Karina Palyutina

Machine learning inference of search engine heuristics

Part II Project

St Catharine's College

March 2, 2013

Proforma

Name:	**Karina Palyutina**
College:	**St Catharine’s College**
Project Title:	**Machine learning inference of search engine heuristic**
Examination:	**Part II Project**
Word Count:	[1]
Project Originator:	Dr Jon Crowcroft
Supervisor:	Dr Jon Crowcroft

[1] This word count was computed by `detex diss.tex | tr -cd ’0-9A-Za-z \n’ | wc -w`

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

Introduction

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 [2]. 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.

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/underfitting[**?**]. 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

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

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.

overview of the chapters here.

Chapter 1

Preparation

This chapter describes work that has been done before coding was started. In particular, it focuses on the reasoning behind the design of the system to be implemented. The first section is devoted to research undertaken to determine what can be done and how best to do it. The second section formulates the system requirements, namely formalizes everything that is developed in this project. The last section outlines the particulars of the software engineering approach to be adopted by this project.

1.1 Formulating the Goals

A particular difficulty in this project has been in planning what has to be done. Due to the exploratory nature of the project the course of action had to be predominantly determined by the outcome of a current tactic. Moreover, the unknowns originating from the machine learning further complicated matters.

System Overview. 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.

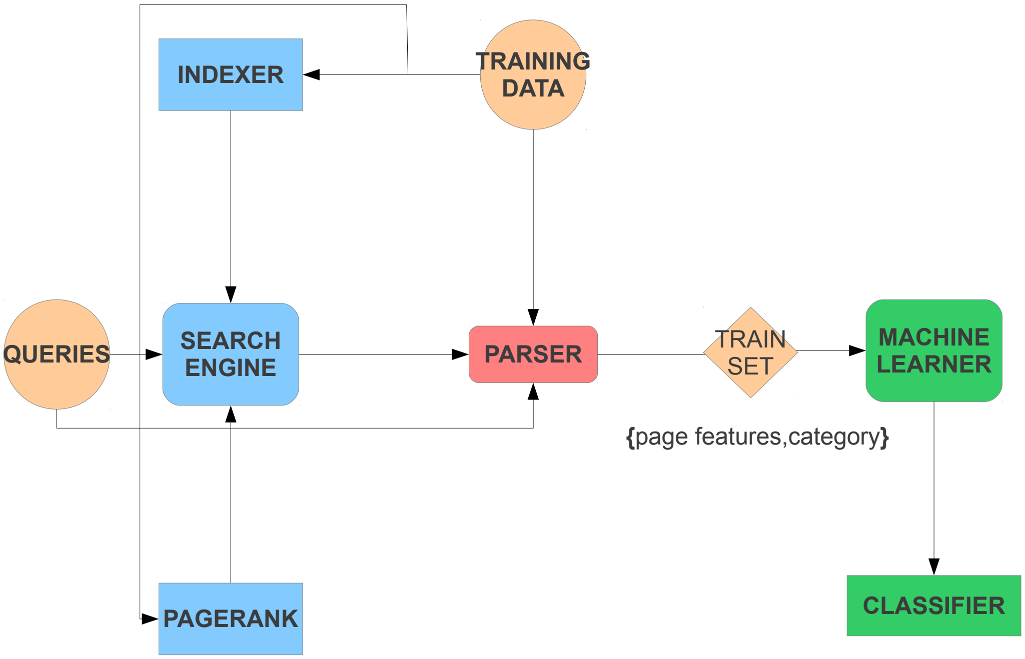


Figure 1.1: Overview of the system. Three major parts from left to right are search engine, parser and machine learner.

This argument motivates a system in three parts: a search engine, a machine learner and a parser to mediate between the two. Figure 1.1 illustrates the proposed learning system. Training data is a set of web pages set aside specifically for training purposes.

Data. 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 [**?**] 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. Link to relevant part or example data here?

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[1]) 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 [**?**]. Therefore, we will concentrate primarily on precision, when designing a search engine.

Pagerank The PageRank algorithm was implemented as described in the original paper by Page and Brin[**?**].

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

Machine Learning. I have now covered main peripheral decisions, but it is machine learning that constitutes the central part of the project. The field was

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

completely new to me to start with, so research of different techniques was a big part of the preparation.

It is generally recommended that the simplest learners are tried first[?]. 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.

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 $P(\vec{F})$:

$$P(C|\vec{F}) = \frac{P(C)P(\vec{F}|C)}{P(\vec{F})} \tag{1.1}$$

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:

$$\hat{C} = argmax_C P(C) \prod_{i=1}^{n} P(F_i|C) \tag{1.2}$$

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[?]. 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 [?]. However, despite poor probability estimates, there exist several frameworks, which make use of Bayesian classification and achieve decent performance in ranking. For example, Zhang [?] 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

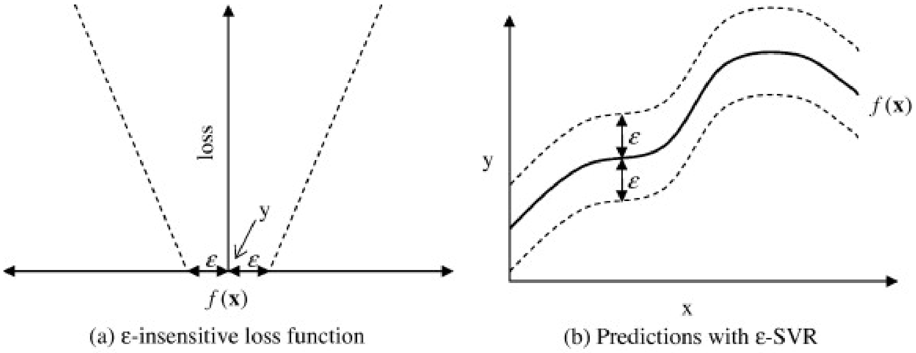


Figure 1.2: TODO: plot these yourself!

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[2]. Another framework for ranking [?] 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 this project. Classification is frequently opposed to regression, so another approach covered by this project is Support Vector Regression. In particular, ϵ-Support Vector Regression.

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 training sequence is 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.

In simple linear regression the aim is to minimize a regularized error function. We will be using an ϵ-insensitive error function(see Figure 1.2 (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 1.2(b).

[2] 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 [?].

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:

$$E = C \sum_{n=1}^{N} (\xi_n + \xi_n^*) + \frac{1}{2} \|\mathbf{w}\|^2 \tag{1.3}$$

E must be minimized subject to four constraints:

$$\begin{align} \xi_n &\geq 0, \tag{1.4} \\ \xi_n^* &\geq 0, \tag{1.5} \\ t_n &\leq y(\mathbf{x_n}) + \epsilon + \xi_n, \tag{1.6} \\ t_n &\geq y(\mathbf{x_n}) - \epsilon - \xi_n^*, \tag{1.7} \end{align}$$

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

$$L(\mathbf{a}, \mathbf{a}^*) = -\frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} (a_n - a_n^*)(a_m - a_m^*) K(\mathbf{x_n}, \mathbf{x_m}) \tag{1.8}$$

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

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

$$\begin{align} \sum_{n=1}^{N} (a_n - a_n^*) &= 0, \tag{1.9} \\ 0 \leq a_n,\ a_n^* &\leq C,\ \ n = 1, ..., l \tag{1.10} \end{align}$$

Evaluation methodology. 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

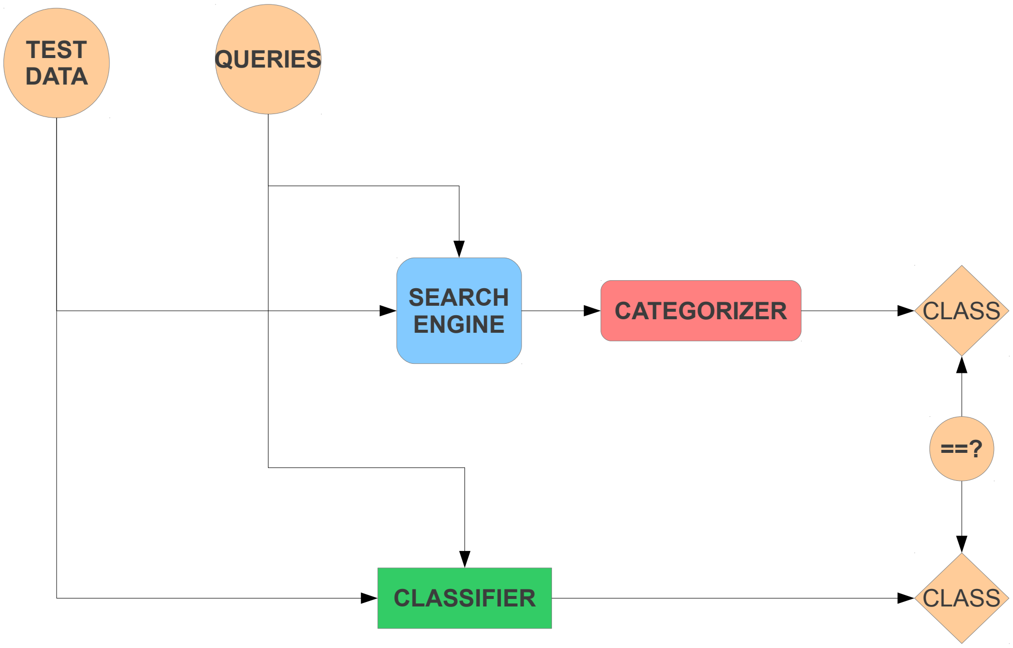


Figure 1.3: Evaluation system.

function behind the rank and directly infer the ranking function instead. More sophisticated ranking models can then be compared to this basic performance.

Figure 1.3 shows an evaluation framework for Naive Bayes. The Test Data is the data carefully set aside at the beginning that is never exposed to the learner.

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

$$K(x, y) = x.y \tag{1.11}$$

where '.' is the dot product.

This is expected to be very high for linear heuristics but a lot lower for non-linear ones.

1.2 Requirements Analysis

The resulting system must comply with the requirements stated below.

- A search engine, which given a query must return a relevant subset of pages within 10 seconds in an order corresponding to a given heuristic, which can also be supplied to the search engine.

- Search engine must base ranking decisions on both dynamic (query dependent) and static (query independent) page features.

- A machine learner must be sufficiently isolated from the search engine through an encapsulating interface to avoid the possibility of undesirable interaction.

- Machine learners must be able to process thousands of pages in a reasonable time.

- The evaluation module must write results to persistent storage.

1.3 Development Strategy

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.

Chapter 2

Implementation

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

2.1 Data

Development, Test and Training data for this project was implemented Separate directories..

2.2 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 an use existing library implementation of the indexer and focus on the sorter.

Indexer The requirements on the indexer included flexibility, speed of indexing and retrieval, 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 timestamp 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.

Sorter 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 is described in more detail in the Parser section.

PageRank 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,

$$M = s * G + t * E \tag{2.1}$$

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

$$p = M * p \tag{2.2}$$

Substituting 2.1 into 2.2

$$p = (s * G + t * E) * p = s * G * p + t * E * p \tag{2.3}$$

Rearranging equation 2.3 gives

$$p * (I - s * G) = t * E * p \tag{2.4}$$

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

$$p = t * (I - s * G)^{-1} * P \tag{2.5}$$

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.

Load class method byname array

Crawler 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 a Wget spider, 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 timestamp when the page has been first discovered. link conversion and parsing 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 datastructure, 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 2.1.

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

Table 2.1: Illustration of non-dangling pages: B and C share A's 'importance' equally.

2.3 Parser

So far we have looked at the search engine. As a functional unit the search engine only retrieves the relevant pages. The Parser is an abstraction for a few classes which together perform a series of tasks involving page parsing. Search engine heuristics are also handled by the parsing module.

The primary function of the parser is to compute feature vectors for pages. At this point we abstract away from the web pages and html parsing: as far as the machine learner is concerned, a feature vector is a complete representation of a page. The infrastructure for feature vector computation is easily extensible to accommodate other related tasks, namely, training set and test set generation. To allow for multifunctionality, I have taken an object oriented approach to the design of the module.

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 2.1 shows a UML class diagram illustrating the main structure of the module.

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

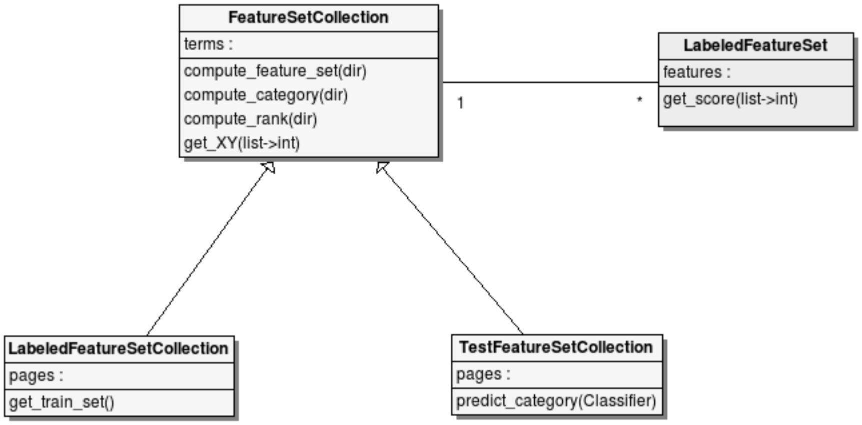


Figure 2.1: A UML class diagram describing operation of the Parser.

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 Natural Language Tool Kit (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.enumerate features here

2.4 Optimization

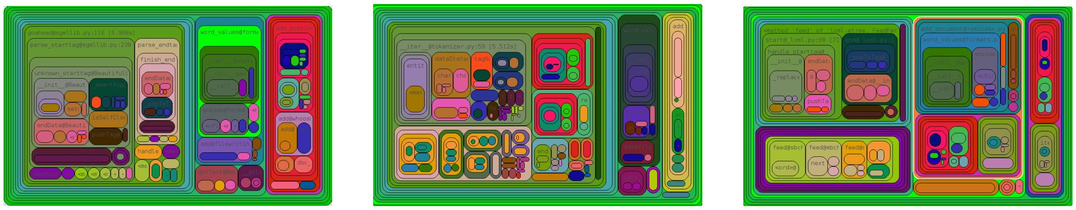
One characteristic peripheral requirement for the system implemented in this project is speed. Even though it is not our direct goal to produce efficient implementations, optimization could not be overlooked, because significant amount of time is spent processing large quantities of data: indexing, pagerank 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 2.2 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.

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.



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

Figure 2.2: 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.

2.5 Machine Learning

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 machine learners have been implemented.

2.5.1 Naive Bayes

In Section 1.1 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.

$$\sum_{l \in labels} (P(l)P(featureset_1|l) \cdots P(featureset_N|l)) \tag{2.6}$$

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.

To batch classify everything in the test directory, the classifier object is passed to the constructor of the texttttTestFeatureSetCollection class. Here the featuresets for the test directories are computed, classified and recorded for the evaluation stage.

2.5.2 Support Vector Machine

The maximization problem (1.8) can be expressed as a minimization problem

$$min_{\boldsymbol{\alpha},\boldsymbol{\alpha}^*} \frac{1}{2}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*)^T P(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*) + \epsilon \sum_{i=1}^{l} (\alpha_i + \alpha_i^*) - \sum_{i=1}^{l} t_i(\alpha_i - \alpha_i^*) \tag{2.7}$$

subject to

$$\mathbf{e}(\boldsymbol{\alpha} - \boldsymbol{\alpha}^*) = 0 \tag{2.8}$$

$$0 \leq \alpha_i, \alpha_i^* \leq C, \ i = 1, ..., l \tag{2.9}$$

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

Solving a constrained optimization problem requires a quadratic programming solver. Cvxopt is one of the few python libraries that implements a QP solver. The specification to the QP function is given as follows:

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

$$\min \frac{1}{2} x^T P x + q^T x \tag{2.10}$$

subject to

$$Gx \leq h \tag{2.11}$$

$$Ax = b \tag{2.12}$$

The goal was, therefore, to express (9) in terms of (12) and their respective constraints. Described below is the transformation I devised to solve equation (9) using existing implementation that requires (12). 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 $\boldsymbol{\alpha}^*$:

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

Take P in equation (12) as

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

where $K_{ij} = K(x_i, x_j)$.

Observe that now $x^T P x$ is the same as $\sum_{n=1}^{N} \sum_{m=1}^{N} (a_n - a_n^*)(a_m - a_m^*) K(x_n, x_m)$ in (6) and

Take q to hold the rest of (9).

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

Now to encode constraint (11) consider the following pair of G and h

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

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

The final constraint (10) is trivial to addapt to (14) by taking $A = [\overbrace{1, \dots, 1}^{N}, \overbrace{-1, \dots, -1}^{N}]$ and $b = 0$.

Chapter 3

Evaluation

Chapter 4

Conclusion

Appendix A

Project Proposal