms, and unit designators with their abbreviated forms (USPS, 1998). Table 1 below presents a sample of the grouping synonyms that help identify real estate listings.

Table 1: Grouping Examples

Group Name	***Synonyms***
MonthOfYear	January, February, March, …, December
Remodel	Renovate, new paint, fresh paint
Neighbourhood	Neighborhood, community, location
Price10^6	$[1-9][0-9]{5}(\.[0-9]{2})?
NorAmPhoneNum	[0-9]{3}-[0-9]{3}-[0-9]{4}
City	Vancouver,Surrey,Richmond,…

4.2.1.5 Other Text Normalization

Several other normalization techniques are used here that do not fit into any of the preceding headings, and do not warrant in-depth discussion. A case filter is used to consistently case all incoming text, with lower case arbitrarily chosen here. A length filter is used to limit the maximum number of characters in an incoming document. Web pages must are meant to display information to a human, and it is unlikely for a real estate listing to exceed several pages of text. Other legitimate content (e.g. research articles) may exceed this length, but hurts the computational performance of feature selection and machine learning. Similar to how structural information does not contribute to HTML pages, the English language contains many words that do not contribute significantly to the meaning of a sentence (at least from a bag-of-words document classification perspective). A stop word filter eliminates these words from all input documents.

4.2.2 Feature Extraction and Selection

Feature extraction and selection work together to solve two problems: 1) converting all training documents to collections of features and selecting the top *n* as the feature set, and 2) converting classification-time documents to feature vectors by filtering their features through the feature set. Both operations begin by converting input documents into a collection of features, as shown in Figure 17.

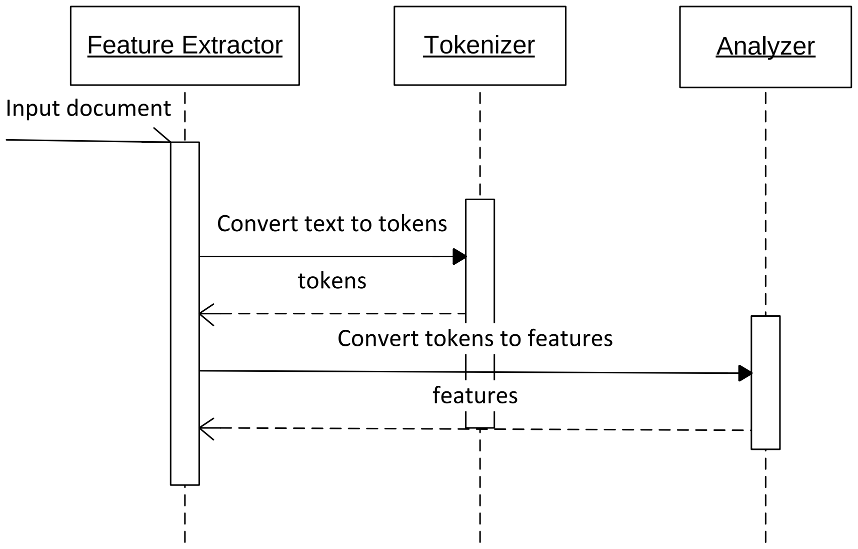


Figure 17: Feature Extraction

The first step is to tokenize the input document's text into an ordered list of tokens. This implementation includes two different tokenizers: a whitespace tokenizer, which splits text by whitespace, and a non-alpha-numeric-character tokenizer, which splits text by any non-letter, non-number character. A more intelligent tokenizer might have knowledge of English grammar. Tokens form the basis for features, but are not necessarily features themselves. To include contextual meaning, it can help to combine contiguous (or nearby) tokens into a single feature. For example, "New York" as a single feature would likely contribute very differently to a document classifier than two individual features "New" and "York". Analyzers convert the token stream into an unordered collection of features, where each feature contains one or more tokens and a count for the number of times it occurs in the document. This implementation includes several different analyzers.

4.2.2.1 Word Selection

Word selection is the simplest form of analysis. A document is tokenized into a collection of tokens, and each token is taken to be a feature.

4.2.2.2 Bi-grams

Real estate listings can contain important features that have no neighbouring tokens. For example, a price may appear between two pictures. In such cases it is useful to allow for single and multi-token features. Bi-gram analysis selects all individual tokens to be features, as well as all contiguous pairs of tokens (Tan et al., 2002).

4.2.2.3 Sparse Binary Polynomial Hash (SBPH)

The shortcoming of word selection is the inability to capture phrases. SBPH examines the document with a sliding window of m tokens at a time and selects all phrases from within this window (Yerazunis, 2003). For example, the first window with $m = 5$ in this definition would be “the shortcoming of word selection”. The second window would be “shortcoming of word selection is”. And so on until the end of the document. Within each window several features are selected (specifically $2m - 1$). The first token of the window is always part of the feature, along with all possible combinations of the remaining $m - 1$ token. Table 2 shows all features for the first window in this paragraph.

Table 2: SBPH Sample Analysis

the ~~shortcoming of word selection~~	the shortcoming ~~of word selection~~
the ~~shortcoming of word~~ selection	the shortcoming ~~of word~~ selection
the ~~shortcoming of~~ word ~~selection~~	the shortcoming ~~of~~ word ~~selection~~
the ~~shortcoming of~~ word selection	the shortcoming ~~of~~ word selection
the ~~shortcoming~~ of ~~word selection~~	the shortcoming of ~~word selection~~
the ~~shortcoming~~ of ~~word~~ selection	the shortcoming of ~~word~~ selection
the ~~shortcoming~~ of word ~~selection~~	the shortcoming of word ~~selection~~
the ~~shortcoming~~ of word selection	the shortcoming of word selection

4.2.2.4 Orthogonal Sparse Bi-gram (OSB)

The problem with SBPH is that it generates a very large number of features. OSB attempts to maintain the benefit of combining tokens without the overhead of such a large feature count (Yerazunis, 2003). The same windowing technique is used as in SBPH, but this time only combinations of two tokens are considered to be features. Table 3 shows features from the preceding SBPH example using OSB analysis instead.

Table 3: OSB Sample Analysis

the shortcoming
the of
the word
the selection

4.2.2.5 Choosing Feature Set

The first step in choosing a feature set is converting the training documents into features using one of the implemented analyzers. As documents are converted, the output features are also *scored* so that they can be ranked. This implementation includes three scoring algorithms: 1) term frequency, 2) document frequency, and 3) term frequency-inverse document frequency, each of which is described in 2.2.3. The features from the positive and negative training sets are scored, collected, and ranked independently so that a desirable ratio, *r*, of positive to negative features can be selected. A total of *n* features are selected from both sets using the ratio *r* to form the feature set. This implementation uses *r* = 1.

4.2.3 Supervised Learning

Vector-based supervised learning has two applications within the proposed solution: 1) the input SRP classifier and 2) the site-specific DP classifier generated once enough training data is collected. This implementation includes four different supervised learning algorithms, varying in complexity from the “simplest” classifier to the state of the art. The simple implementation is included as both a sanity check on more complex implementations and to test whether good training data can overcome simple classification assumptions.

4.2.3.1 Simple (Density) Classifier

The density classifier accepts a feature set of weighted words or phrases as input. Weights can be positive or negative, and occurrences of features with positive weights suggest a document’s inclusion in the desired class (and vice versa for negative weights). Longer documents tend to contain more occurrences of all features, including those in the feature set. Document length normalization adjusts a document’s score to reflect the density of features it contains. The output score is compared against a threshold to produce binary classification. Density classification is shown below in Figure 18.

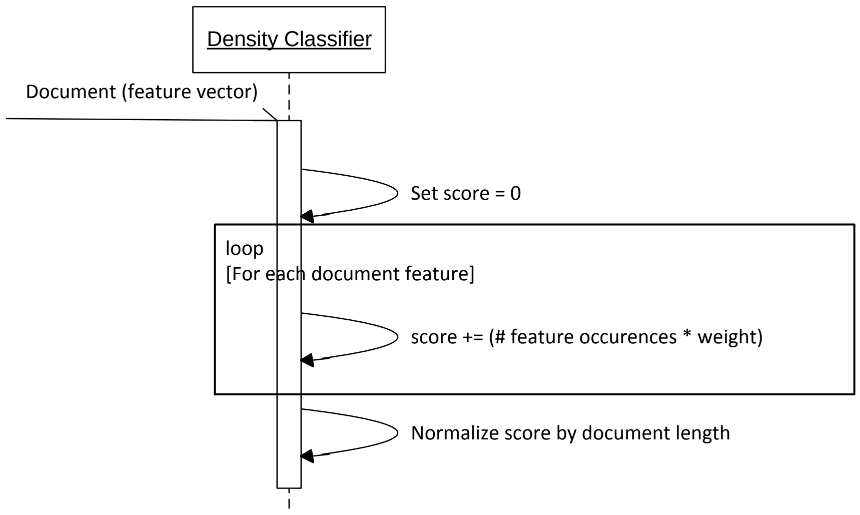

Density Classifier
Document (feature vector)
Set score = 0
loop
[For each document feature]
score += (# feature occurences * weight)
Normalize score by document length

Figure 18: Density Classifier Algorithm

4.2.3.2 k-Nearest Neighbour Classifier

The challenge implementing k-NN is with finding neighbours. Ternary search trees (TST) offer a fast and efficient way to find neighbouring strings using hamming distance to define neighbourhoods (Bentley et al., 1997). Each node in a TST is a modified binary search tree where the left child represents a value greater than the node, the right child represents a value less than the node, and a third child is used if the lookup character is found on the node. This allows string lookup to consist of a series of binary searches for individual characters. More importantly to implementing k-NN, it also allows for fast neighbouring string matches. This algorithm is shown in C++ style pseudo code in Algorithm 2.

Algorithm 2: Ternary Near Search

Inputs	Ternary search tree, $\boldsymbol{p}$ Search string, $\boldsymbol{s}$ Maximum distance from $\boldsymbol{s}$, $\boldsymbol{d}$
Output	List contains near matches, *srcharr*

Algorithm	
	```
	void NearSearch(Tptr p, char *s, int d)
	{
	    if (!p || d < 0) return;
	    nodecnt++;
	    if (d > 0 || *s < p->Value)
	        NearSearch (p->LowChild, s, d);
	    if (p->Value == 0) {
	        if ((int) strlen(s) <= d)
	            srcharr[srchtop++] =(char *) p->EqualChild;
	    }
	    else
	        NearSearch(p-> EqualChild, *s ? s+1:s,
	                                  (*s==p->Value) ? d:d-1);
	    if (d > 0 || *s > p->Value)
	        NearSearch (p->HighChild, s, d);
	}
	```

This algorithm has four *if* statements. The first statements checks for bad input arguments or for searching past the end of the tree. The second and fourth *if* statements check if the query character is larger (or smaller) than the node's Value, and recursively search the appropriate child (low or high). The third *if* checks for matches within the desired distance and adds them to the output, and also recursively searches the middle child.

k-NN logically deals with vector-space features, but its implementation for neighbourhood search converts these vectors back into strings. This thesis translated a C# library (de Halleux, 2004) to Java for its implementation.

4.2.3.3 Naïve Bayes Classifier

CRM114 is an open source spam filtering project that includes several machine learning implementations, including Naïve Bayes (Yerazunis, 2007). The CRM114 implementation is translated from C++ to Java for use in this project. Naïve Bayes classification is described in 2.1.2.1.3.

4.2.3.4 Support Vector Machine Classifier

LIBSVM is a software library for support vector classification, regression and distribution estimation (Chang et al., 2007). The original software is written in C++, but LIBSVM has an active community that offers translations in many languages, including Java. LIVSVM's parameter list, shown in Table 4, is a concise summary of its capabilities.

Table 4: LIBSVM Parameter List

Parameter Name	Available Values / Description
SVM Type	C-SVC nu-SVC one-class SVM epsilon-SVR nu-SVR
Kernel Type	Linear: u'*v Polynomial: (gamma*u'*v + coef0)^degree Radial Basis Function: exp(-gamma*|u-v|^2) Sigmoid: tanh(gamma*u'*v + coef0)
Degree	Degree in the kernel function (default = 3)
Gamma	Gamma in kernel function (default = 1/#features)

Coefficient 0	Coefficient 0in kernel function (default = 0)
Cost	C in C-SVC, epsilon-SVR and nu-SVR (default = 1)
Nu	Nu in nu-SVC, one-class SVM, and nu-SVR (default = 0.5)
Epsilon	Epsilon of loss function in epsilon-SVR (default = 0.1)
Cache Size	Cache size in MB (default = 100)
Tolerance	Tolerance of termination criterion (default = 0.001)
Shrinking	Boolean indicates whether to use the shrinking heuristics (default = 1)
Probability Estimates	Boolean indicates whether to train a SVC or SVR model for probability estimates (default = 0)
Weight_i	Set C for class *i* to weight*C in C-SVC (default = 1)

This implementation uses LIBSVM's classification functionality. C-SVC and nu-SVC are both soft margin algorithms (errors are acceptable but are penalized). In C-SVC, C is the capacity constant that is proportional to the penalty. A large C may lead to over fitting. In nu-SVM the soft margin has to lie in the range of zero and one. Nu does not control the trade off between the training error and the generalization error, but has two different roles. It is an upper bound on the fraction of margin errors and is the lower bound on the fraction of support vectors.

4.2.4 Clustering

Carrot^2 is an open source clustering library that includes several algorithm implementations and a pluggable framework for adding new algorithms (Osiński et al., 2005). This project uses two Carrot^2 algorithms and integrates (and contributes to open source) a third.

4.2.4.1 Lingo Clustering

Lingo, which Carrot^2's default clustering algorithm, implements the paradigm: “description comes first” (Osiński et al., 2004). Before clustering documents, Lingo determines the descriptions, or clusters, that are relevant to the document set before assigning each document into ones of these clusters. The four step algorithm prepares the document set, extracts key indicators, creates labels and clusters documents, and then adds meta data to each cluster as shown below in Algorithm 3. The cluster label induction in Step 3 is conceptually similar to k-NN classification.

Algorithm 3: Lingo Clustering

Inputs	Document set to cluster, $\boldsymbol{D}$
Output	List of labelled clusters

Algorithm

```
{Step 1: Pre-Processing}
for all $d \in D$ do
  perform text segmentation of $d$; {Detect word boundaries etc.}
  if language of $d$ recognized then
    apply stemming and mark stop-words in $d$;
  end if
end for

{Step 2: Frequent Phrase Extraction}
concatenate all documents;
$\mathcal{P}_c \leftarrow$ discover complete phrases;
$\mathcal{P}_f \leftarrow p: \{p \in \mathcal{P}_c \wedge frequency(p) > Term\ Frequency\ Threshold\}$;

{Step 3: Cluster Label Induction}
$A \leftarrow$ term-document matrix of terms not marked as stop-words and
      with frequency higher than the $Term\ Frequency\ Threshold$;
$\Sigma, \mathrm{U}, \mathrm{V} \leftarrow SVD(A)$; {Product of $SVD$ decomposition of $A$}
$k \leftarrow 0$; {Start with zero clusters}
$n \leftarrow rank(A)$;
repeat
  $k \leftarrow k + 1$;
  $q \leftarrow \left(\sum_{i=1}^{k} \Sigma_{\mathrm{ii}}\right) / \left(\sum_{i=1}^{n} \Sigma_{\mathrm{ii}}\right)$;
until $q < Candidate\ Label\ Threshold$;
$P \leftarrow$ phrase matrix for $\mathcal{P}_f$;
for all columns of $U_k^T P$ do
  find the largest component $m_i$ in the column;
  add the corresponding phrase to the $Cluster\ Label\ Candidates$ set;
  $labelScore \leftarrow m_i$;
end for

calculate cosine similarities between all pairs of candidate labels;
identify groups of labels that exceed $Label\ Similarity\ Threshold$;
for all groups of similar labels do
  select one label with the highest score;
end for

{Step 4: Cluster Content Discovery}
for all $L \in Cluster\ Label\ Candidates$ do
  create cluster $C$ described with $L$;
  add to $C$ all documents whose similarity to $C$ exceeds the
    $Snippet\ Assignment\ Threshold$;
end for
```

	put all unassigned documents in the “Others” group; {STEP 5: Final Cluster Formation} for all clusters do $clusterScore \leftarrow labelScore \times \|\|C\|\|$; end for

4.2.4.2 Suffix Tree Clustering

Suffix tree clustering (STC) is unique in that a single document can appear in multiple clusters (Zamir et al., 1999). Algorithm 4 shows the summary below.

Algorithm 4: STC Clustering

Inputs	Document set to cluster, $\boldsymbol{D}$
Output	List of labelled clusters
Algorithm	1. Construct suffix tree 2. Score nodes in the tree 3. Find clusters 3.1 Construct an undirected graph whose vertices are nodes in the suffix tree. An arc exists between two nodes if a. either node is a top scoring node b. the number of documents in the intersection of two nodes is at least half of the bigger of the two nodes 3.2 Each connected component of this graph is a cluster

Constructing the suffix tree is like building an inverted index of phrases in the document set. Each phrase is associated with a collection of documents. A node’s score is based on the number of documents in its sub-tree. For each node N in the tree, let $D(N)$ be the set of documents in N’s sub-tree, and $P(N)$ be the phrase label of N. The score of the node is $s(N) = |D(N)| \times f(|P(N)|)$, where $f(K) = K$ for $K \leq 6$, and $f(K) = 6$ for $K > 6$. The top 500 scoring nodes and all nodes that have intersecting documents with these nodes are candidates to become part of a cluster. A graph is constructed by comparing each pair of candidate nodes. If the size of the intersection between any two of these nodes is greater than half the size of either node then an arc is drawn between the two nodes in the graph. After all candidate nodes have been compared, each connected component in the graph forms a cluster.

4.2.4.3 URL Clustering Algorithm

The previous two clustering algorithms were designed for documents, not URLs. This project includes a simple clustering algorithm specifically designed to cluster URLs, not documents, as shown in Algorithm 5. The runtime performance of this algorithm is exponential.

Algorithm 5: URL Clustering

Inputs	Document set to cluster, $\boldsymbol{D}$
Output	List of labelled clusters

Algorithm

```
$threshold \leftarrow mean(distance\ between\ each\ document\ pair)$
for all document pairs $d_1, d_2 \in D$ do
  $distance \leftarrow Levenshtein(d_1, d_2)$
  if $distance < threshold$ then
    if $d_1$ or $d_2$ belong to an existing cluster
      merge all clusters that $d_1$ or $d_2$ belong to
    else
      create a new cluster with $d_1, d_2$ as its content
    end if
  end if
end for
```

This algorithm was contributed to Carrot[2] under the name *ByFullUrlClusteringAlgorithm*. The output clusters are sorted in ascending order by the standard deviation of the length of the URLs within each cluster. An output cluster is assumed to contain links to detail pages if its URL length standard deviation is less than 0.001 (an empirically chosen threshold). If no *good* clusters are found, the distance threshold is loosened (using a multiple of the distance standard deviation) and the algorithm is run again. This is repeated until a good cluster is found, or until 20 attempts have been made. The measure of success for this algorithm is if exactly one output cluster contains links to detail pages. Both other cases are considered failures:

1) More than one output cluster meets the detail pages assumption (regardless of how many clusters are actually created)
2) No output clusters meet the details page assumption (also regardless of how many clusters are actually created)

URL Clustering (UC) is similar to Affinity Propagation Clustering (APC) (Frey et al., 2007) because both create clusters based on pair wise distance between every pair of points. APC uses maximum similarity to determine clusters. URL Clustering creates clusters based on similarity above some threshold, not just the maximum.

5: RESULTS

This thesis proposes an adaptive classification algorithm to improve document classification of rich web documents. A series of experiments was run to test this hypothesis using real estate listings as the target document type. A baseline result is needed to observe improvement. The first set of experiments optimizes the configuration of a static real estate listing classifier by applying different normalization, feature selection, and machine learning algorithms. This provides both a baseline and configuration for the site-specific classifier created by the adaptive algorithm. Each baseline configuration was tested using precision and recall to measure success of identifying real estate listings. The class labels for classifying real estate listings are

$$\begin{cases} H = real\ estate\ listing \\ -H = other\ page\ from\ website \end{cases}$$

The next set of experiments applies the adaptive classification algorithm, including parameter variation to the supervised learner and varying the clustering algorithm used to select training sets. A set of test sites is chosen and the baseline detail page classifier’s results are compared against the site-specific detail page classifier created by the adaptive algorithm. The adaptive algorithm evaluation includes testing the algorithm’s ability to produce a site-specific classifier. Producing a site-specific classifier is controlled by two mechanisms within the algorithm:

1) ability to identify search result pages, and

2) ability to cluster links to detail pages from the search result page

5.1 Static Document Classification

The purpose of building a baseline real estate listing classifier is to test whether ADC can improve classification. The tuned choice of parameters (including learning algorithm) will be re-used twice in ADC: first to build the search result page classifier that is passed in, and second to build site-specific classifiers at runtime. Optimizing parameter selection for the static classifier then re-using these parameters in the proposed algorithm suggests that the baseline results are an upper bound of what can be expected, and the proposed algorithm’s results have room for improvement.

The first step in training a supervised learning algorithm is to build the training set. Preliminary experiments and current research indicate that time spent “tuning” the training set was more productive than time spent tuning the learner. Approximately three thousand positive and negative training documents were collected from across five hundred websites, and then sampled to construct training and test sets. The top ten most visited real estate websites in the United States account for approximately 35% of traffic to all American real estate websites (Experian, 2011). The next hundred websites account for approximately the same traffic, and the remaining traffic is split among thousands of small websites. In the context of American real estate websites, a sample using five hundred websites is representative.

Diversity and correctness were the quality metrics for collecting documents. To a learner, many documents from a single website can look the same. The number of documents per website was limited to five positive and five negative examples to reduce the influence of any one website on the training set (the total size of the training set is discussed in 5.1.1). The five positive documents were made as different as possible by selecting a variety of listings (high price and low price, single family homes and land, etc). The five negative documents consisted of the home page, a contact page, a search form, and two other pages relevant to each website. Collecting documents is a manual process, so error is inevitable. Each document was screened twice to ensure correct labelling. In addition to labelling documents as positive or negative, websites as a whole were marked by template groups. Several prominent website vendors offer slight variations of the same website to real estate agents. For the purpose of machine learning these websites are all identical.

A utility was written to randomly select three document sets, each consisting of a positive and negative subset, from the superset of training documents. One set was used to train the learner, and the other two sets were used to test it. The sets were chosen using the following rules:

1. Choose *m* documents for each positive set, and *m* for each negative set, where 2*m* is the desired size of the feature set
2. Choose at most three positive and three negative documents from each template group (websites that look the same).
3. Each template group can only appear in one of the sets.

The intention behind these rules is to mimic real-world scenarios as closely as possible by eliminating training data from the test sets. Classifiers that are part of large scale crawlers encounter a negligible number of documents from their training sets at runtime. Using two distinct test sets helps eliminate manual collection errors such as not recognizing some collection of websites as belonging to a single group.

The following sections present experiments and results for optimizing classification parameters. These experiments were run iteratively to retest prior optimizations given some improved downstream configuration. The results of each parameter's most significant optimization are shown.

5.1.1 Training Set Size

The size of the training set plays an important role in trade-off between generalization and over-fitting. Figure 19 shows the classification accuracy of a real estate listings classifier against its training set size. Classification accuracy was measured against a set of documents whose domains and templates were not in the training set.

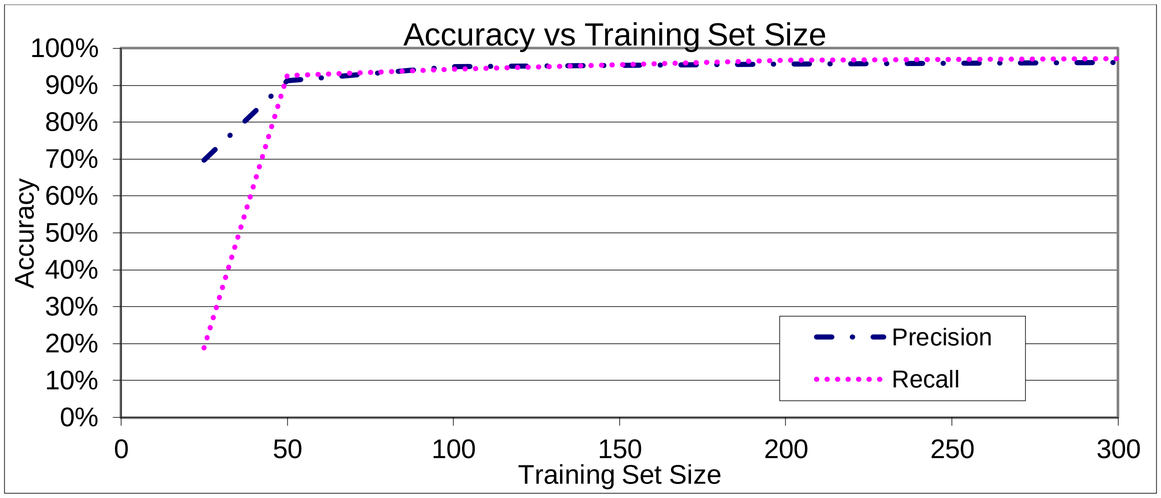


Figure 19: Accuracy vs. Training Set Size for Baseline Real Estate Listing Classifier

Training sets up to 950 documents in size were tested, but showed no improvement over the presented data.

5.1.2 Pre Processing

The following sections present four measurements for each parameter change:

1. True positive: the percent of documents accurately classified into the desired set.
2. False positive: the percent of documents inaccurately classified into the desired set.
3. True negative: the percent of documents accurately classified into the undesired set.
4. False negative: the percent of documents inaccurately classified into the undesired set.

Depending on the system, a different type of error is acceptable. In static document classification false positives are preferred to false negatives. In adaptive document classification's SRP classifier the opposite is true.

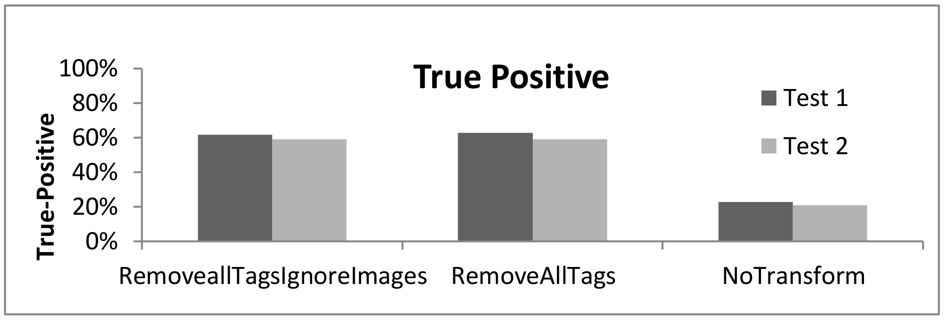
5.1.2.1 HTML Analysis

Figure 20 (a) – (d) shows results for the following HTML analyzers:

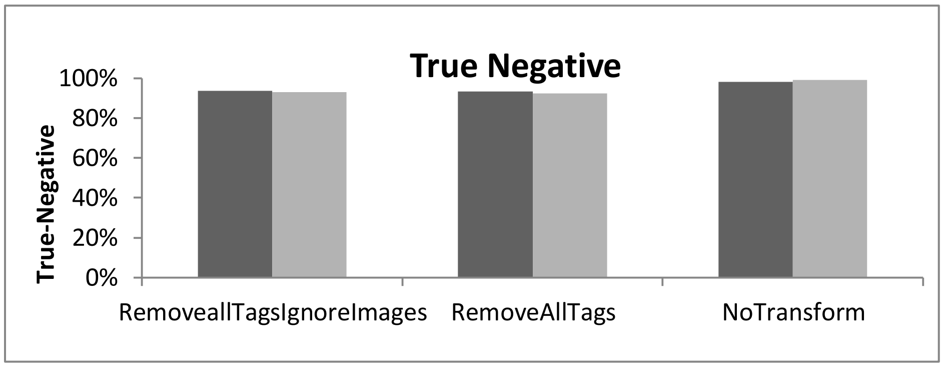
- **NoTransform**: Baseline test with no HTML analysis.
- **RemoveAllTags**: Remove all HTML tags, leaving only each element's value behind.
- **RemoveallTagsIgnoreImages**: Remove all tags except for images.

Each setting was tested with two independent test sets (labelled Test 1 and Test 2).

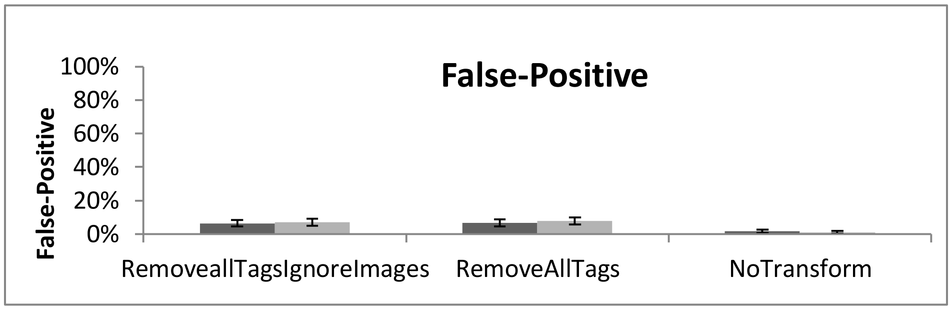
(a)



(b)



(c)



(d)

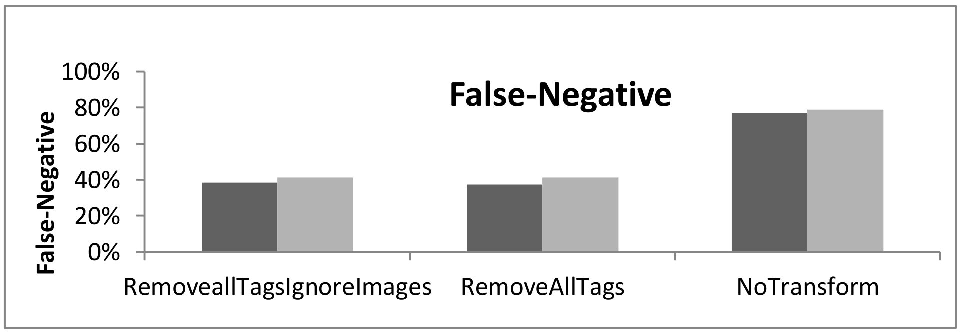


Figure 20: HTML Analysis

Figure 20 shows the importance of removing HTML mark-up from each document. Customizing the behaviour of individual HTML elements (such as images) had little effect on the overall performance.

5.1.2.2 Part of Speech Analysis

Figure 21 (a) – (d) shows results for the following Part-of-Speech analyzers:

- **NT (no transform)**: Baseline test with no part-of-speech analysis.
- **NNDV (nouns, numbers, dollars and verbs)**: Keep only nouns, numbers, dollars and verbs.
- **NND (nouns, numbers and dollars)**: Keep only nouns, numbers and dollars.
- **FT (final transform)**: TODO: describe

As with HTML analysis, each setting was tested with two independent test sets (labelled Test 1 and Test 2).

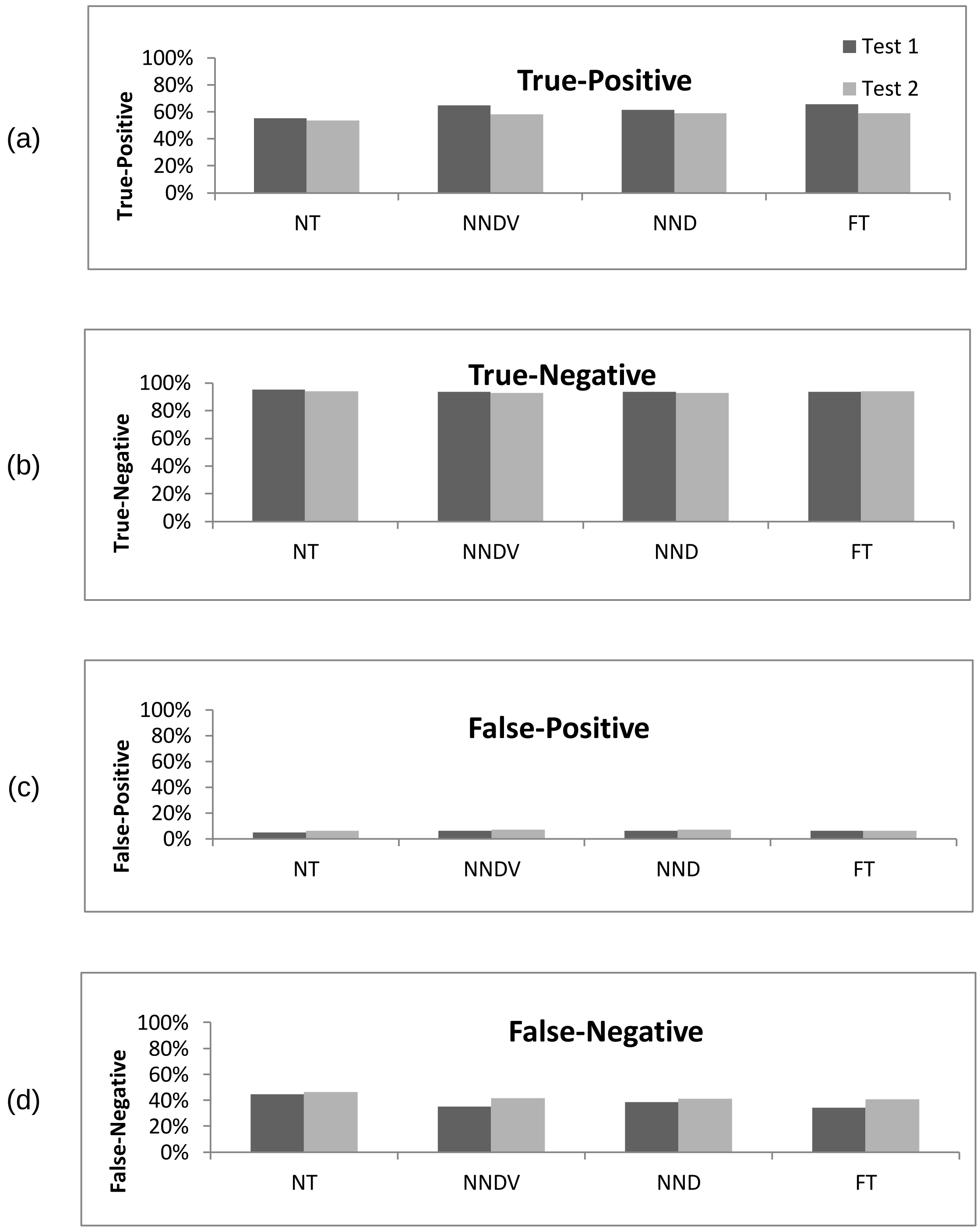


Figure 21: Part of Speech Analysis

The effect of part of speech analysis is less than HTML analysis, but does improve classification by using only some parts of speech.

5.1.2.3 Stop Word Analysis

Figure 22 (a) – (d) shows the effect of applying stop words to documents before classification.

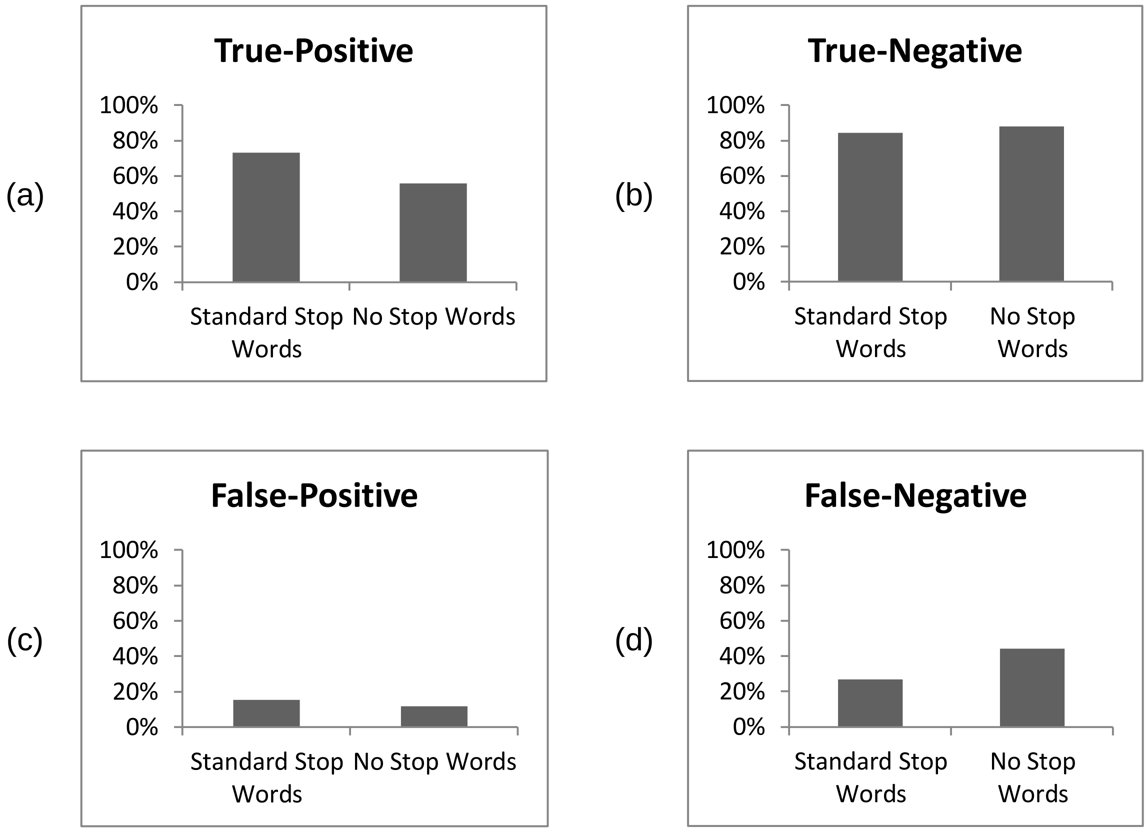


Figure 22: Stop Word Analysis

5.1.2.4 Stemming Analysis

Figure 23 (a) – (d) shows the effect of stemming words before classification.

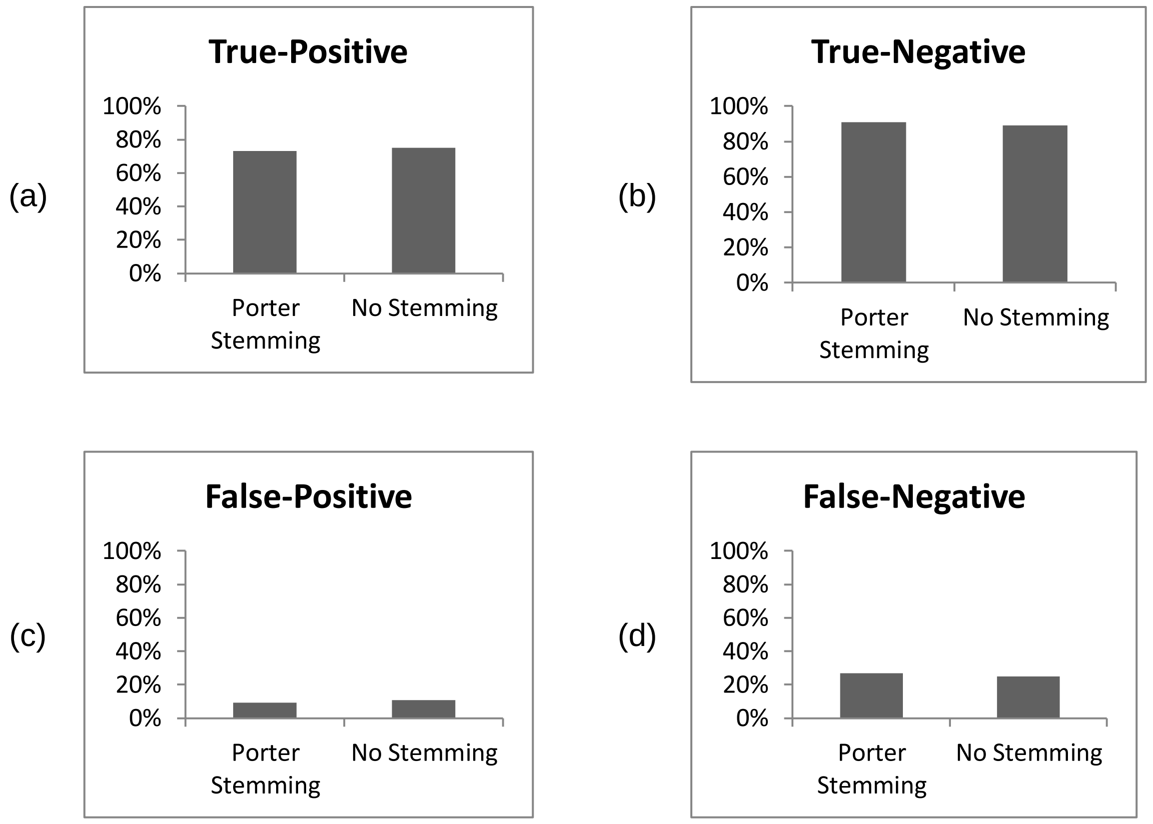
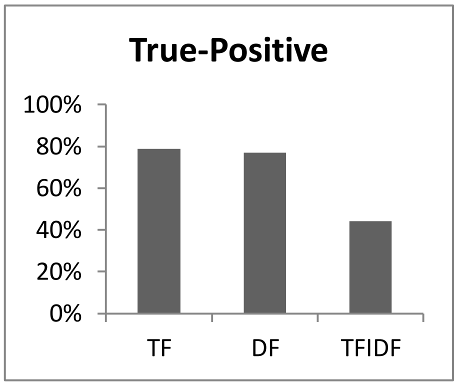


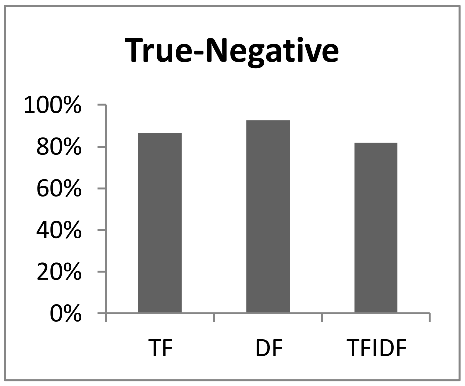
Figure 23: Stemming Analysis

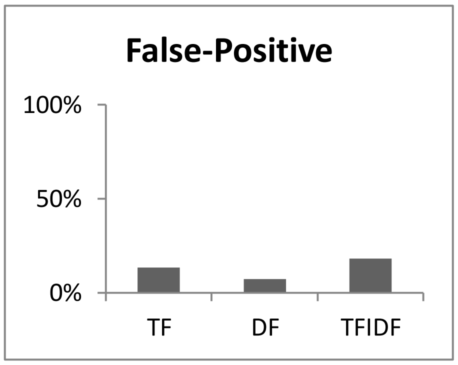
5.1.2.5 Feature Weighting

Figure 24 (a) – (d) shows results for the following phrase weighting algorithms:

- **TF**: Term Frequency.
- **DF**: Document Frequency.
- **TFIDF**: Term Frequency Inverse Document Frequency.

(a)


(b)


(c)


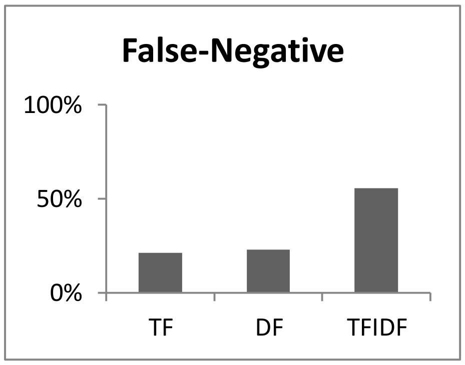
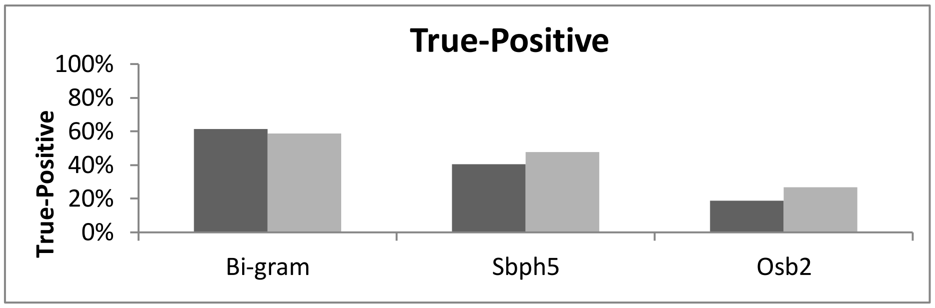
(d)


Figure 24: Feature Weighting

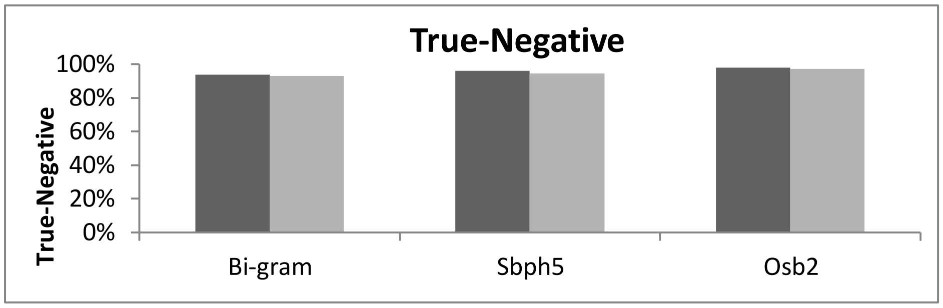
5.1.3 Feature Selection

Figure 25 (a) – (d) shows the results of applying various feature selection algorithms.

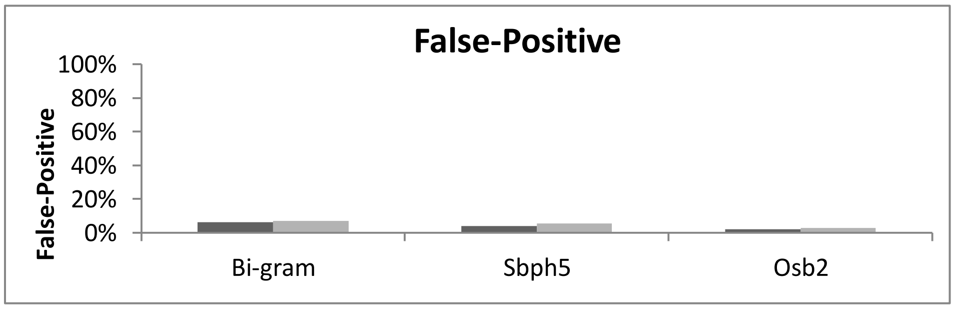
(a)



(b)



(c)



(d)

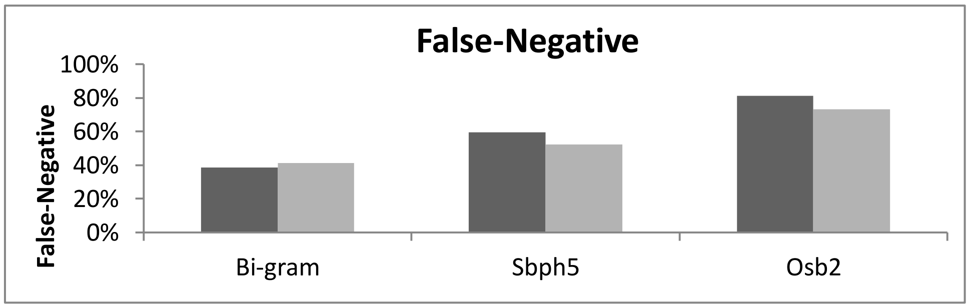


Figure 25: Feature Selection

5.1.4 Learning Algorithms

The best pre-processing parameters for one machine learning algorithm do not necessarily make them the best parameters for another algorithm. Figure 26 (a) – (c) presents optimized configuration results for three algorithms:

- **k-NN**: k-Nearest Neighbour.
- **NB**: Naïve Bayes.
- **SVM**: Support Vector Machines.

Each algorithm is trained with the same set of data, and tested against three independent test sets.

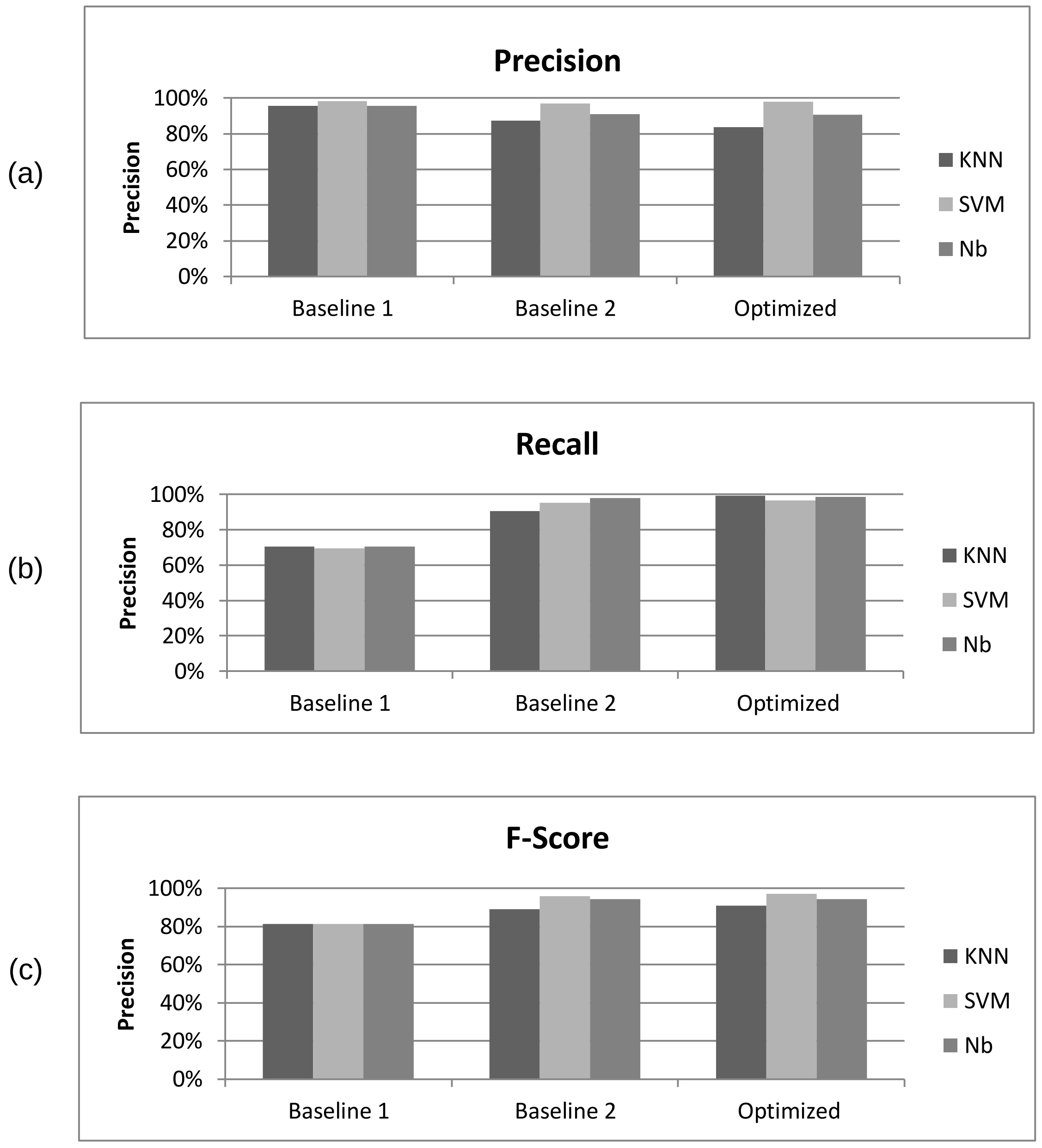


Figure 26: Algorithm Comparison

5.2 Adaptive Document Classification

The best performing parameters from 5.1 were used in all the following tests, the most significant of which was using SVM as the machine learning algorithm. Seven real estate listings websites were chosen to compare adaptive document classification to the optimized static results obtained earlier. A search result page classifier was built using the optimized static method, but trained to detect search result pages instead of listing detail pages.

5.2.1 Baseline Static Classification

Table 5 presents the chosen seven listing websites, along with classification accuracy for the optimized SVM static classifier for real estate listings. Note that Average* indicates the average when ignoring the best and worst results.

Table 5: Baseline Static Classification

Site	*Precision*	*Recall*	*F-Score*
ApartmentHunterz	99.06%	100.00%	99.53%
BillClarkHomes	66.67%	4.21%	7.92%
JpmStGeorge	89.87%	86.59%	88.20%
LasVegasLiving	66.30%	100.00%	79.74%
ScottFindlay	98.10%	100.00%	99.04%
SeattleRentals	91.60%	100.00%	95.62%
WorldProperties	95.80%	66.31%	78.37%
Average	86.77%	79.59%	78.34%
Average*	88.07%	88.75%	86.55%

5.2.2 Search Result Page Classification

The search result page classifier was built using the same configuration as the baseline detail page classifier, but was given different training examples from the same set of sample websites. Its accuracy is better than the detail page classifier’s accuracy, as shown in Table 6. This is expected based on the assumptions stated earlier: data elements in the summary view of a given vertical are consistent across most websites. The same is not true for detail pages.

Table 6: Search Result Page Classification

Site	*Precision*	*Recall*	*F-Score*
ApartmentHunterz	98.59%	99.29%	98.59%
BillClarkHomes	100.00%	100.00%	100.00%
JpmStGeorge	90.91%	95.24%	90.91%
LasVegasLiving	90.00%	94.74%	90.00%
ScottFindlay	88.89%	94.12%	88.89%
SeattleRentals	94.90%	97.38%	94.90%
WorldProperties	98.35%	99.17%	98.35%
Average	94.52%	97.13%	94.52%
Average*	96.21%	98.04%	96.21%

5.2.3 Detail Page Link Clustering

The output of the search result page classifier is mined for clusters of links to detail pages. Several clustering algorithms were tested and only *ByFullUrlClusteringAlgorithm* produced useful results. Most clustering algorithms are designed for “richer” input (i.e. full documents). Table 7 shows *ByFullUrlClusteringAlgorithm’s* accuracy on the test websites.

Table 7: Link Clustering Accuracy

Site	*Able to find detail page link cluster*
ApartmentHunterz	Yes
BillClarkHomes	Yes
JpmStGeorge	Yes
LasVegasLiving	No
ScottFindlay	Yes
SeattleRentals	Yes
WorldProperties	Yes

ByFullUrlClusteringAlgorithm has one success condition: a cluster is found that looks like it contains links to detail pages. If more or less than one such cluster are found then the algorithm does not return any clusters.

5.2.4 Adaptive Classification Results

Table 8 presents the classification results for the system proposed by this thesis as a direct comparison against Table 5. Adaptive site-specific detail page classifiers were constructed using the same configuration and SVM classifier used in both the baseline detail page classifier and the SRP classifier.

Table 8: Adaptive Document Classification

Site	*Precision*	*Recall*	*F-Score*
ApartmentHunterz	100.00%	99.46%	99.73%
BillClarkHomes	100.00%	100.00%	100.00%
JpmStGeorge	100.00%	84.15%	91.39%
LasVegasLiving	100.00%	0.00%	0.00%
ScottFindlay	100.00%	100.00%	100.00%
SeattleRentals	100.00%	100.00%	100.00%
WorldProperties	100.00%	98.77%	99.38%
Average	100.00%	83.20%	84.36%
Average*	100.00%	94.26%	95.81%

5.2.5 Performance Analysis

Adaptive document classification enhances the classification piece of the general purpose crawling design presented earlier in Figure 5. The runtime performance of the entire system is limited by its slowest piece, which is web crawling. Efficient web crawling is bottlenecked by network bandwidth (Najork et al., 2001). In steady-state operation, adaptive document classification does not change the system’s performance characteristics; performance is only affected while the site-specific classifier is being built. Building the adaptive classifier requires uses additional resources to identify and collect possible training samples. The identification process uses a classifier similar in complexity to the adaptive classifier that is built for each website. The runtime penalty is paid immediately after the site-specific classifier is built, and all previously crawled web pages are classified. Document classification, even in batch mode, is still significantly faster than downloading web pages. Collecting training samples uses additional memory to avoid re-fetching training documents for re-classification. Websites that contain either very deep or no acceptable training examples will cause the crawler to use the most amount of memory. The websites explored during the writing of this thesis did not cause excessive memory usage. A simple work-around is to impose artificial memory usage limits, which would require the crawler to re-fetch some (or all) documents visited before the site-specific classifier is built. The additional work imposed by adaptive document classification does not significantly change runtime performance.

5.3 Discussion

Building a static, site-specific classifier with manually selected training examples is both easy and very accurate (assuming one has time to manually collect all these examples). Adaptive document classification shows that similar accuracy can be obtained using an automated process for selecting training examples. For the selected set of websites adaptive document classification improves classification f-score by about 9%, with a significant gain in both precision and recall. Improvements in classification become exponentially harder as accuracy increases, so achieving this same improvement using only static methods is unlikely.

Comparing Table 5 and Table 8 shows clear patterns developing. The ratio of precision to recall indicates which of the two measures a classifier is tuned for. Table 9 shows this ratio for the benchmark and proposed algorithms, where Recall/Precision* ignores the best and worst results.

Table 9: Recall to Precision Ratio Comparison

	Benchmark	*Proposed*
Recall/Precision	0.92	0.83
Recall/Precision*	1.01	0.94

The proposed algorithm is tuned more towards precision than the benchmark algorithm. This is quite intuitive when remembering the training data that was supplied to each. The benchmark classifier is trained using examples from many websites, which causes it to prefer generic features and leads to more false positives. The proposed classifier is trained only with examples from the target website, so it prefers website-specific features including presentation structure, language, and terminology that are all consistent across detail pages within the website. This leads to near-perfect precision.

The proposed algorithm is tuned for precision, but also shows improvement in recall. This can again be explained by analyzing the training sets of both classifiers. The benchmark algorithm requires a diverse set of features to work well across all websites. Inevitably, the classifier will be missing some features that are key indicators for some listings or even websites, resulting in false negatives. The proposed algorithms can be much more selective in choosing features; a few very well chosen features can be absolute indicators within a website. Absolute indicators prevent any listing from being missed.

Each classifier has one site that is completely fails on in the test data. For the benchmark classifier, the site is BillClarkHomes. Structurally, this site is consistent with the other sites in that it has search result pages that point to detailed listing pages. The problem is that the detail pages have very sparse content, so the classifier does not have many features to work with. The proposed classifier has no problems because the structure of the site lets it choose the required site specific training documents. The proposed classifier failed on the LasVegasLiving site. The structure of this website's documents is also consistent with the assumed model (search result pages leading to detail pages). Table 7 shows clustering accuracy per website, and in particular, shows that the clustering algorithm is unable to cluster LasVegasLiving's links. Table 10 shows sample links from the website that do not lend themselves to easily be clustered.

Table 10: Difficult Links for Clustering

URL	*Page Type*
http://www.lasvegasliving.com/adiamo/adiamo_rates.cfm	Not details page
http://www.lasvegasliving.com/adiamo/adiamo_fp.cfm	Not details page
http://www.lasvegasliving.com/adiamo/fp_montellano.cfm	Details page
http://www.lasvegasliving.com/adiamo/fp_sorrento.cfm	Details page

WordProperties is a site that produced good data for both classifiers, but the proposed algorithm significantly outperformed the benchmark algorithm on recall. The training data for the benchmark classifier was only from North

American websites, so it is expected to perform poorly in other contexts where language and terminology are different. The adaptive algorithm’s search result page classifier was also trained with North American data. Search result pages shows concise summary information, and are more consistent across sites than detail pages. In this example, the search result pages are consistent across both North American and international listings. This allows the adaptive algorithm to correctly identify search result pages and build a site-specific classifier that takes international listings into account.

The proposed classifier demonstrated hit-or-miss results. If it was able to classify even a single listing detail on a website then it scored well for all documents on the website. However, there is no guarantee that even one listing will be correctly classified. There are two classes of failure for the adaptive algorithm: 1) the SRP classifier fails to detect any search result pages, meaning that the site-specific classifier cannot be built, and 2) the SRP classifier works correctly, but the link clustering algorithm fails again meaning that the site-classifier cannot be built. Both classes of failure are easier to fix than improving a static detail page classifier. Re-training the SRP classifier with examples from previously failed websites is unlikely to break existing “good” classification results because of the concise nature of the data. New heuristics for link clustering can target only specific sites (or patterns of links) to maintain existing desirable behaviour.

6: CONCLUSION

Building a website-specific classifier produces high accuracy classification results within a given website. The results above indicate that the proposed adaptive document classification algorithm can be used to dynamically generate this high accuracy solution resulting in a significant improvement over general-purpose classifier accuracy. The idea that structure can be used to improve classification accuracy is not a new one (Cohen, 2002), but this particular approach is new to the best of this author's knowledge.

6.1 Limitations

The sample implementation includes several limitations. The sample size for static document classification and parameter selection optimization was sufficiently large to call the results generalizations. Adaptive document classification was tested with a set of only seven websites due to the manual work involved in testing the results. The set of test websites was selected using similar guidelines that were used for selecting test sets for static classification: only one website from any template group appeared in the set. The classes of problems encountered in the test set was an accurate representation of what is expected on a "random" real estate listings website. The results presented are accurate and predictive of other sites; the uncertainty is how many sites are BillClarkHomes versus how many are like LasVegasLiving.

Another limitation of the sample implementation is the collection of link clustering algorithms used. The selected clustering algorithm (*ByFullUrlClusteringAlgorithm*) worked well for the test sets, but was not based on the academic community's current standards. Broader applications of adaptive document classification would likely exposes weaknesses in *ByFullUrlClusteringAlgorithm*.

A third limitation of this work is intentional: the classification parameters were not optimized for adaptive document classification; instead they were inherited from the best performing static classifier. The intention was to demonstrate that well chosen training sets are significantly more valuable to a classifier than incremental improvements in the underlying mathematical models. The results agreed with approach.

6.2 Future Work

The future work falls in to three classes: 1) overcome the described limitations of this implementation, 2) implement the full system design of the solution, and 3) improve known vulnerabilities in the existing implementation.

This work's most significant limitation is the sample size. The existing solution should be verified against more real estate listing websites (including baseline tests using a static document classifier). Real estate listings were chosen as the sample domain, but the proposed algorithm is not domain-specific. Applying adaptive document classification to other domains coincides nicely with increasing the sample size.

The system design chapter describes an end-to-end solution that includes more intelligent web crawling than was implemented here, and a data extraction system to consume the output of the adaptive classifier. The data extraction component has more unknowns, so should be implemented next. Similar to document classification, data extraction benefits strongly from data normalization. When chained together with classification, many of the normalization techniques could and should be shared across both components.

Adaptive document classification exhibits hit-or-miss behaviour tied to two inputs: 1) the search result page classifier, and 2) the link clustering algorithm. Both inputs include high confidence thresholds to allow ADC to execute. Dynamically reducing the confidence threshold when an error condition is detected would allow ADC to provide some output. Experiments are needed to understand the usefulness of such output.

REFERENCE LIST

Bach, Francis R. and Michael I. Jordan. Learning Spectral Clustering. In *Advances in Neural Information Processing Systems*, Vol. 16. 2003. MIT Press.

Baldridge, Jason, Thomas Morton, Gann Bierner and Jörn Kottmann. (2003, Mar. 21). *The OpenNLP Maximum Entropy Package* [Online]. Available: http://opennlp.sourceforge.net.

Bentley, Jon L. and Robert Sedgewick. 1997. Fast algorithms for sorting and searching strings. In *SODA '97 Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*.

Berkhin, Pavel. 2002. *Survey of clustering data mining techniques*. Technical report, Accrue Software.

Bialecki, Andrzej. (2009, Nov. 5). *Web-scale search engine toolkit: Today and tomorrow* [Online]. Available: Apache Conference 2009, Apache Software Foundation: http://wiki.apache.org/nutch/Presentations?action=AttachFile&do=get&target=apachecon09.pdf

Bomhardt, Christian. 2004. NewsRec, a SVM-driven Personal Recommendation System for News Websites. In *Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*.

Brown, David J. 2010. *Developing an Automatic Document Classification System*. Technical Memorandum, Defence Research and Development Canada.

Chakrabarti, Soumen. 2000. Data mining for hypertext: a tutorial survey. In *ACM SIGKDD Explorations*, 1-11.

Chang, Chih-Chung and Chih-Jen Lin (2007, Apr 1). *LIBSVM - A Library for Support Vector Machines* [Online]. Available: http://www.csie.ntu.edu.tw/~cjlin/libsvm/.

Chen, Yan, Lili Qiu, Weiyu Chen, Luan Nguyen and Randy H. Katz. 2002. Clustering Web Content for Efficient Replication. In *Proceedings of the 10th IEEE International Conference on Network Protocols (INCP)*.

Chekuri, Chandra, Michael H. Goldwasser, Prabhakar Raghavan and Eli Upfal. 1997. Web Search Using Automatic Classification. In *Proceedings of the 6th International Conference on the World Wide Web*.

Choi, Ben and Xiaogang Peng. Dynamic and hierarchical classification of Web pages. In *Online Information Review*, Vol. 28. 2004. Emerald Group Publishing.

Cohen, William W. 2002. Improving A Page Classifier with Anchor Extraction and Link Analysis. In *Advances in Neural Information Processing Systems 15, Papers from Neural Information Processing Systems*.

Cortes, Corinna, Vladimir Vapnik. 1995. Support vector networks. *Machine Learning*, 20:273–297.

Dong, Jianxiong, Ching Y. Suen and Adam Krzyzak. 2008. Effective Shrinkage of Large Multi-Class Linear SVM Models for Text Categorization. In *Proceedings of International Conference on Pattern Recognition*.

de Halleux, Jonathan. (2004, Jan. 27). *Ternary Search Tree Dictionary in C#: Faster String Dictionary!* [Online]. Available: http://www.codeproject.com/KB/recipes/tst.aspx

Experian Hitwise United States. (2011). *Top 10 visited Real Estate sites* [Online]. Available: http://www.hitwise.com/us/datacenter/main/dashboard-10133.html

Frey, Brendan J., and Delbert Dueck. 2007. Clustering by Passing Messages Between Data Points. In *Science* 315: 972-76.

Hastie, Trevor, Robert Tibshirani and Jerome Friedman, *The elements of statistical learning: Data mining, inference and prediction*. Springer Verlag, 2001.

Hofmann, Thomas, Jan Puzicha, and Michael I. Jordan. Unsupervised learning from dyadic data. In *Advances in Neural Information Processing Systems*, Vol. 11. 1999. MIT Press.

Hsu, Chih-Wei and Chih-Jen Lin. A Comparison of Methods for Multiclass Support Vector Machines. In *IEEE Transactions on Neural Networks*, Vol. 13. 2002.

Joachims, Thorsten. 1998. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. In *Proceedings of the 10th European Conference on Machine Learning*.

Jericho, Martin. (2009, Jun 11). *Jericho HTML Parser* [Online]. Available: http://jericho.htmlparser.net.

Kroch, Anthony. 1989. Reflexes of Grammar in Patterns of Language Change. In *Language Variation and Change* 1:199–244.

Kuhn, Harold W., and Albert W. Tucker. 1951. Nonlinear programming. In *Proceedings of 2nd Berkeley Symposium*: 481-492.

Lawson, Charles. L. and Richard. J. Hanson. *Solving Least Squares Problems*. Prentice-Hall, 1974.

Liu, Rey-Long and Yun-Ling Lu. 2002. Incremental context mining for adaptive document classification. *In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*.

MacQueen, James B. 1967. Some Methods for classification and Analysis of Multivariate Observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*.

Mitchell, Tom M. *Machine Learning*. The Mc-Graw-Hill Companies, Inc., 1997.

Murray, Brian H. and Alvin Moore. 2000. *Sizing the internet*. White Paper, Cyveillance, Inc. http://www.cs.toronto.edu/~leehyun/papers/Sizing_the_Internet.pdf.

Najork, Marc and Allan Heydon. 2001. High Performance Web Crawling. SRC Research Report 173, Compaq Systems Research Center.

Ng, Andrew and Michael I. Jordan. On Discriminative vs. Generative Classifiers: A comparison of logistic regression and Naïve Bayes. In *Advances in Neural Information Processing Systems*, Vol. 14. 2002. MIT Press.

Nilsson, Nils J. (1998, Nov. 3). *Introduction to Machine Learning: An Early Draft of a Proposed Textbook* [Online]. Available: Department of Computer Science, Stanford University: http://ai.stanford.edu/~nilsson/MLBOOK.pdf

Osiński, Stanisław, Jerzy Stefanowski and Dawid Weiss. 2004. Lingo: Search Results Clustering Algorithm Based on Singular Value Decomposition. In *Proceedings of the International IIS.*

Osiński, Stanisław and Dawid Weiss. 2005. Carrot2: Design of a Flexible and Efficient Web Information Retrieval Framework. In *Proceedings of the third International Atlantic Web Intelligence Conference (AWIC 2005)*.

Potamias, George. 2001. Adaptive Classification of Web Documents to Users Interests. In *Proceedings of the 8th Panhellenic conference on Informatics*.

Rodriguez, Carlos C. (2004, Oct. 24). *The Kernel Trick* [Online]. Available: Department of Mathematics and Statistics, University at Albany: http://omega.albany.edu:8008/machine-learning-dir/notes-dir/ker1/ker1-l.html

Rogati, Monica and Yiming Yang. 2002, High-Performing Feature Selection for Text Classification. In *Proceedings of the 11th international conference on Information and Knowledge Management*.

Salton, Gerard and Christopher Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing & Management*, Vol. 24. 1988. Elsevier Ltd.

Shao, Fubo, Guoping He, and Xin Zhang. 2008. An Improved Algorithm for Multiclass Text Categorization with Support Vector Machines. In *Proceedings of International Symposium on Computational Intelligence and Design*.

Tan, Chade-Meng, Yuan-Fang Wang and Chan-Do Lee. 2002. The Use of Bigrams to Enhance Text Categorization. In *Information Processing and Management: An International Journal*.

Torunski, Lars. (2009, Jun. 7). *Smart and Simple Web Crawler - v1.3* [Online]. Available: java.net: https://crawler.dev.java.net

USPS National Customer Support Center. (1998). *Official USPS Abbreviations* [Online]. Available: http://www.usps.com/ncsc/lookups/usps_abbreviations.html

Valpola, Harri. 2000. Bayesian Ensemble Learning for Nonlinear Factor Analysis. Ph.D. thesis, Neural Networks Research Centre, Helsinki University of Technology.

van Rijsbergen, Keith. *Information Retrieval* (2nd Ed.). Butterworths, 1979.

van Rijsbergen, Keith, Stephen E. Robertson and Martin F. Porter. 1980. New models in probabilistic information retrieval. London: British Library.

Vapnik, Vladimir. *The Nature of Statistical Learning Theory* (2nd Ed.), Springer Verlag, 2000.

Weston, Jason, Sayan Mukherjee, Olivier Chapelle, Massimiliano Pontil, Tomaso Poggio and Vladimir Vapnik. Feature Selection for SVMs. In *Advances in Neural Information Processing Systems*, Vol. 13. 2000. MIT Press.

Yang, Yiming. 1994. Expert network: effective and efficient learning from human decisions in text categorization and retrieval. In *Proceedings of SIGIR-94, 17th ACM International Conference on Research and Development in Information Retrieval*.

Yang, Yiming and Xin Liu. 1999. A Re-Examination of Text Categorization Methods. In *Proceedings of SIGIR-99, 22nd ACM International Conference on Research and Development in Information Retrieval*.

Yang, Yiming, Sean Slattery and Rayid Ghani. 2002. A Study of Approaches to Hypertext Categorization. In *Journal of Intelligent Information Systems*.

Yerazunis, William S. 2003. Sparse Binary Polynomial Hashing and the CRM114 Discriminator. In *Cambridge Spam Conference Proceedings*.

Yerazunis, William S. (August 10, 2007). *CRM114 - the Controllable Regex Mutilator* [Online]. Available: http://crm114.sourceforge.net.

Zamir, Oren and Oren Etzioni. 1999. Grouper: a dynamic clustering interface to Web search results. In *Proceedings of the eighth international conference on World Wide Web*.

Zhu, Xiaojin and Andrew B. Goldberg. 2009. *Introduction to Semi-Supervised Learning*. Morgan and Claypool Publishers.