Karina Palyutina

Machine learning inference of search engine heuristics

Part II Project

St Catharine's College

April 4, 2013

Proforma

Name:	**Karina Palyutina**
College:	**St Catharine's College**
Project Title:	**Machine learning inference of search engine heuris**
Examination:	**Part II Project**
Word Count:	
Project Originator:	Dr Jon Crowcroft
Supervisor:	Dr Jon Crowcroft

Original Aims of the Project

Work Completed

Special Difficulties

Declaration of Originality

I, Karina Palyutina of St Catharine's College, being a candidate for Part II of the Computer Science Tripos , hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

1. Introduction

1.1. Motivation

This project is inspired by increasing importance of search engine rankings. Today major search engines given a query return web pages in an order determined by secret algorithms. Such algorithms are believed to incorporate multiple unknown factors. For instance, Google claims to have over 200 unique factors that influence a position of a webpage in the search results relative to a query [1]. Only a handful of these factors are disclosed to the webmasters in the form of very general guidelines. Moreover, the Google algorithm in particular is updated frequently. However, most of the knowledge around the area amounts to speculation. Despite the fact that it is possible to pass a vast number of queries through the black box of any existing search engine,the immensity of the search space, and instability of such algorithms make them impossible to reverse engineer.

1.2. Challenges

Machine learning is a natural approach to inferring the true algorithm from a subset of all possible observations. However, applying machine learning techniques to real search engines would be hardly effective, as the dynamic nature of the algorithms and the web as well as lack of meaningful feedback would prevent incremental improvement: when there are as many as 200 features in question, false assumptions made by a learner may have an unpredictable effect on its performance.

More generally, there are certain ambiguities associated with machine learning, which are 'problem-specific'. For example, it proves difficult to decide how much training data is necessary, as well as and selecting it to avoid over/under-fitting[1].

[1] http://www.google.com/competition/howgooglesearchworks.html

Similarly, it is not straightforward which machine learning technique is best for a particular problem.

This project is concerned with application of machine learning techniques to search engines. The aim of the project, in particular, is to explore how machine learning techniques can be used effectively to infer algorithms from search engines. To address the limitations imposed by existing search engines, part of the task is to develop a toy search engine that allows me to comtrol the nature and complexity of used heuristics. Such transparency addresses the problems stated above and, more importantly, allows for useful evaluation of machine learning techniques by providing meaningful feedback.

Even though this study does not attempt to reverse engineer any existing heuristics, the results can be applied to such an ambitious task. Moreover, such a framework is potentially more general and can be used for a range of problems.

1.3. Related Work

2. Preparation

The original aims of the project were defined very broadly, so some further structuring and planning were crucial at the early stages to make sure the goals are understood and subsequently achieved.

The beginning of this chapter introduces the principles of machine learning and the particular techniques implemented during this project in order to familiarise the reader with the field and the research done at the beginning. The rest of the chapter describes the analysis undertaken before the development, in particular, formulating the goals and evaluating the relative importance of parts, as well as associated risks.

2.1. Introduction to Machine Learning

Machine learning constitutes the central part of the project. The field was completely new to me to start with, so research of different techniques was a big part of the preparation.

Machine learning is a vast field and very little is prescribed. The supervised learning approach, where the hidden function is inferred from the labelled training data, best fits our purposes for a number of reasons. Firstly, it applies well to the real-world problem of inferring the heuristics of the real search engines, as we can select training data and obtain the labels simply by querying. Secondly, supervised learning is widely used and offers a variety of flexible techniques.

An interesting known problem with machine learning in general is referred to as the Curse of Dimensionality. In the real world scenario – suppose if we were inferring an existing search engine from the observed query responses – we could not be sure about which features are and which are not relevant to the hidden algorithm. Potentially, some features we might choose to use will increase the dimensionality, but would not actually be used by the real algorithm. Gathering

more features can hurt, as it makes the learner infer nonexistent dependencies. Clearly within the scope of this project we will know at each point which features are or are not used. So observations can be made as to how using more features than the search engine will affect the classifier.

Another major issue that is common to all machine learning methods is over/under-fitting. These refer to the problem of finding balance between the generalized model and the training data at hand. This is yet another source of interest to this project, as we can observe the behaviour of the learner when certain control parameters are changed.

It is generally recommended that the simplest learners are tried first[1]. Of all learners Naive Bayesian is one of the most comprehensible.This in itself is a major advantage according to the Occam's razor principle, which finds ample application in machine learning. Hence, we start with describing the principles of the two machine learning techniques used - Naive Bayes and Support Vector Machines.

2.1.1. Naive Bayes

Naive Bayes is a probabilistic classifier based on the Bayes Theorem. The posterior probability $P(C|\vec{F})$ denotes the probability that a sample page with a feature vector $\vec{F} = (F_1, F_2, \ldots, F_n)$ belongs to class C. The posterior probability is computed from the observable in the training data: the prior probability $P(C)$ – the unconditional probability of a page belonging to the class C, the likelihood $P(\vec{F}|C)$ and the evidence $P(\vec{F})$:

$$P(C|\vec{F}) = \frac{P(C)P(\vec{F}|C)}{P(\vec{F})} \tag{2.1}$$

The simplicity of Bayesian approach owes to the conditional independence assumption: each F_i in $\vec{F}$ is assumed to be independent of one another to get $P(\vec{F}|C) = P(F_1|C) * P(F_2|C) * \cdots * P(F_n|C)$. This leads to a concise classifier definition:

$$\hat{C} = argmax_C P(C) \prod_{i=1}^{n} P(F_i|C) \tag{2.2}$$

where C is the result of classification of a page with feature vector $F_1, F_2, \ldots, F_n$.

In practice, the crude assumption rarely holds and is likely to be violated by our data, as we expect features of pages to be interdependent. However, it has been

shown that Naive Bayes performs well under zero-one loss function in presence of dependencies[2]. This has a few implications for this project, particularly, on evaluation methods

As we have seen, Naive Bayes assigns probabilities to possible classifications in the process of classifying. Even though it generally performs well in classification tasks, these probability estimates are poor [3]. However, despite poor probability estimates, there exist several frameworks, which make use of Bayesian classification and achieve decent performance in ranking. For example, Zhang [8] experimentally found that Naive Bayes is locally optimal in ranking. The paper defines a classifier as locally optimal in ranking a positive example E if there is no negative example ranked after E and vice versa for a negative exaple. A classifier is global in ranking if it is locally optimal for all examples in the example set: in other words, it is optimal in pairwise ranking. It is particularly interesting that the paper discovered that Naive Bayes is globally optimal in ranking on linear functions that have been shown as not learnable by Naive Bayes[1]. Another framework for ranking [7] is based on Placket-Luce model, which reconciles the concepts of score and rank. This framework is based on minimizing the Bayes risk over possible permutations.Existence of such frameworks suggest that Naive Bayes is an adequate choice for a prototype learner.

Although classification is a useful technique to try, it feels more natural to represent our score or rank as real numbers rather than classes. Regression is another approach to machine learning and the next learning technique we explore – Support Vector Regression – is non-probabilistic, to try something as distant from Naive Bayes as possible.

2.1.2. ϵ-Support Vector Regression

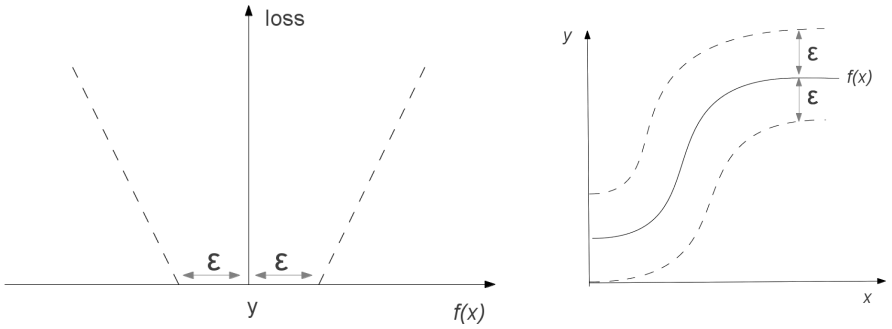
While the binary classification problem has as its goal the maximization of the margin between the classes, regression is concerned with fitting a hyperplane through the given training sequence.

A great advantage of Support Vector machines is that they can perform non-linear classification or regression by using what is referred to as the *Kernel Trick* - an implicit mapping of features into a higher dimensional space, in which the

[1] *M-of-N Concepts* and *Conjunctive Concepts* can't be learnt by Naive Bayes classifier but can be optimally ranked by it according to Zhang [8].

data is linearly separable. The choice of a kernel function is problem specific and the best one is usually decided by experiment.

We begin with the theoretical foundations of Support Vector Regression, which were first proposed by Vapnik **??**.
Define a training sequence as a set of training points $D = \{(\mathbf{x_1}, t_1), (\mathbf{x_2}, t_2), ..., (\mathbf{x_l}, t_l)\}$ where $\mathbf{x_i} \in R^n$ is a feature vector holding features of pages and $\mathbf{t_i} \in R$ is the corresponding ranking of each page.



(a) *ϵ-insensitive loss function* (b) *Predictions with ϵ-SVR*

In simple linear regression the aim is to minimize a regularized error function. We will be using an ϵ-insensitive error function(see Figure 2.1a (a)).

$$E_\phi(y((x) - t)) = \begin{cases} 0 & \text{if } |y(\mathbf{x}) - t)| < \epsilon \\ |y(\mathbf{x}) - t)| - \epsilon & \text{otherwise} \end{cases}$$

where $y((x) = \mathbf{w^T}\phi(\mathbf{x}) + b$ is the hyperplane equation (and so $y(\mathbf{x})$ is the predicted output) and t_n is the target (true) output.

The regression tube then contains all the points for which $y(\mathbf{x_n}) - \epsilon \leq t_n \leq y(\mathbf{x_n}) + \epsilon$ as shown in Figure 2.1a(b).

To allow variables to lie outside of the tube, slack variables $\xi_n \geq 0$and $\xi_n^* \geq 0$ are introduced. The standard formulation of the error function for support vector regression (ref Vapnik 1998) can be written as follows:

$$E = C \sum_{n=1}^{N} (\xi_n + \xi_n^*) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{2.3}$$

E must be minimized subject to four constraints:

$$\xi_n \geq 0, \tag{2.4}$$
$$\xi_n^* \geq 0, \tag{2.5}$$
$$t_n \leq y(\mathbf{x_n}) + \epsilon + \xi_n, \tag{2.6}$$
$$t_n \geq y(\mathbf{x_n}) - \epsilon - \xi_n^*, \tag{2.7}$$

This constraint problem can be transformed into its dual form by introducing Lagrange multipliers $a_n \geq 0, a_n^* \geq 0$. The dual problem involves maximizing

$$L(\mathbf{a}, \mathbf{a}^*) = -\frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}(a_n - a_n^*)(a_m - a_m^*)K(\mathbf{x_n}, \mathbf{x_m}) \tag{2.8}$$

$$-\epsilon\sum_{n=1}^{N}(a_n + a_n^*) + \sum_{n=1}^{N}t_n(a_n - a_n^*)$$

where $K(x_n, x_m)$ is the kernel function, t_n is the target output,

subject to constraints

$$\sum_{n=1}^{N}(a_n - a_n^*) = 0, \tag{2.9}$$
$$0 \leq a_n,\ a_n^* \leq C,\ \ n = 1, ..., l \tag{2.10}$$

2.2. Introduction to PageRank

The PageRank algorithm was originally described in a research paper by Page and Brin[6]. It was first introduced as a way 'to measure the relative importance of web pages'. PageRank is interesting to include in this project not only because it is a defining feature of the Google search engine, but because it is a unique feature of its type, as it depends on the link structure of the whole web as opposed to other web features such as word count and alike.

The basic idea is to capture the link structure of the web and provide a ranking of pages that is robust against manipulation. To achieve this, the importance 'flows' forward: the backlinks share their importance with the pages they link to. Such a simplified ranking of a set of pages can be expressed as an assignment

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{2.11}$$

An intuitive justification for such ranking is by analogy with a citation network, we are likely to find highly cited pages more important. The equation is recursive, so iterating until convergence results in a steady state distribution, which corresponds to the PageRank vector. Together with this intuition the paper introduces the *Random Surfer Model*: we are interested in a steady state distribution of a random walk on the Web graph. At each step the surfer either follows a random link or 'teleports' to a random page. The paper justifies the approach of ignoring the dangling links, as it doesn't have a significant affect on the ranking.

The teleportation vector determines whether the PageRank is personalized. For the non-personalized version the teleportation vector holds equal probabilities for all pages in the Web, whereas in a personalized approach the probabilities are distributed according the knowledge of the surfer's previous activity. In this project we are only concerned with nonpersonalized ranking, which simplifies things a little.

2.3. Requirements Analysis

Non Requirement – usability

2.4. Evaluation Approach

Measuring performance of machine learners accurately is a major challenge primarily because both training and testing are dependent both on the quantity and quality of data used. This fact motivates repeating experiments with a variety of data (**TODO: n fold sampling**). Another difficulty arises from the degrees of freedom that are internal to the learners: the hyperparameters. Especially SVM kernels are vulnerable to mis-evaluation due to the number of parameters which need to be set. The issues of hyperparameter tuning and data sampling

Requirement description	Priority	Difficulty	Risk
Functional Requirements			
Implement a simple search engine	High	Low	Low
Ensure search engine can use both score and rank	Medium	Medium	Low
Implement two different machine learning techniques	High	High	High
Classifiers must achieve better than random results	High	Medium	High
Implement PageRank	Medium	Medium	Low
Non-Functional Requirements			
The search engine must be able to index a few thousand pages in reasonable time[2]	Medium	High	Medium
PageRank computation on a few thousand pages must complete in reasonable time	Medium	High	Medium
Machine learning modules must learn and classify within reasonable time	High	High	High

Table 2.1.: *Project objecives*

will be addressed in the Evaluation chapter later. This section briefly touches upon evaluation models exploited.
Two proposed techniques – Naive Bayes and SVM regression – are quite different, so comparing them is potentially erroneous. However, comparisons can be done within each method, as there is a lot of scope for variety of implementations in each.
As a baseline for the Bayesian approach, a very primitive ranking model will be used. We will simply disregard the scoring function behind the rank and directly infer the ranking function instead. Precision and Recall metric is used to quantify performance. More sophisticated ranking models can then be compared to this basic performance. In particular, weak ranking property can be evaluated.**TODO: explain** Figure 2.1 shows a simplified evaluation framework for Naive Bayes. The Test Data is the data carefully set aside at the beginning that is never exposed to the learner. **TODO: bayes and heuristics?!**

For an SVM learner the baseline can be set to the performance using a linear kernel below.

$$K(x, y) = x \cdot y$$

where '$\cdot$' denotes the dot product. This is expected to be very high for linear heuristics but a lot lower for non-linear ones. Mean Squared Error will be used

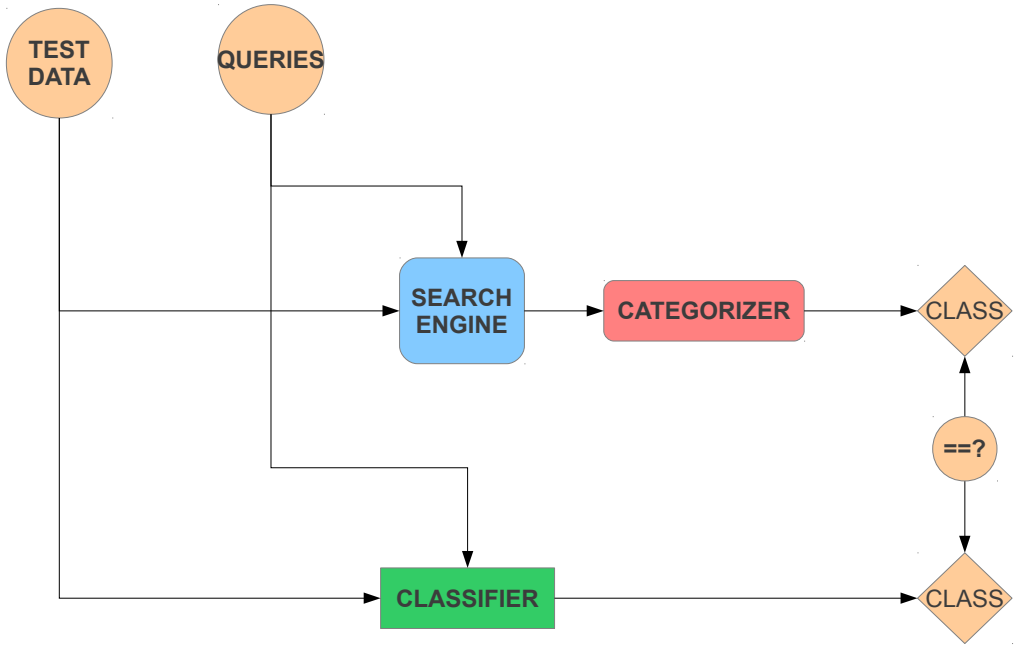


Figure 2.1.: *Evaluation process.*

as the risk function to minimize.

$$MSE = \frac{1}{|Actual|}\sqrt{\sum_{i=1}^{N}(Actual_i - Predicted_i)^2}$$

where $Actual$ is an array of true values of size $|Actual|$ and $Predicted$ is the corresponding array of predictions made by the classifier.

2.5. Choice of Tools

2.5.1. Data Gathering

Web pages used as Training and Test data are not required to have any special properties, but diversity and typicality are seen as advantageous. As for the size of the training data, Domingos [1] suggests that a primitive learner given more data performs better than a more complex one with less data. This, of

course, is under certain assumptions of data quality, namely the assumption that the training data is a representative subset of all the possible data. Intuitively, provided there is no bias in data gathering, more data implies better generality. I have started with a training set spanning an order of a few thousands of pages, however, in practice, I found that there is no particular improvement beyond a thousand pages. **TODO: Link to relevant part or example data here?**

2.5.2. Search Engine

Next important decision regarded the search engine. Originally, I considered using open source existing engines, in particular, Lucene. Even though I could freely modify it for the purposes of the project, the complexity of it was superfluous. I saw writing a simple search engine as a more beneficial exercise, as developing it in the first place potentially gives an insight into the problem.

Functionally our search engine is a black box that takes a set of webpages and a set of queries and outputs an order. The order is determined by the features of the page, which together make up a score. The score is the function we want to infer using the ranking assigned by the search engine, however, we are only given the order as evidence. Machine learning paragraph below will address this issue in more details.

In general there are two aspects of information retrieval that have to be accounted for: precision and recall. Precision is the fraction of retrieved pages that are relevant to the query, whereas recall is the fraction of relevant documents that are retrieved. Even though both are important for a good search engine, but in practice, the web is very large, and so precision, or even *precision at* n[3]) has become more prominent in defining a good search engine: very rarely the user actually browses returns that are not in the top few tens of returned pages. Therefore, modern search engines tend to focus on high precision at the expense of recall [5]. Therefore, we will concentrate primarily on precision, when designing a search engine.

2.5.3. Programming Language

When choosing a programming language, main considerations reduced to library availability and simplicity. The project imposes no special requirements on

[3]'Precision at n' only evaluates precision with respect to n topmost returned pages.

the language, apart from, perhaps, library infrastructure for parsing web pages. Python is simple language with extensive library support. As for efficiency, all the mathematical operations in this project rely on python math libraries, which are implemented in C. I have not programmed in Python before the project, so a slight overhead was caused by having to learn a new language.

2.5.4. Libraries

2.5.5. Development Environment

2.6. Software Engineering Techniques

While the set up of Part II projects encourages waterfall-like development model, this project takes an iterative approach. The first iteration renders a prototype: a primitive search engine with a Naive Bayesian baseline classifier. The next iteration modifies each part of the system towards a more complex solution. Evaluation is performed at each iteration. Within each iteration the development follows the evolved waterfall model – the incremental build model. Each increment represents a functionally separate unit of the system: a search engine, a learner, a parser and an assessment module. Increments are developed sequentially and regression testing is performed separately before integration.

The backup of the code is twofold: every time a substantial change is made a remote version control repository is updated to hold the newest version. Regularly both the code and the data used and obtained during evaluation are also backed up onto an external hard drive.

During the development of the learners a development pool of pages will be used, which must be disjoint with the Training or Test data. This ensures that no optimization is tailored to the data used for evaluation.

2.7. Summary

3. Implementation

This section describes parts of the system that I have implemented. **TODO: overview**

3.1. System Architecture

To achieve the goal of the project, a machine learning techniques comparison framework was necessary. In the Introduction I mentioned the benefit of having a transparent system as an object of learning. To further justify this decision, it is worth mentioning that generalisation using machine learning is different from most optimization problems in that the function that is being optimized is out of our reach, and all that is visible to the machine learner is the training error. Because our goal is not the correct classification of real data, but identifying the means to correct classification, it is important that informed choices are made towards improvement of the learner. Taking this into account, knowing the function that we want to learn and having direct control over it will guide the improvement of the search engine.

This argument motivates a system in three parts: a search engine, a machine learner and a parser to mediate between the two. Figure 3.1 illustrates the proposed learning system. Training data is a set of web pages set aside specifically for training purposes.

3.2. Data

Data gathering is of major importance, as data decisions have tremendous impact on the rest of the implementation and the success of the project. The rest of the system assumes the existence of a pool of web pages available offline for fast indexing and parsing. Ideally the corpus of web pages would be 'representative'

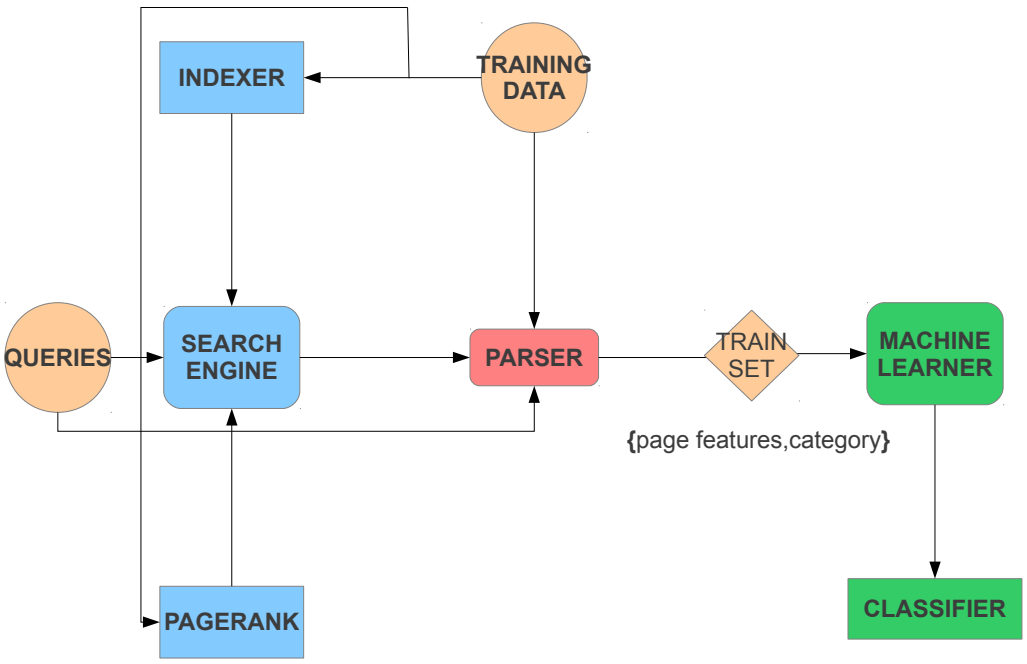


Figure 3.1.: *Overview of the system. Three major parts from left to right are search engine, parser and machine learner.*

zo

of the whole web, to make generalisations more accurate. Originally I was hoping to get such collection from a resource, like, for example the web corpus of the *Text Retrieval Conference* (TREC). However, such data is not easily available, so I had to retrieve pages using *Wget*. To approach the similarity to the web, I restricted the corpus to one semantic area by only downloading websites related to construction materials[1]. This limitation of topic has certain advantages: relatively few pages need to be downloaded before the pages appear sufficiently interlinked, so the corpus has a distinct structure (as web pages similar in the field link to each other).

To conform with machine learning guidelines I have originally separated data for Test and Training. Different seed pages were used, as well as each resides in their own directory. Each directory is estimated at around 3000 pages. In addition to this, I have taken the further precaution of setting aside a small Development corpus to be used only while development to ensure no bias towards training data at the implementation stage. Also, its comparatively small size meant faster testing. All the links are converted, so that the link structure is preserved.

3.3. Search Engine

The first basic block of the system – the search engine – can be logically split into two main parts: an indexer and a sorter. Because we concentrate on precision, as discussed in the Preparation chapter, we will assume for simplicity that all relevant documents are returned. This allowed me to use an existing library implementation of the indexer and focus on the sorter. This crude assumption also distinguishes the project's goals from Informational Retrieval: we can ignore the evaluation of the search engine itself and move away from the specifics of web pages towards the more general problem, so that the nature of features makes little difference and the existence of a hidden function is all that matters.

[1] There is no particular reason to have chosen this particular topic, except it has been used by the external project originator in his experiment trying to infer Google's ranking factors. This is described in the Proposal in more detail.

3.4. Indexer

The requirements on the indexer include flexibility, speed of indexing and retrieval as well as simplicity and usability. The *Whoosh* python library provides all of these, so I used it to build an indexer. Whoosh is an open source indexing library, so I had the option of modifying any part of it. It is built in pure Python, so it has a clean pythonic API. Its primary advantage is fast indexing and retrieval, although we are mostly concerned with retrieval speed, as indexing is done rarely. The predecessors of Whoosh have served as the basis of well-known browsers such as *Lucene*, so it is also a powerful indexing tool, should I have needed more sophistication.

I have defined a very simple schema for indexing. Perhaps, one notable detail is that Whoosh can store timestamps with the index, which enabled me to provide both clean and incremental index methods. The incremental indexing relies on the time stamp stored with the index and compares it to the last-modified time provided by the file system. The user can specify whether indexing has to be done from scratch or updated to accommodate some document changes or document addition/deletion. I haven't originally expected to need an incremental indexing capability, but throughout the project it has permitted for a significant speedup.

Previously, I have defined the sorter as a logical unit that provides ranking to the retrieved documents. For the first prototype, the sorter returned the pages in the order of decreasing PageRank. Subsequently, the sorting has been decoupled from retrieval and incorporated into the Parser module, which is described later in detail.

3.5. PageRank

The PageRank vector is computed using matrix inversion. All matrix operations were performed with the help of the python numerical library 'numpy'. Take t to be the teleportation probability, s=1-t is the probability of following a random link, E is the teleportation probability: equiprobable transitions, as

using non-personalized PageRank (see Preparation), $E_i, j = 1/N$ for all i, j. G is a stochastic matrix holding the link structure of the data, such that

$$G_i, j = \begin{cases} 1/L & \text{there is a link from i to j and L = number of links from i} \\ 0 & \text{there is no link from i to j} \end{cases}$$

Then M is a stochastic matrix representing the web surfer activity, such that M_i, j is the probability of going from page i to page j,

$$M = s * G + t * E \tag{3.1}$$

In one step the new location of the surfer is described by the distribution Mp. We want to find a stationary distribution p, so must have

$$p = M * p \tag{3.2}$$

Substituting 3.1 into 3.2

$$p = (s * G + t * E) * p = s * G * p + t * E * p \tag{3.3}$$

Rearranging equation 3.3 gives

$$p * (I - s * G) = t * E * p \tag{3.4}$$

where I is the identity matrix

We can express E*p as P where P is a vector $[\overbrace{1/N, 1/N, \ldots, 1/N}^{N}].T$,as members of p must sum to one. So computing PageRank amounts to

$$p = t * (I - s * G)^{-1} * P \tag{3.5}$$

where $(I - s * G)^{-1}$ denotes a matrix inverse operation.

This solution is simple at the expense of being slow. Although computing inverse of a matrix is computationally expensive, we don't need to scale beyond a few thousands of pages. To avoid recomputation, I used python object serialization module - Pickle to store the PageRank vector for each directory. The resultant performance was actually very reasonable, the time spent computing PageRank was insignificantly small in comparison to the time spent crawling the directory.

The PageRank vector computation happens within the `Pagerank` class. The whole `Pagerank` object is written to memory using the python object serialization module `Pickle`. A `load` class method is defined on the `PageRank` class to retrieve the relevant object for a given directory. The class is instantiated with an instance of the `Crawler` class, which embodies the link structure of a directory and is described in the next section.

Page	A
A	0
B	1/2
C	1/2
D	0

Table 3.1.: *Illustration of non-dangling pages: B and C share A's 'importance' equally.*

3.6. Crawler

The `Crawler` abstracts away the underlying data directories and computes their link structures as matrices. The matrix G, used for the PageRank computation, represents random link following activity. To obtain such a link structure each page has to be parsed, and all links recorded. Because our data is obtained from a single source page by recursive link following, every page in a directory is guaranteed to be discovered by a spider.

The Crawler class recursively traverses the pages depth first starting with the seed page, the same as the seed page used for recursively downloading the pages from the web. To make sure each page is only explored once, a dictionary is used to hold pairs of absolute path, which uniquely identifies the page, and a numerical value corresponding to the time stamp when the page has been first discovered.

Although every page has a unique path, the links to other pages are relative. Such links need to be normalized to maintain consistency. A page object is used to encapsulate path complexity: all link paths are converted to absolute paths before addition to the dictionary. All outbound links are stored with the page in a Set data structure, such that no link is added more than once.

To produce the stochastic matrix G, we start with an empty NxN matrix, where N is the total number of pages. We assume that whenever a surfer encounters a dangling page – a page that has no outbound links – a teleportation step occurs. Therefore, every dangling page links to every page in the pool including itself with equal probability 1/N. For non-dangling pages, all links are assumed equiprobable and all pages that are not linked to have probability of 0. So if page A links to pages B and C, but not itself or D, its row in G is described by the Table 3.1.

3.7. Naive Bayes

In the course of this project two distinct machine learning algorithms have been used. The design goals for the implementations were as usual: speed and correctness. Another important aspect of the design is that the system and the machine learning modules must be sufficiently decoupled. This is an issue of scalability and code reuse. The implication on the machine learner implementations is that both must communicate with the system via the same interface. The rest of the section describes in detail how the two proposed machine learners were implemented.

In Section **??** it has been shown that Naive Bayesian is a good learner to implement for the first prototype, so a very quick implementation was preferable to make sure the system can potentially function as intended. Due to its simplicity and popularity, Naive Bayesian is widely available in the libraries. Because the implementation of this module is straightforward, not central to the project, I decided to use one of many existing python implementations.

The *nltk* library implementation was particularly appealing as it offers a very concise interface. A classifier object is initialised by the `train` method on the `NaiveBayesClassifier` class. The format of the training set is defined as a list of tuples of featuresets and labels, e.g.
$[(featureset_1, label_1), \cdots, (featureset_N, label_N)]$.

The `train` method simply computes the prior – the probability distribution over labels P(label) and the likelihood – the conditional probability P($featureset = f|label$) by simply counting and recording the relevant instances. The method outputs a `NaiveBayesClassifier` instance parametrized by the two probability distributions. P(features) is not computed explicitly, instead, a normalizing denominator **??** is used.

$$\sum_{l \in labels} (P(l)P(featureset_1|l) \cdots P(featureset_N|l)) \tag{3.6}$$

The `classify` method on the `NaiveBayesClassifier` object takes exactly one featureset and returns a label which maximizes the posterior probability P(label—featureset) over all labels. Previously unseen features are ignored by the classifier, so that to avoid assigning zero probability to everything.

The only tangible difficulty with the implementation was the framework in which classification is done. This task is, however, achieved by the Parser. To batch classify everything in the test directory, the classifier object is passed to the Parser, where the featuresets for the test directories are computed, classified and recorded for the evaluation stage later. The particulars of this process is described in section **??**.

3.8. Support Vector Regression

In the Preparaion chapter we have looked at the theory of support vector machines as found in textbooks. This section builds on the theory described and explains further transformations that were an essential part of the implementation.

To implement a support vector machine one must solve a problem of optimizing a quadratic function ssubject to linear constraints – usually referred to as the *Quadratic Programming* (QP) problem. Therefore, the first implementation task was to convert our existing optimization problem into a generic QP form to make use of the available solvers.

The maximization problem 2.8 can be trivially expressed as a minimization problem (equation 3.7).

$$min_{\boldsymbol{\alpha},\boldsymbol{\alpha^*}} \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha^*})^T P(\boldsymbol{\alpha} - \boldsymbol{\alpha^*}) + \epsilon \sum_{i=1}^{l}(\alpha_i + \alpha_i^*) - \sum_{i=1}^{l} t_i(\alpha_i - \alpha_i^*) \tag{3.7}$$

subject to constraints 3.8 and 3.9 below.

$$\mathbf{e}(\boldsymbol{\alpha} - \boldsymbol{\alpha^*}) = 0 \tag{3.8}$$

$$0 \leq \alpha_i, \alpha_i^* \leq C,\ i = 1, ..., l \tag{3.9}$$

where $\mathbf{e} = [1, ..., 1],\ P_{ij} = K(x_i, x_j),\ t_i$ is the target output, $C > 0$ and $\epsilon > 0$.

At this point in the implementation, for the first time, python did not seem like an ideal choice. *Cvxopt* is one of the few python libraries that implements a QP solver. The specification to the QP function is as follows:

`cvxopt.solvers.qp(P,q,G,h,A,b)` solves a pair of primal and dual convex quadratic programs

$$\min \frac{1}{2}x^T Px + q^T x \tag{3.10}$$

subject to

$$Gx \leq h \tag{3.11}$$
$$Ax = b \tag{3.12}$$

Described in the next few pages are the transformations I devised to reconcile the minimization problem 3.7 and the library specification 3.10 and their respective constraints.
We take x to encode both $\boldsymbol{\alpha}$ and $\boldsymbol{\alpha}^*$ simultaneously, treating the upper half of x as $\boldsymbol{\alpha}$ and the lower half as $\boldsymbol{\alpha}^*$:

$$x = \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\alpha}^* \end{bmatrix}$$

We will see later how this representation allows for elegant representation of the problem (3.7).

First, we express the first term in 3.10 to hold $(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*)^T P(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*)$. Take matrix P in equation 3.10 as

$$P = \begin{bmatrix} K & -K \\ -K & K \end{bmatrix}$$

where $K_{ij} = K(x_i, x_j)$ is the kernel.
Observe that now

$$x^T Px = \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\alpha}^* \end{bmatrix} \begin{bmatrix} K & -K \\ -K & K \end{bmatrix} \begin{bmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\alpha}^* \end{bmatrix}$$

is equivalent to the first term of equation (2.8)

$$\sum_{n=1}^{N} \sum_{m=1}^{N} (a_n - a_n^*)(a_m - a_m^*) K(x_n, x_m)$$

Now expressing the remaining two terms of 3.7 by taking

$$q = \begin{bmatrix} \epsilon * \mathbf{e} - t_0 \\ \vdots \\ \epsilon * \mathbf{e} - t_{N-1} \\ \epsilon * \mathbf{e} + t_0 \\ \vdots \\ \epsilon * \mathbf{e} + t_{N-1} \end{bmatrix}$$

to achieve

$$q^T x = \epsilon \sum_{i=1}^{l} (\alpha_i + \alpha_i^*) - \sum_{i=1}^{l} t_i (\alpha_i - \alpha_i^*)$$

To encode constraint 3.9 consider the following pair of G and h.

$$G = \begin{bmatrix} \overbrace{-1 & 0 & 0}^{\text{N}} & \overbrace{0 & 0 & 0}^{\text{N}} \\ 0 & -1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & \ddots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \ddots & 0 & 0 \\ 0 & \cdots & \cdots & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & \cdots & \cdots & 0 \\ 0 & 0 & \ddots & 0 & \cdots & 0 \\ 0 & \cdots & 0 & \ddots & 0 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \\ a \\ a \\ a \end{matrix}}\right\} \text{2N} \\ \left.\vphantom{\begin{matrix} a \\ a \\ a \\ a \\ a \\ a \end{matrix}}\right\} \text{2N} \end{matrix}$$

$$h = [\overbrace{0 \dots 0}^{2N} | \overbrace{C \dots C}^{2N}]^T$$

The final constraint 3.8 is trivial to adapt by simply taking

$$A = [\overbrace{1, \dots, 1}^{N}, \overbrace{-1, \dots, -1}^{N}]$$

and

$$b = 0$$

Having worked out all the matrices, the rest of the implementation dealt simply with coding the matrices up. The Cvxopt library has its own matrix constructor, however, has generally limited functionality when it comes to matrix operations, so Numpy was used a lot for matrix manipulations. During the first implementation attempt, the solver gave mostly unintelligible error messages, so for the prototype SVM I actually used Matlab. Matlab's `quadprog` function had an identical specification to the Python solver I intended to use. Matrices in Matlab are a lot more straightforward to manipulate, which was also a good reason to first check the correctness of my matrices in Matlab.

3.8.1. SVM Kernel Functions

The kernel functions offer SVMs great flexibility, however, it can be difficult to pick an appropriate one. If the expected pattern is well-defined, kernel selection might be intuitive. For example, a linear kernel is perfect for linear heuristics and the problem is reduced to fitting a hyperplane. Similarly, Radial Basis Kernel picks out hyperspheres and so on. However, when we have no expectation of data, we need a most general kernel.

In this section we will examine a few kernels with various score functions: both intuitive and not to observe how performance of each degrades as we use less Here we are using kernels with the parameters determined as shown in the precious section.

Kernel functions are the central component of support vector machines, and its choice is highly dependent on the application. I have considered a variety of kernel functions and experimented with their combinations to see how the performance differs. Table 3.2 shows the kernel functions I have used. The simplest kernel function is the linear kernel. It is simply a dot product of two vectors. Its generalized version is the polynomial kernel, which takes degree as a parameter. Intuitively, the polynomial kernel is a lot more flexible than linear, but the larger the degree, the less 'smooth' it becomes, so it might overfit the training data. The Radial Basis Function kernel is most popular in Support Vector Machines. We take $||x - y||$ to denote the Euclidean distance of the two feature vectors. The γ parameter is equivalent to $\frac{1}{2\sigma^2}$. The choice of γ

Name	$\boldsymbol{K(\vec{x}, \vec{y})}$
Linear	$ax \cdot y^T + c$
Gaussian	$\exp^{-\gamma\|\|x-y\|\|^2}$
Sigmoid	$\tanh(ax \cdot y^T + c)$
Polynomial	$a(x \cdot y^T)^d$
Weighted Sum	`a*Gaussian(x,y,`γ`)+(1-a)*Linear(x,y)`
Product	`Linear(x,y)*Gaussian(x,y,`γ`)`

Table 3.2.: *Kernel Functions*

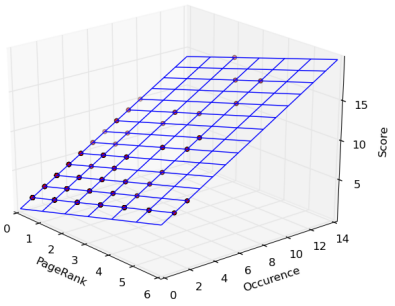
has a major effect on the performance of the kernel and represents a tradeoff between over- and underfitting. To find the best parameter, cross validation is used. **TODO: CROSS VALIDATION** The sigmoid kernel is commonly used in neural networks and in combination with an SVM classifier forms a two layer perceptron network, where the scale parameter a is usually set to 1/N, where N is the number of dimensions (features)[4]. The Sum and Product kernels are both a combination of the linear and RBF kernels. In the Evaluation chapter we will discuss the relative performance of each of the kernels.

3.8.2. SVM Hyperparameter Tuning

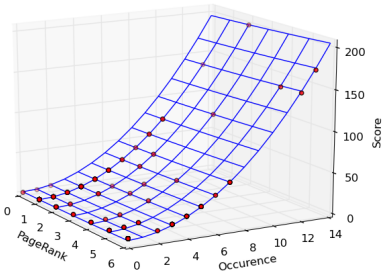
In comparison to Naive Bayes, the SVM has a lot of degrees of freedom. Kernel functions is one such degree of freedom and each kernel has more adjustable parameters. Selecting a kernel itself is discussed in more detail in the next section. This section illustrates how kernel parameters might be tuned.

There are several ways of adjusting parameters of which Grid search is the simplest. In grid search we specify a range of parameters and note down the MSEs for each value (or combination of values). To chose the domain successfully, we will be analyzing at different scales, starting from coarser (wider spread) parameter values, each time only choosing the best for finer tuning. This method does not scale beyond a few variables.

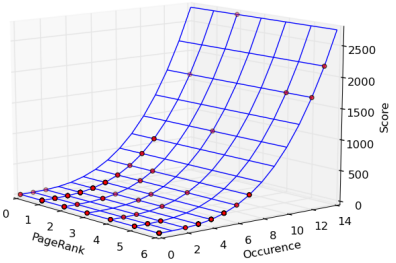
As an example a combined product kernel will be used. The data used is deliberately noisy and non-injective. Given a positive sigma,The Gaussian component varies between one and zero We want to find out how perturbing with Gaussian can make a better fit.



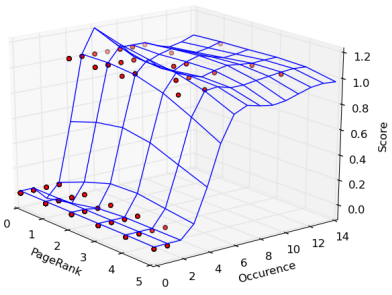
(a) *Linear*



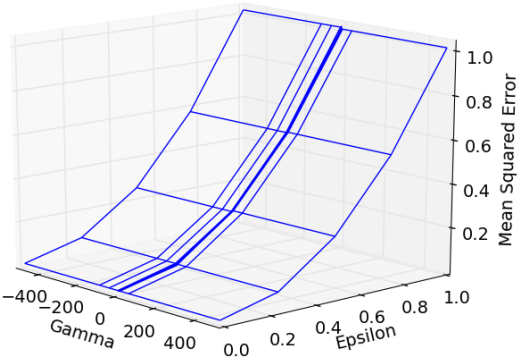
(b) *Quadratic*



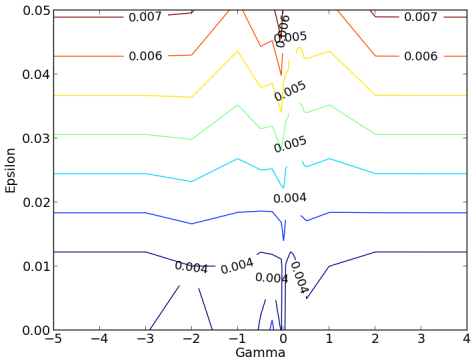
(c) *Cubic*



(d) *Step*



(e) *Increasing epsilon affects the fitting of-test data so greatly, that the changes due to gamma variation are insignificant in comparison.*

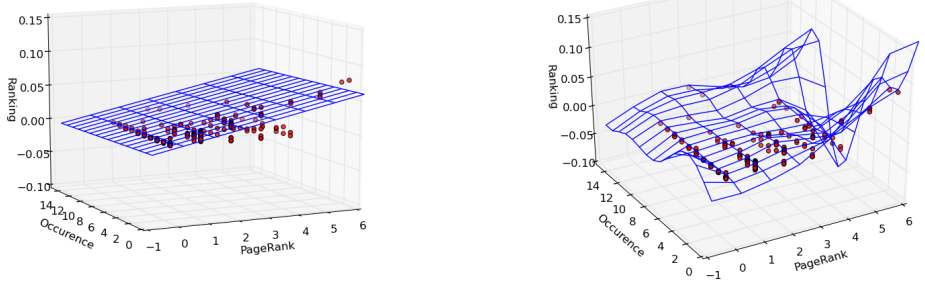


(f) *On a finer scale minimums are visible.*

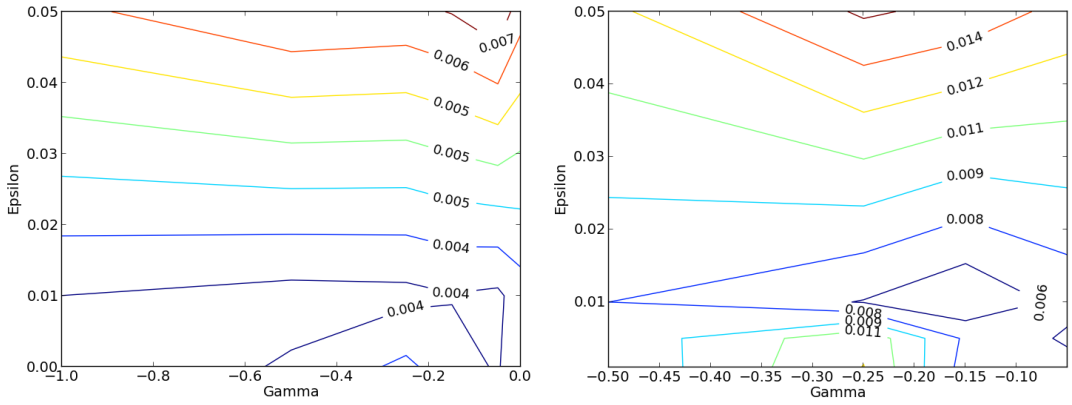
Varying gamma determines how far from linear the fit as well as overfitting.

(g) *Gamma is zero, so the fit is linear*

(h) *Gamma is at its best: the surface slightly perturbed to fit best.*



3.9. Parser



(i) (j)

So far we have looked at the two separate blocks – the search engine and the machine learners. As a functional unit the parser must provide an interface between the two. The search engine simply retrieves pages in response to a query and the machine learner expects as its input a set of labelled features for training and unlabelled features for classification/regression. The Parser, therefore, must hide the nature of the data we are dealing with by translating it into the universal language of machine learners. Hence, the primary function of the Parser is to compute feature vectors for pages. However, it is easy to outsource search engine heuristics into the Parser, too, as they require page parsing.

To accomplish these multiple goals, I have taken an object oriented approach

to the design of the module. What I refer to as the Parser is a few classes, which together perform a series of tasks related to page parsing. The module operates in two modes: rank and score and handles both classification and regression. The high level specification is that we create two objects: a `TestFeatureSetCollection` and a `LabelFeatureSetCollection` objects and they encapsulate all the data, so can be passed around to the machine learners, as well as evaluation and plotting modules. These objects each operate within their own directories, to keep training and testing sufficiently separate.

Both category and rank are treated as page features and hence are part of the `LabeledFeatureSet` class state. The `LabeledFeatureSetCollection` class generates training sets from queries, whereas the `TestFeatureSetCollection` class computes predicted category given an instance of a classifier. Both, however, need to generate feature sets, one for the Training data and the other for the Test data. This common functionality is embodied in their abstract base class, `FeatureSetCollection`. Figure 3.2 shows a UML class diagram illustrating the main structure of the module.

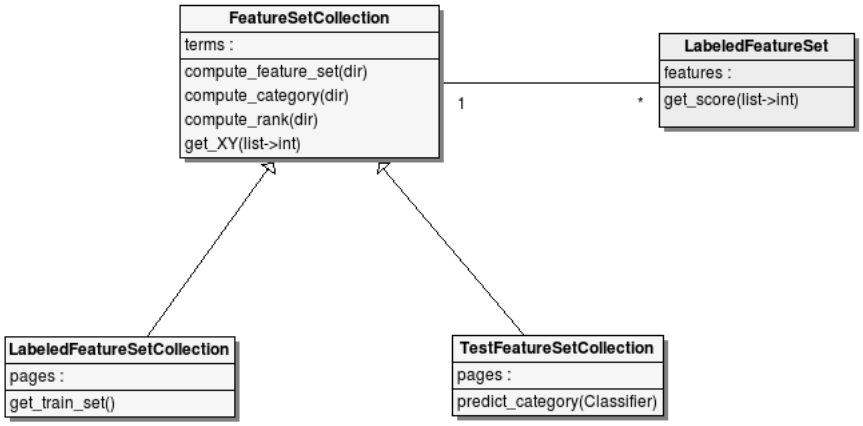


Figure 3.2.: *A UML class diagram describing operation of the Parser.*

The rest of this section talks about the specifics of the implementation, in particular, the features used and how html parsing is done.

Rank mode. Conceptually a `FeatureSetCollection` is a dictionary of `LabeledFeatureSet` objects indexed by both path to page and query term. After initialisation, all the features are computed and 'filled up' per query term per page.

Score mode.

Features. The requirements analysis does not prescribe the use of any particular features. However, it states that both dynamic (query dependent) and static(query independent) features must be used. PageRank is a dynamic feature that has quite special place among others: it takes into account all the pages in the pool and reflects on the structural hierarchy of the web pages. The parser loads the pre-computed PageRank vector indexed by page name to extract the PageRank of any particular page.

An example of a dynamic feature used in the project is query term count: the number of times the query term occurs on the page. This is obtained directly from the html files using the BeautifulSoup library, just like for the link parsing in the crawler. The html is parsed into clean text and the count is computed on the string.

A related but different feature – stem count – is the number of times the stem of the word occurs in the text. This is obtained using the `PorterStemmer` module in Nltk library.

Boolean features are a slightly different variety of feature, so was worth putting in. An example of this is a presence of an image on the page.

The features have been added incrementally as the project progressed. All other features are similar to the ones described above and are obtained using the same methods.**TODO: enumerate features here**

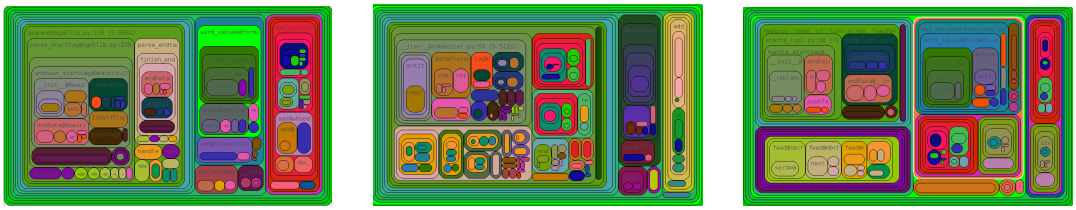
3.10. Optimization

One characteristic peripheral requirement for the system implemented in this project is speed. Even though it is not our direct goal to produce efficient implementations, optimization could not be overlooked, because significant amount

of time is spent processing large quantities of data: indexing, PageRank and feature set computations all must complete in 'feasible' time, i.e. in the final implementation the longest computation takes order of minutes and processes a few thousand pages. Various optimizations have been used to achieve this.

Both PageRank vector and the index are precomputed and kept in persistent storage. Incremental indexing feature allows us to edit parts of the index as opposed to recomputing the whole index from scratch. These precomputations provide certain speedups, but were not enough. Because the system is fairly complex and a lot of library code is used in places, it was hard to determine which code most affects the speed. 'Blind' attempts at optimization did not work well, which motivated the use of a profiling tool.

Profiling the first prototype of the complete system revealed a surprising fact: most time was spent in the library code parsing pages. To mitigate this issue I have tried using custom parsers instead. Apart from speed, robustness was another important consideration, as a failure of a parser increases compute time. Two of the most renowned python parsers are*html5lib* and *lxml*. Figure 3.3 below shows visual representation of time profiling of 3 different runs obtained using the *RunSnakeRun*, each exploiting a different parser. Despite *html5lib* being quoted as the most robust/lenient, *lxml* was sufficiently faster to be preferable.



(a) *Default* (b) *html5lib* (c) *lxml*

Figure 3.3.: *Visual profiles of three parsers from left to right: default python html parser, html5lib and lxml. The internal boxes are sized in proportion to the time spent in each function. In all profiles three distinct major compartments can be seen, the leftmost being the compartment of interest: time spent in parsing. Taking into consideration that the other modules are unaffected by changing the parser implementation, it can be observed that lxml (rightmost) is the fastest and html5lib is the slowest.*

Another useful limitation was discovered due to profiling. The PageRank vector was loaded into memory every time a page was parsed: once for each page. This clearly is undesirable. I used caching to ensure that each of the two possible PageRank vectors (corresponding to the test and train directories) is loaded from

memory exactly once. This is achieved via a double singleton class, which loads the vectors lazily.

4. Evaluation

4.1. Overall Results

The original success criteria for the project stated in the Proposal (see Appendix A) are summarized below:

Criterion 1: *Implemented classifier can identify the importance of PageRank factor in the heuristic.*

The goal has been achieved as described in Section 4.2. Both classifiers were evaluated with PageRank being one of the features. Although PageRank is indistinguishable from any other feature as far as machine learning is concerned, it has been implemented as described in Section 3.5.

Criterion 2: *Various heuristics have been tried with different machine learners.*

This criterion constitutes the core part of the project and the evidence for it can be found in Sections 3.8.1, 3.9 and 4.2.

To achieve the original criteria the following modules have been implemented:

1. Search Engine
2. PageRank vector computation
3. Support Vector Regression and various kernels
4. Naive Bayes Classifier
5. Parser with a data structure supporting various heuristics

The project has been successful in achieving the goals stated in the original proposal. The result of the development is an extensible framework for classifier

evaluation that can be used for further evaluation and analysis of machine learning techniques in general.

4.2. Classifier Evaluation

4.2.1. Methodology

Evaluation and comparison of classifiers is largely unprescribed: there is no method that applies well to every case. A lot of ambiguity is due to the fact that results obtained are ultimately dependent on the data used for training and testing. There are, however, generic guidelines to ensure adequate evaluation and comparison where possible. This section highlights several such aspects, which are then applied throughout the whole chapter.

Classification and Quantization Error

To compare classification and regression effectively, the error incurred in quantizing scores is tracked. In a two-class scenario there is one threshold score value that separates the two classes. This score value is computed as a median to have adequate class representation. When Bayes assigns a class to a feature set, it is perceived as assigning the mean value of the scores present in the class.

Data

To avoid biased results Development corpus was used during development, which is disjoint from the real Test corpus and is of smaller size. The development corpus is further subdivided into the Training corpus and the Validation corpus to mirror the Training and the Test corpora, which are used to obtain all the results presented in this chapter. The evaluation was performed after the development of the whole system was complete and validated on the Development corpus.

Mean Squared Error (MSE)

As a measure of classifier performance I have chosen to use mean squared error. It is computed as

$$\frac{1}{n}\sum_{i=1}^{n}(Actual - Predicted)^2 \tag{4.1}$$

where *Actual* is the true score and *Predicted* is the score assigned by the classifier. MSE is the second moment of the error and hence, incorporates both variance and bias of it. Squaring penalizes large errors heavily, but small errors vanish, therefore MSE is sensitive to outliers as opposed to Mean Absolute Error (MAE). On the other hand, MAE is more sensitive to noise, which we expect when predicting continuous scores. Altogether, MSE provides a good measure that is easily visualised.

Tolerance intervals

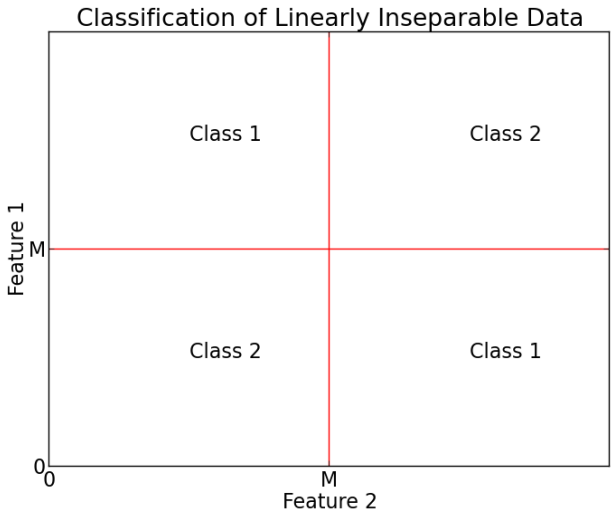
As MSE already captures the variance of the squared error, computing error bars to be the standard deviation of the squared error would offer no new information. In contrast, tolerance interval of the mean error itself is a good visualisation of how spread the error is. In the best case, the interval is tight, indicating the stability of the seen error. If the error is spread, the classifier is making inconsistent mistakes. It is particularly important to include these in our evaluation due to the presence of quantization error. The mean error of the ceiling and its spread is a good indicator of the mean and variance introduced by quantization.

To compute confidence intervals one could repeat the measurement on different data subsets. The number of subsets would have to be around 20 or more for the Central Limit Theorem to apply. In practice, the data is scarce and recomputing the errors is time consuming, so a way to compute the intervals with just one error measurement is desirable. Bootstrapping achieves exactly that by sampling randomly from the array of squared errors to recompute a new mean error. Even though bootstrapping might be less reliable, it provides a simple way to check the stability of the results.

4.2.2. Linear and non-linear classification

Hypothesis: Naive Bayes performs better with linearly separable data.

Naive Bayes is a linear classifier and, therefore, linear separability of data is a premise of successful classification. To test this hypothesis, two minimal two-dimensional examples are constructed as shown in figures 4.1 and 4.2 below. Example 4.1 cannot be solved by fitting a separating line between the classes, so Bayes is expected to fail in this case. Example 4.2 illustrates a 'cut-off sum' function and is trivially separable. Such a linear heuristic should be easy for Bayes to pick up on.

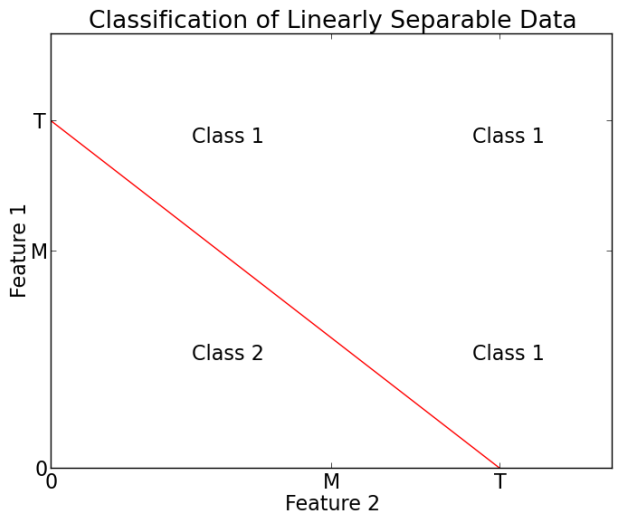



```
def score_inseparable:

    if f1 > M(f1):
        if f2 > M(f2):
            return class2
        else: return class1
    elif f2 > M(f2):
        return class1
    else: return class2
```

Figure 4.1.: *An example of inseparable data. M represents median points for corresponding features. The red lines visualize the boundaries between the classes, separating four distinct quadrants. The M value is chosen to be a median, so that adequate number of training examples was available for all quadrants.*

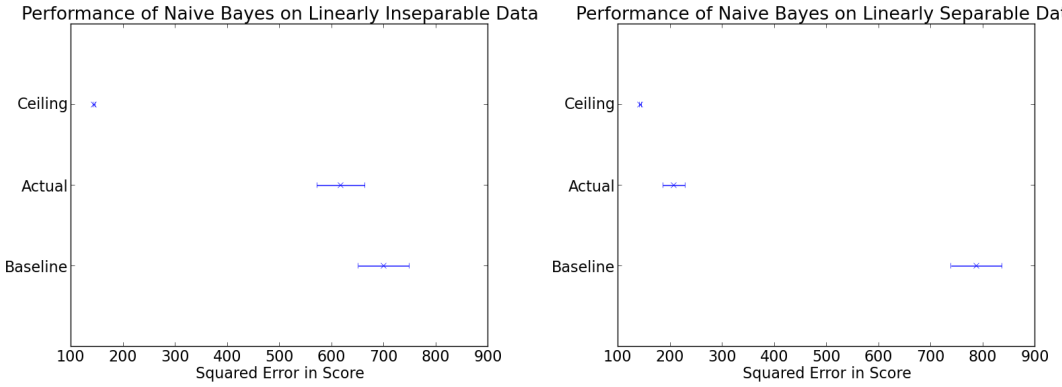
To evaluate the real Bayes behaviour the programs described above are ran with class 1 score equal to 1 and class 2 score equal to 50. Mean squared errors are computed for both cases. Figures 4.3a and 4.3b show the mean squared errors within their corresponding confidence intervals. Along with the actual Bayes classification error (denoted as 'Actual' on the y axes) the Baseline and Ceiling errors are plotted for comparison. The Baseline performance is evaluated by randomly assigning classes, whereas the Ceiling error illustrates the contribution of quantization error, inherent in classification of non-discrete scores.




```
def score_separable:

    if f1 +f2 > T:
        return class1
    else:
        return class2
```

Figure 4.2.: *An example of separable data. On the red line the sum of features is equal to the threshold value T. As before, T is chosen to be the median of the sum f1+f2 to supply equally many training examples on either side of the line.*



(a) *The mean error is near the baseline performance with similarly wide spread.*

(b) *The mean error is distinctly better than the baseline and approaches the goal performance.*

The mean squared error in the inseparable case is significantly larger: in fact, its confidence interval overlaps with the one of the error in random classification (the Baseline error). Similarly, the standard deviation of the error is similar in width to the Baseline. This is due to the fact that the classifier is making mistakes as frequently. In contrast, the separable data case shows a convincing improvement in classification error, however, the classifier is still not perfect. This might be a result of insufficient or unbalanced training data. In this particular

case and in general, adding more training data improves the error up to a point when overfitting occurs. Nonetheless, it is rarely possible to gather a subset of training data that will have enough points very near to the class boundary to allow for perfect classification. Therefore, the result is consistent with the proposed hypothesis.

Hypothesis: SVM with linear kernel performs well for linear heuristics, but becomes unusable for non-linear ones.

For comparison, SVM with a linear kernel is tested for linearly separable and non-separable data. In this particular experiment the number of features used is fixed at 2 and the heuristics used are shown in the table 4.1 below.

Name	Function
Linear	$x + y$
Quadratic	$x^2 + y$
Cubic	$x^3 + y$
Exponential	$exp(x) + y$

Table 4.1.: *Heuristic functions used to evaluate the performance of linear kernel.*

The SVM with Linear Kernel is run with each of the heuristics shown above. The error parameter is tuned as described in Section **??**. The error that produces the best result is used in each case, for example, the error is zero in the case of a linear heuristic. The baseline performance is a hyperplane fitted through the mean score. The ceiling performance is computed as a least squares solution of an equation which has the form of the particular heuristic used. *Numpy* provides a *lstsq* function which finds the coefficients of a particular heuristic such that the squared error is minimimal.

Figure **??** shows a failure of the linear kernel to fit non-linear data. The ceiling error due to numerical error in computation: this explains why the ceiling error in the exponential case is the largest of all.

Clearly, as in the Bayes case, the degradation is rapid. In the cubic and exponential cases the SVM seems to perform worse than the baseline, which can be explained by overfitting.

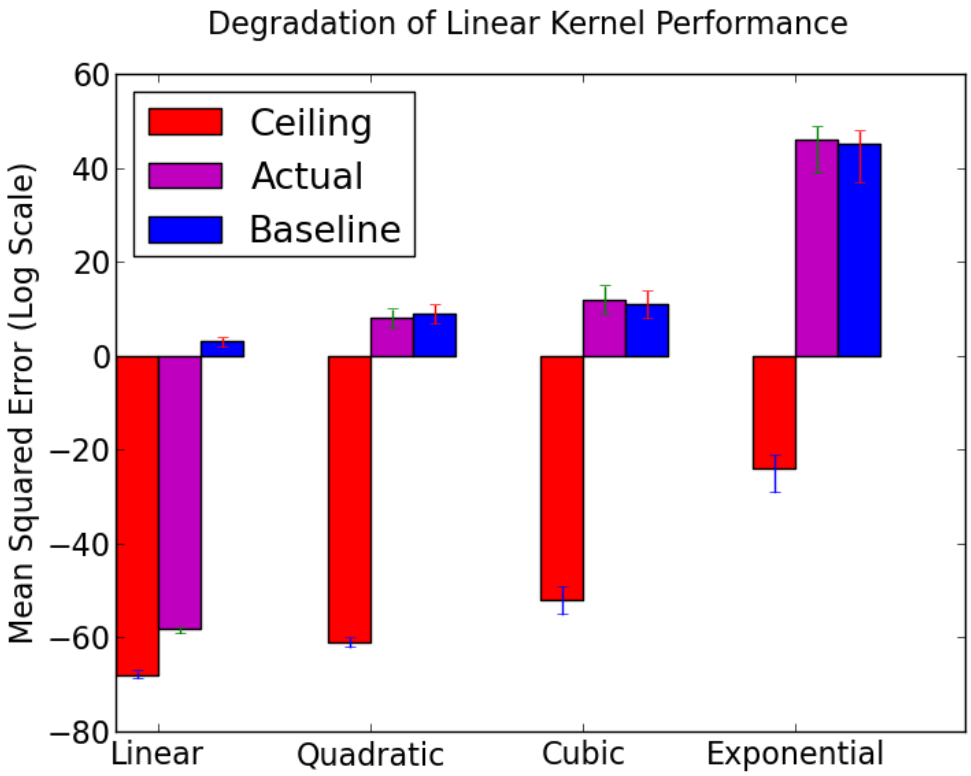


Figure 4.3.: *SVM with a linear kernel is evaluated against four different heuristics (x axis). Note than the MSE is in log scale. The error in the linear case is negligibly small, but is similar to the baseline case for all other heuristics.*

4.2.3. Bayes: Effects of Quantization

Hypothesis: Bayes classifies better when fewer classes are present.

It is intuitive that at a large number of classes, the more classes we introduce, the more precise the classifier has to be to perform equally well. However, taking into account the quantization error, non-linear behaviour might be expected: performance will improve due to the quantization error decreasing faster than precision up to a certain point. This experiment aims to determine the number of classes that minimizes the overall error of the classifier. In some sense, quantization is akin to a hyperparameter: the results are specific to the heuristic and the data used.

In this experiment, a linear heuristic is used with two features as before. The number of classes is varied starting with two classes. Classes are added until the

maximum possible number is reached – the number of distinct scores.

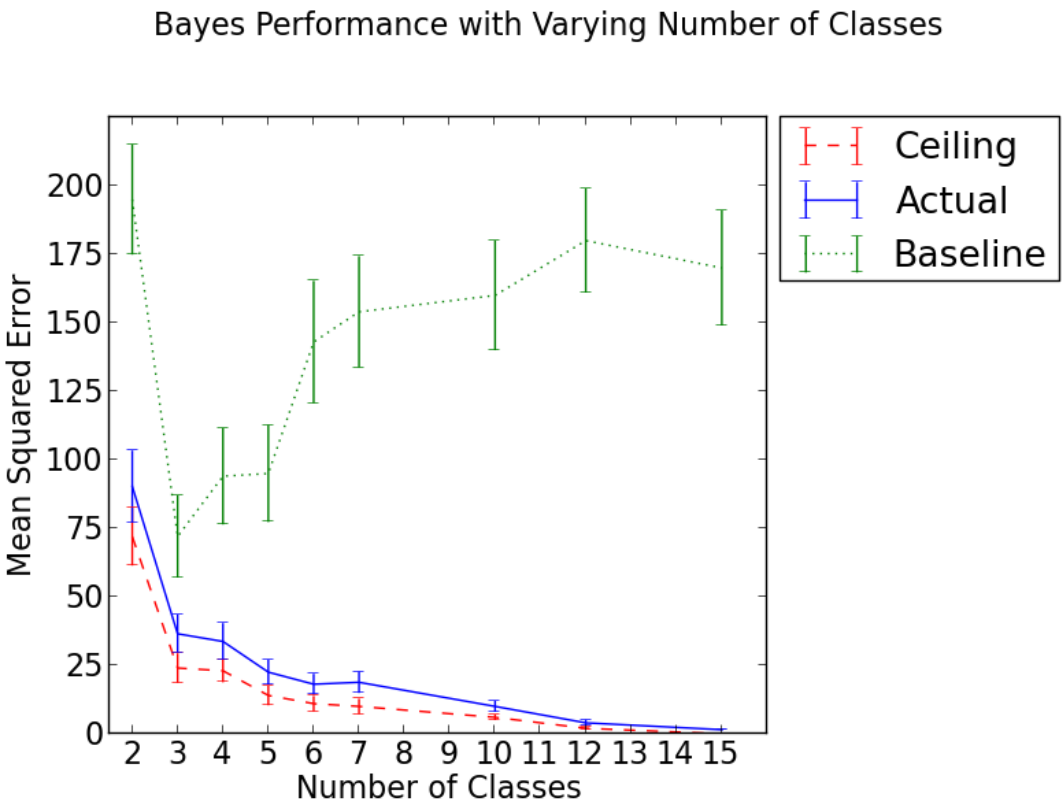
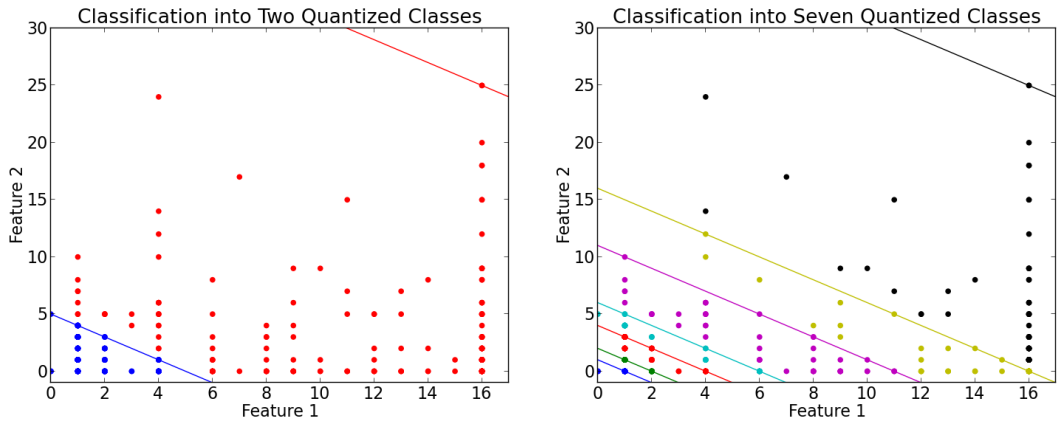


Figure 4.4.: *Error response to varying number of classes.*

Figure 4.4 shows a result inconsistent with the proposed hypothesis: the error is diminishing as rapidly as the quantization error (ceiling). It is visible from figure 4.5 that the data remains separable (as expected) independently of the number of classes. Clearly, Bayes is very effective at classification with the given heuristic. It is worth noting, that the actual performance is still worse than the ceiling and does not converge to zero as opposed to quantization error. It is possible that Bayes consistently misclassified a few particular examples, which lead to a gap in the plot.

The example chosen does not verify the hypothesis and proves the intuition wrong: the Bayes erformance depends solely on whether the data is separable or not.



(a) *Two classes* (b) *Seven classes*

Figure 4.5.: *Classification with different number of classes. The data is consistently separable.*

4.2.4. Curse of Dimensionality

Hypothesis: Bayes performance degrades rapidly as the number of features grows.

So far I have explored two dimensional cases where only two features were used for convenience of visualization. Dimensionality curse is one of the well-known issues with classification: when new features are added, the size of the training set has to grow exponentially to cover all the new dimensions. Naive Bayes, however, lessens this problem a little due to the assumption of conditional independence. It is interesting to see how rapidly its performance degrades when more features are added while the training data is unchanged. It is important that all the features used are conditionally independent, as otherwise, the results would not be valid.

The number of classes is fixed at a value of 7, at which the performance is reasonable but still has some quantization error. This is done to prevent the effects of the extremes of performance. We are only interested in relative performance, so it should not matter how many classes are used, however, it is important that the number of classes used produces a significantly better result than the baseline, to make sure a random guess could not produce a good result.

The result for a few linearly separable functions is computed and the average errors are plotted in figure 4.6. The actual classification diverges from the ceiling

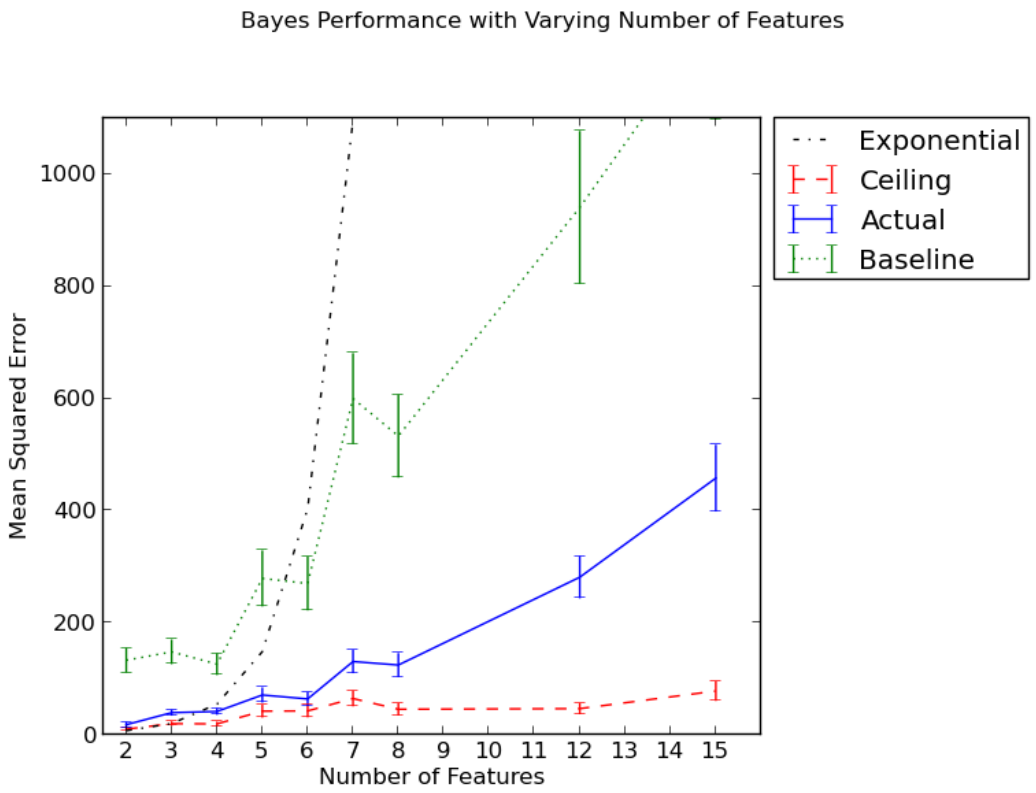


Figure 4.6.: *Error response with varying number of features.*

when the number of features is about 6. The rapid growth of the random classification curve is explained by the curse of dimensionality: degrees of freedom grow exponentially. The exponential growth is plotted in black for comparison: initially the baseline growth is near exponential, but less steep subsequently. This is due to the quality of training and test data: as the number of features grows, the number of training examples for each class is reduced, influencing the overall performance.

The ceiling and the actual curves are growing almost monotonously, as is expected. The flat regions at 5-6 and 7-8 are explained by the data specifics: the particular features added at those point appeared insignificant in the scores as they occurred infrequently and had small values.

It has proven hard to pick many conditionally independent features. Among the ones used were pagerank, image count, word count, frequency of search term occurrence, quality of html, price information availability and alike.

The hypothesis is verified: the error grows as the number of features is increased.

The growth, however, depends also on the quality of features – how much the addition of a new feature alters the classification, as well as on the training set – how many training examples are supplied.

Hypothesis: SVM performance does not degrade as the number of features grows.

As before, linear kernel is used in combination with a linear heuristic. Figure 4.7 supports the hypothesis: the error looks almost constant even in logarithmic scale. Hyperplane fitting is more resilient to more dimensions in the presence of linear heuristics.

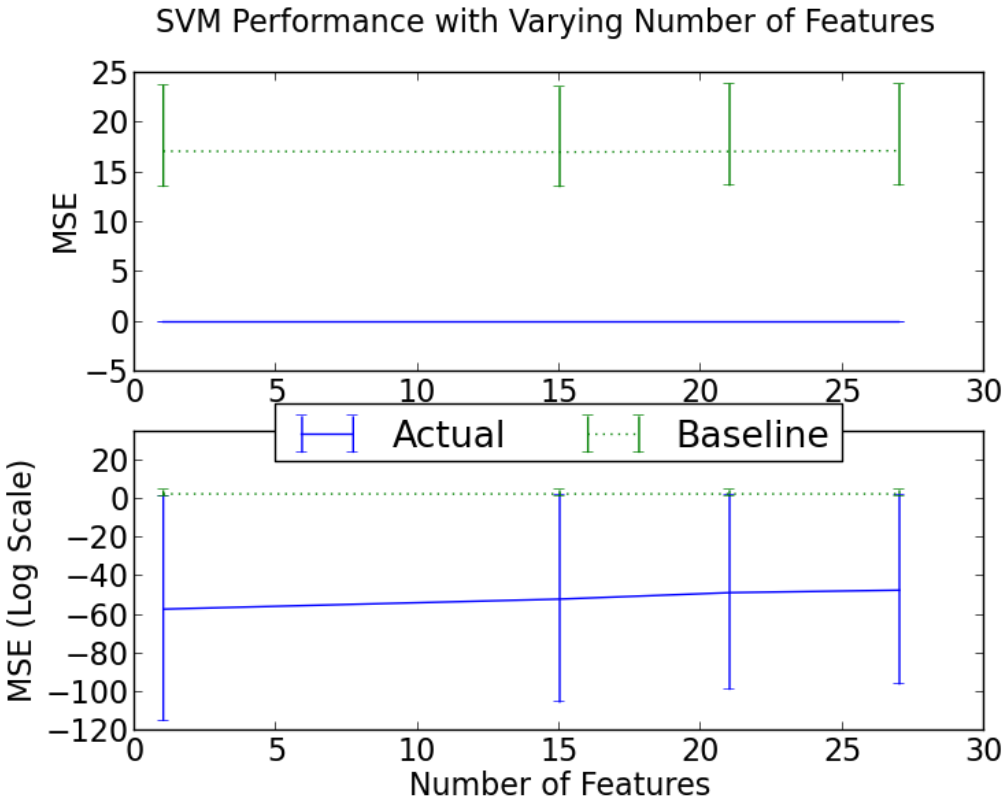


Figure 4.7.: *Both normal (upper figure) and logarithmic (lower figure) scales are shown. The error grows too slow for the growth to be noticeable without the logarithmic scale.*

4.3. Performance Evaluation

As stated in the Preparation, the project does not have any hard timing requirements. PageRank computation, indexing and learning were envisaged as most time consuming. In the end, the time taken in learning and classifying was negligible and same can be said about parsing. Throughout the implementation, it has become obvious that the most time was spent in indexing and PageRank computation. Therefore, in this section, I am looking in more detail in the timings and scalability of the two modules.

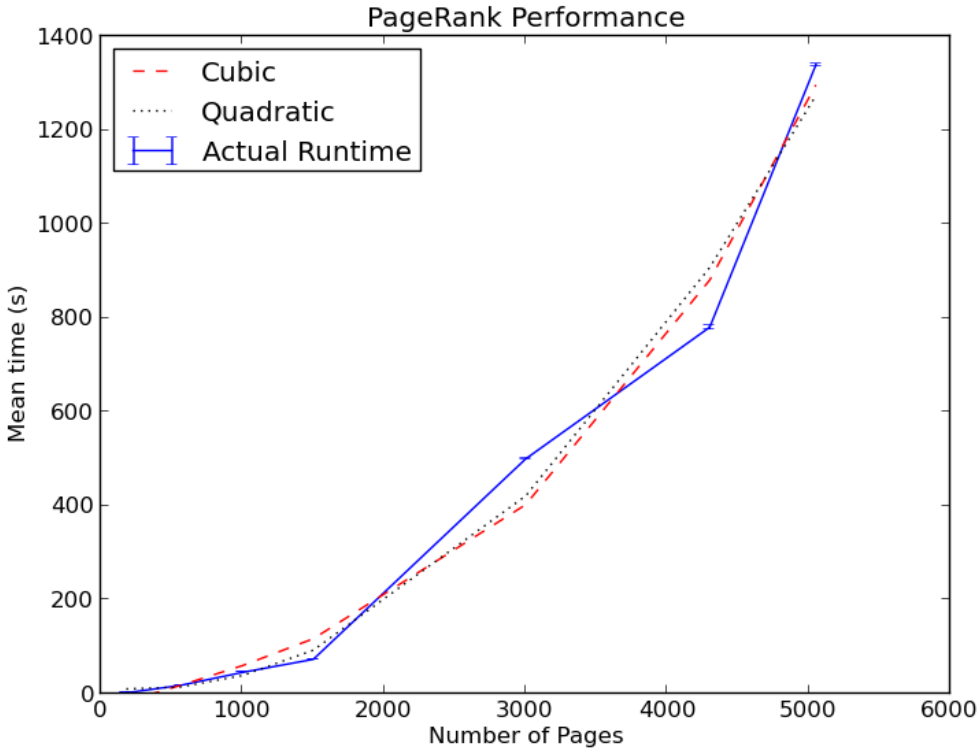


Figure 4.8.: *Time taken in computing PageRank for a growing set of pages. For convenience of comparison, a cubic and a quadratic curves are fitted.*

The plots were generated using a benchmark suite, which makes repeated measurements and calculates means and confidence intervals. I have configured the suite to take a hundred repeat measurements for a range of page numbers. The 95% confidence intervals are represented by the error bars on the plots.

The benchmarks were performed on my personal computer with the following specifications:

Memory	1.7 GB
Cache Size	3072 KB
CPU	Intel(R) Core(TM)2 Duo CPU
CPU Clock Frequency	2.26 GHz

Both plots also display fitted cubic and quadratic curves. The curves are fitted using the *Scipy* library curve fitting function, which uses Levenburg-Marquardt algorithm[1]. Figure

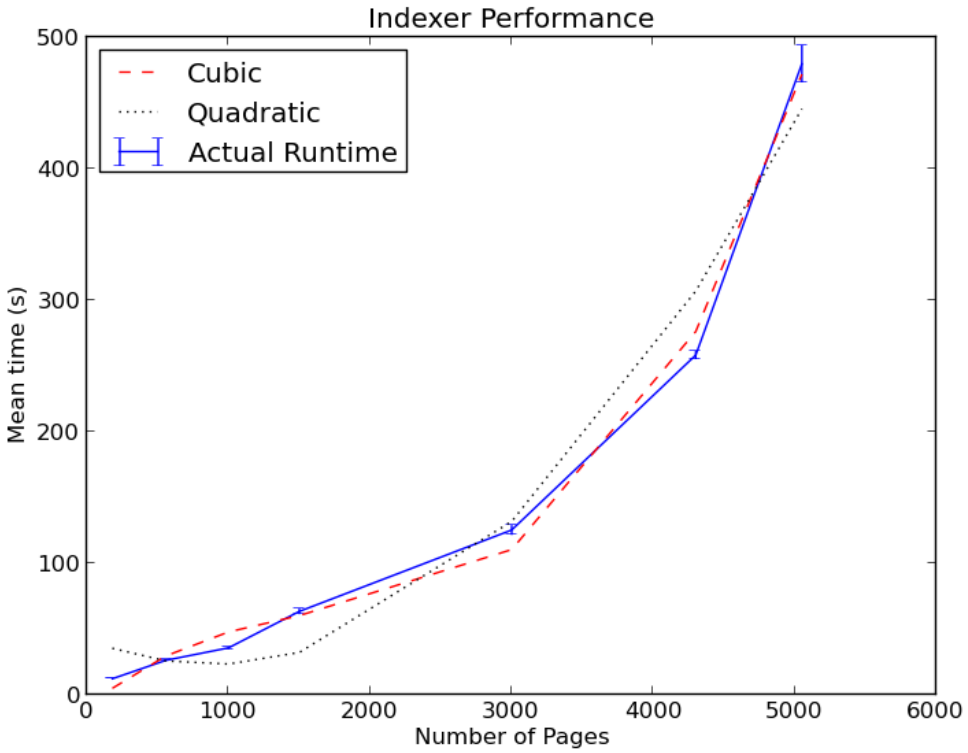


Figure 4.9.: *Time taken in indexing for a growing set of pages. For convenience of comparison, a cubic and a quadratic curves are fitted.*

4.8 shows how the performance of the PageRank computation degrades with the number of pages. Although neither cubic nor quadratic curve fits well, it can be said that up to around 5000 pages the growth is bound by $O(n^2)$. The average number of pages used in the project was around 3000 for each directory, which means it took around seven minutes to crawl and generate the PageRank vec-

[1]Also known as Damped Least Squares method, the Levenburg-Marquardt algorithm works by minimizing a function over the space of its parameters.

tor. However, it is expected that the growth is more rapid beyond the limit of 5000 pages. If dealing with more pages was within the scope of the project, the PageRank computation would have to be optimized further, as matrix inversion does not scale for a problem bigger than a few thousand pages.

Indexing shows similar behaviour, although it seems a lot smoother and unambiguously exhibits cubic behaviour. Indexing 300 pages takes around two minutes, which is satisfactory considering that speed was not the primary objective of the project.

Altogether, both modules could potentially become a bottleneck. However, as development was carried out on a comparatively small development corpus (only around 500 pages), and the verified program was then run on the actual data only once, which means the time loss was infrequent. Both the index and the PageRank vector were stored and reused to avoid expensive recomputation. All things considered, it was not worth further optimizing the modules in question, as speed of implementation was prioritized over the speed of computation due to its apparent infrequency.

4.4. Testing

Formal correctness proofs are neither within the aims of the project, nor would they be feasible in the time given. Therefore, the main purpose of the testing is to gain a degree of certainty in the correctness of the program. The system developed is in a sense a prototype: it only gets used one time in order to obtain certain results. Therefore, tests do not need to be run as often as for a system that would be used in production. In addition, human analysis was often required to assess the quality of the programs. These aspects of the system motivate the decision to use manual testing, which is described further in this section.

4.4.1. High Level Test Plan

Table 4.2 summarizes the test plan used for manual testing. Each of these tests was conducted at the end of module development and whenever a substantial change occurred that could potentially change the behaviour in the cases described.

Module	Test Objectives
Crawler	The output matrix accurately reflects the link structure of the pages
PageRank	The PageRank vector accurately reflects the hierarchy of the web pages
Indexer	Clean indexing overwrites existent index
Indexer	Incremental indexing adds/removes relevant index changes to reflect the changed web pages
Parser	Parser is robust in the face of bad formatting and encodings
Naive Bayes	Classification is better than random for separable data
SVM	The hyperplanes fit the data and change with the data to provide better fitting

Table 4.2.: *Summary of Test Objectives*

4.4.2. Example Test Cases

Crawler and PageRank

Both Crawler and PageRank were tested on artificially engineered small examples of link structure to verify the test objectives have been achieved.

Indexer

Indexing is heavily reliant on the library, so only the implemented parts required extensive testing – incremental and clean indexing capabilities. These were tested on the verification corpus. To test incremental indexing, pages were added, removed and altered and the relevant queries executed to verify that the changes have taken effect. The clean index was build both without an existent index – in which case a new index was created – and with an existent index – to verify that the old index is replaced by the new one.

Parser

The objective of the Parser testing was mainly its resilience to failure: examples of malformatted html and pages with special characters in the address were used to verify the robustness.

SVM and Bayes

Machine Learning modules both were tested with the visual aid of plotting in the two- and three-dimensional cases. The data was plotted alongside the hyperplane (SVM) or the separating line (Naive Bayes). The data was altered to provoke change in the hyperplane/separator and the subsequent change was verified.

4.5. Summary

5. Conclusions

5.1. Lessons Learnt

5.2. Future Work

Bibliography

[1] Pedro Domingos. A few useful things to know about machine learning. University of Washington.

[2] Pedro Domingos. On the optimality of the simple bayesian classifier under zero-one loss. Kluwer Academic Publishers, 1997.

[3] Pedro Domingos and Michael Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. pages 105–112, 1996.

[4] Hsuan-Tien Lin and Chih-Jen Lin. A study on sigmoid kernels for svm and the training of non-psd kernels by smo-type methods. National Taiwan University.

[5] Larry Page and Sergey Brin. The anatomy of a large-scale hypertextual web search engine. Stanford University, 1998.

[6] Larry Page and Sergey Brin. The pagerank citation ranking: Bringing order to the web. 1998.

[7] Jen-Wei Kuo Pu-Jen Cheng, Hsin-Min Wang. Learning to rank from bayesian decision inference. National Taiwan University.

[8] Harry Zhang and Jiang Su. Naive bayesian classifiers for ranking. pages 501–512, 2004.

A. Project Proposal