

Project Report

Extraction of Questions from the Internet using a Machine Learning approach

*Submitted in partial fulfillment of
the requirements for the award of the degree of*

*Bachelor of Technology
in
Computer Science and Engineering*

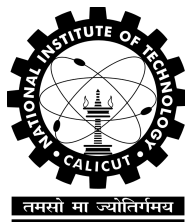
by

Roll No

Name

B090924CS	Alok Saw
B090437CS	Jerrin Shaji George
B090904CS	Shubhangam Agrawal
B090006CS	Stein Astor Fernandez

Under the guidance of
Dr. Priya Chandran



Department of Computer Science & Engineering
National Institute of Technology, Calicut
Kerala - 673601

Department of Computer Science & Engineering

NATIONAL INSTITUTE OF TECHNOLOGY CALICUT

Certificate

This is to certify that the project work entitled “**Extraction of Questions from the Internet using a Machine Learning approach**”, submitted by Alok Saw (B090924CS), Jerrin Shaji George (B090437CS), Shubhangam Agrawal (B090904CS) and Stein Astor Fernandez (B090006CS) to National Institute of Technology Calicut towards partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering, is a bonafide record of the work carried out by them under my supervision and guidance.

Place : Calicut
Date :

Project Guide
Dr. Priya Chandran
Professor

Head of Department

Office Seal

Abstract

We propose a Support Vector Machine based approach to extract questions pertaining to a topic from the internet. We adapt certain features from *SQUINT* [1] to identify relevance of text and propose two new features to identify questions in a text *section*.

This Machine Learning component is developed as an application which takes a topic as input and produces a list of questions related to the input topic as output.

Problem Statement

Given input a topic t , search the internet for pages which may contain questions related to t and output a set R comprising of relevant questions extracted from these pages.

Contents

I	Introduction	4
1	Literature Survey	1
II	Work Done in S7	3
2	Architecture	4
2.1	Query Generation	5
2.2	Search Engine	5
2.3	Page Preprocessor	5
2.4	Feature Generator	6
2.4.1	Word Rank Based Features	6
2.4.2	Bigram Rank Based Features	6
2.4.3	Coverage of Top Ranked Tokens	7
2.4.4	Coverage of Interrogative Indicators	7
2.4.5	Absence of Specific Keywords	7
2.5	Labelling (Training Set Generation)	8
2.6	Support Vector Machine	8
2.7	Output	8
3	Prototype	9
III	Work Done in S8	10
4	Class Design	11
4.1	Main Component	12
4.2	Data Component	12
4.3	Query Component	13
4.4	Section Component	13
4.5	Feature Component	14
4.6	SVM Component	15
5	Data Structure Design	16
5.1	Node Design	16
5.2	Tree Design	17
5.2.1	Query Generation	18
5.2.2	Link Generation	18
5.2.3	Section Generation	19

5.2.4	Labelling	19
5.2.5	Feature Generation	20
5.2.6	SVM Classification	20
6	Data File Design	21
7	Config File Design	23
8	SVM Package Identification	24
9	Section Generation Heuristic	25
10	Frequency List Generation	26
10.1	Unigram Frequency Generation	26
10.2	Bigram Frequency Generation	26
11	Implementation and Analysis	27
11.1	Training Data Generation	27
11.2	Performance Measure	28
11.3	Feature Analysis	29
11.3.1	Procedure	29
11.3.2	Observations	30
11.4	Training Set Size Analysis	34
11.4.1	Procedure	34
11.4.2	Observation	34
11.5	Proposed Features Analysis	35
11.5.1	Procedure	35
11.5.2	Observation	35
11.6	Real Run	36
12	Conclusion and Future Work	37
	References	38

List of Figures

2.1	Architecture Diagram	4
4.1	Component Diagram	11
4.2	Main Component	12
4.3	Data Component	12
4.4	Query Component	13
4.5	Section Component	13
4.6	Feature Component	14
4.7	SVM Component	15
5.1	Node	16
5.2	The data tree	17
5.3	Query Generation Data	18
5.4	Link Generation Data	18
5.5	Section Generation Data	19
5.6	Labelling Data	19
5.7	Feature Data	20
5.8	SVM Data	20
6.1	data.xml example	22
7.1	config.properties	23
11.1	Word Rank Features	30
11.2	Bigram Rank Features	31
11.3	Word Coverage Features	32
11.4	Optimal Feature Configuration Detection	33
11.5	Learning Curve	34
11.6	Proposed Features	35
11.7	Sample Output for Topic – Game Theory	36

Part I

Introduction

Introduction

Today, there is a huge amount of data available on the internet. While advancements in search technologies have made searching for relevant pages trivial, each page contains a lot of irrelevant data, images and advertisements. Thus, having some way to extract only the relevant sections of data from the huge set of search results would result in massive effort and time savings.

There is often a need for students as well as faculty to obtain a list of questions related to certain topics while studying or teaching these topics. The internet has a vast number of such questions for any such topic, however one must know what terms to search for and then manually visit each page returned in the search results and find out such questions in a tedious manner.

We plan to use Machine Learning techniques to identify whether a section of text is a question and relevant to a topic and thus make an application to automate this process which will take a topic as input and return a list of questions based on the topic from the internet.

Chapter 1

Literature Survey

The trivial way of approaching this problem would be to simply search for the presence of the input topic in the various *sections* and output matches — however this would not yield a good performance and would have no way to identify questions. Teevan et al. [2] have discussed the relevance of using search methodologies that focus more on contextual information as opposed to simple keyword matching as that may not be enough to capture the relevant text.

Thus rather than simple keyword matching, we must be able to identify patterns in the text *sections* which make it a question and also make it relevant to the input topic. Machine Learning is a good approach to solve this problem as it finds the various patterns in the text without the need for detailed statistical analysis.

Arthur Samuel defines Machine Learning as

The field of study that gives computers the ability to learn without being explicitly programmed.

Machine learning, a branch of artificial intelligence, is concerned with the design and development of algorithms that take as input empirical data and yield patterns or predictions thought to be features of the underlying mechanism that generated the data.

A learner can take advantage of examples (data) to capture characteristics of interest of their unknown underlying probability distribution. Data can be seen as instances of the possible relations between observed variables.

A major focus of machine learning research is the design of algorithms that recognize complex patterns and make intelligent decisions based on input data [3].

Machine learning problems can be categorized broadly into 2 categories [4] —

Supervised Learning The goal is to predict the value of an outcome measure based on a number of input measures.

Unsupervised Learning There is no outcome measure, and the goal is to describe the associations and patterns among a set of input measures.

As our problem is to predict whether each *section* is a related question or not, we see that ‘Supervised Learning’ is the correct approach to our problem.

Joachims has also concluded [5] that Support Vector Machines are an effective text classification tool which uses the Supervised Machine Learning model. This is due to the high dimensionality of the feature vector [1].

Inoue et al. [1] propose an SVM based approach to identify the most relevant subsection in Web pages returned by a search query (*SQUINT*). They have obtained very accurate results using this model.

A Support Vector Machine takes a set of input data and predicts, for each given input, which of two possible classes forms the output, making it a non-probabilistic binary classifier.

It constructs a hyperplane or a set of hyperplanes in a high- or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane (called the *functional margin*) that has the largest distance to the nearest training data point of any class, since in general the larger the margin, the lower the generalization error of the classifier [6].

Our problem is similar to *SQUINT* with the significant challenge being the identification of questions. Thus, we explore a Support Vector Machine based approach similar to *SQUINT* in order to detect relevance to the input topic and aim to identify various features which indicate that a *section* is a question.

Part II

Work Done in S7

Chapter 2

Architecture

We identified a component-based architecture for the application which would be feasible to implement.

A component based architecture is suitable for this application since each component has a well defined function which is dependent only on the output of the other subsections.

This will help in having a de-coupled structure and each component can be modified without affecting the other components. We can also observe the state of data at each *section* for better debugging and analysis.

The high-level overview of the architecture is as follows —

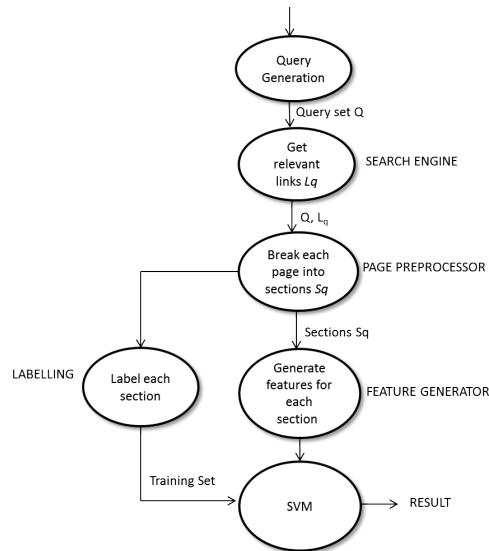


Figure 2.1: Architecture Diagram

2.1 Query Generation

- Take the topic t as input.
- Query Set $Q := T \times \{\text{'questions'}, \text{'problems'}, \text{'solved examples'}\}$.

Firstly, we will take the input of the topic t .

Topic t will be appended with chosen suffixes such as *'questions'* and *'problems'* to generate the *query set* Q which will be queried on the search engine.

2.2 Search Engine

$\forall q \in Q :$

- Search the internet for the query phrase q using *Google Custom Search API* (or any other search engine).
- Add the top $nLinksPerQuery$ links to set L_q .

2.3 Page Preprocessor

$\forall l \in L_q :$

- Load the link l using *Jsoup* package and use the *HtmlToPlainText* class to strip away all markup data, images etc. and yield only the text portions.
- The text of the page is then broken up into *sections* on the basis of —
 - Number of lines
 - Paragraph Structure
 - Interrogative Indicators - *'what', 'how', 'explain', 'define', '?', 'elucidate'* etc.
- Each *section* s thus obtained is added to the *section set* S_q .

For each *query* phrase $q \in Q$, we will load each link we have in our link set L_q using *Jsoup* package functions and use the *HtmlToPlainText* class to strip it of all useless markup data, links and pictures and parse out only the text portions available on the page which may contain the questions.

Each page will then be broken into various *sections*. This will be done by a parser which will determine *sections* which may be questions on the basis of a maximum number of lines, the paragraph tags of html and interrogative indicators like a line ending with a *'?'* or lines starting with words like *'what', 'define'* etc. as these are likely to be starting or ending words of a question.

Each *section* s thus obtained will be added to the *section set* S_q for a *query*.

2.4 Feature Generator

The feature generator will generate the features and statistics for each *section*, which will be analysed by the SVM to create the regression model.

We adapt the first three features from *SQUINT* to determine the relevance of a *section* to a *query* [1]. Furthermore, we propose two additional features to determine whether a *section* contains a question or not.

The features are —

2.4.1 Word Rank Based Features

The rank of a word is defined to be its position in the list if the words were ordered by frequency of occurrence in S_q [1].

We would have a feature each for say, the top 200 most frequent words in S_q .

For the i^{th} ranked word, this feature would basically have the value for the frequency of this word in the current *section*.

We can limit the dimensionality of the input vector by bucketing words by a certain range of ranks. For example, we can bucket ranks 1-5, 6-10, 11-15...etc to aggregate word counts, and come up with a feature vector of reduced dimensionality.

We also normalize for the length of the *section* since we do not want to be biased towards long *sections*.

The optimal number of words and the bucket size needs to be determined experimentally.

2.4.2 Bigram Rank Based Features

A bigram is defined to be two consecutive words occurring in a *section* [1].

This feature is computed in a manner similar to the previous set of features.

This feature is based on the intuition that the correlation between two words might be more informative than the words taken individually.

For instance, ‘*machine learning*’ suggests a stronger relation to a query ‘*AI SVM*’ than the individual words ‘*machine*’ or ‘*learning*’.

For this feature as well, we will adjust the dimensionality by bucketing and limiting coverage of ranks depending on experimental results.

2.4.3 Coverage of Top Ranked Tokens

Relevance to a topic may also be captured by the coverage of top ranked token types in the *section* [1].

For example, if we have a bucket size of 5, we might be interested in knowing how many of the top 5 ranked words occur in this *section*, how many of the next 5 highly ranked words occur in this *section* and so forth.

For instance, if the top 5 ranked token types are ‘*learning*’, ‘*machine*’, ‘*data*’, ‘*access*’, and ‘*database*’, and a *section* contains ‘*learning*’ and ‘*data*’, the corresponding value for this feature is 2.

2.4.4 Coverage of Interrogative Indicators

Presence of words such as ‘*what*’, ‘*why*’, ‘*explain*’, ‘*define*’, ‘*elucidate*’ etc. are strong indicators that the *section* contains a question.

Thus we propose this feature whose value is the coverage of a predefined set of such interrogative indicators present in the *section*.

We need to experimentally determine whether the frequency or coverage of interrogative indicators is a better feature.

2.4.5 Absence of Specific Keywords

There are certain words or patterns, the presence of which strongly indicate a *section* of text to not have a question.

For example, if a sentence begins with a ‘*Yes*’ or ‘*No*’; or certain words like ‘*because*’ are present, there is a high chance that the *section* is not a question.

We check for the coverage of such words in the *section* using a pre-defined list.

2.5 Labelling (Training Set Generation)

For generating the training set, we will manually label a set of *sections* as to whether they are relevant questions of the given topic or not and this data will be used to train the SVM to generate the model.

2.6 Support Vector Machine

This component will be trained using the Training Set comprising of the *section* sets with the generated feature vectors and manually tagged labels. The training data will be analysed to generate a SVM model.

During the real runs of the application, the model generated during training will be used by the SVM to identify the relevant *sections* and tag them appropriately.

The *sections* tagged as '*relevant*' will be added to the result set *R*.

2.7 Output

The output component will take all the *sections* tagged as '*relevant question*' present in set *R* and output them in a clean format.

Chapter 3

Prototype

Java was chosen as the programming language owing to the availability of well supported classes to implement SVMs and the object-oriented features in general.

We made a prototype program with placeholder stub functions for each component. Each stub did no useful work but just passed the data along.

Each component would be individually developed and further refined to produce the optimal output.

Part III

Work Done in S8

Chapter 4

Class Design

The overall design consists of six components.

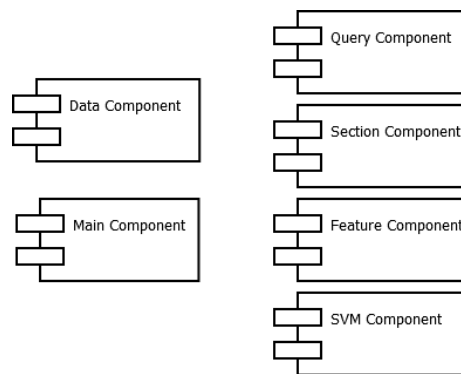


Figure 4.1: Component Diagram

4.1 Main Component

This component is responsible for calling all other components and finally generating the list of questions as output.

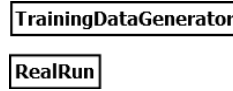


Figure 4.2: Main Component

TrainingDataGenerator Calls all components in order and calls the *labeller* so that the training data can be generated.

RealRun Once the SVM model is ready, this is used to enter a topic and get list of questions as output.

4.2 Data Component

The *data component* stores the overall data and its classes provide interfaces to access and modify this data. All other components necessarily interact with this component.

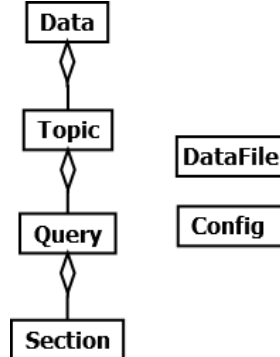


Figure 4.3: Data Component

DataFile Interface to the data file.

Config Has all the config parameters as set in *config.properties* file.

Data Stores a list of *topic objects*.

Topic Stores the input topic and a list of *query objects*.

Query Stores the details related to each *query*. Has a list of associated *section objects*.

Section Stores the *section* text and relevant attributes.

4.3 Query Component

The *query component* is responsible for taking the input topic and generating queries related to the same.



Figure 4.4: Query Component

TopicListGenerator Generates a list of related and/or sub-topics related to the input.

QueryGenerator Generates list of *query objects* by adding various suffixes to the given topic.

4.4 Section Component

The *section component* processes each *query* and generates a list of *section objects* for each *query*.



Figure 4.5: Section Component

LinkGenerator Uses a Search API to search for the *query* and generate links of the top results.

HTMLParser Uses the *JSoup* package to load links and return plaintext form of the web-pages.

SectionGenerator Breaks pages into sections and makes a *section object* for each.

4.5 Feature Component

The *feature component* has classes to generate all the requisite features for the *sections*.

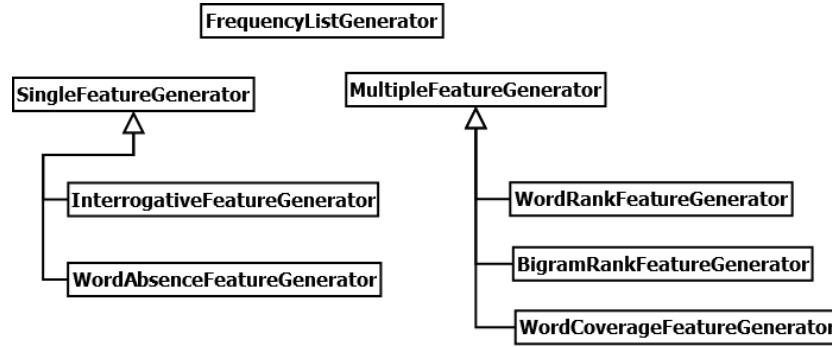


Figure 4.6: Feature Component

FrequencyListGenerator Processes all the *sections* related to a *query* to generate a list of words and bigrams in descending order of frequency.

SingleFeatureGenerator Interface for feature-types that generate a single feature.

MultipleFeatureGenerator Interface for feature-types that generate a list of features.

WordRankFeatureGenerator Generates the Word Rank based feature list of the *section*.

BigramRankFeatureGenerator Generates the Bigram Rank based feature list of the *section*.

WordCoverageFeatureGenerator Generates the Word Coverage based feature list of the *section*.

InterrogativeFeatureGenerator Generates the Interrogative Indicator frequency based feature of the *section*.

WordAbsenceFeatureGenerator Generates the feature based on absence of specific words of the *section*.

4.6 SVM Component

Consists of all the classes which are related to the machine learning aspect of the program.

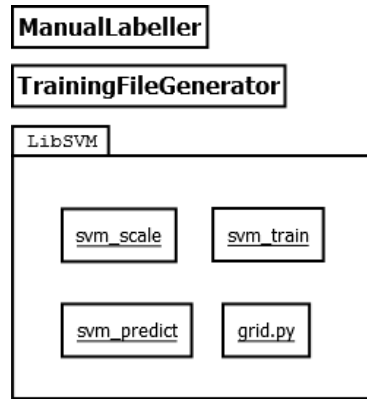


Figure 4.7: SVM Component

ManualLabeller Loads each *section* one by one and allows the user to manually label the *sections*.

TrainingFileGenerator Generates the Training File with all labels and features in the correct format.

LibSVM The Support Vector Machine library. It consists of the following —

svm_scale Scales the training file features.

svm_train Trains the SVM and generates the model file.

svm_predict Predicts the output as per model file for a data file.

grid.py Python script to determine best C^1 and γ^2 values for the SVM.

¹The SVM parameter C trades off misclassification of training examples against simplicity of the decision surface [14].

²The RBF Kernel parameter γ defines how far the influence of a single training example reaches [14].

Chapter 5

Data Structure Design

We have chosen to use an n-ary tree as the data structure keeping in mind the component-based architecture.

The tree structure allows us to model the data easily while being completely extensible at any point. It allows quick access to required data and allows us to group data in a functional manner.

5.1 Node Design

The node will have the following attributes :

Type The type of the node eg. *Topic, Query, Section*

Data The corresponding data eg. *'SVM Questions'*

ChildList List of child nodes

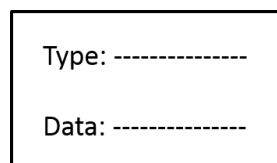


Figure 5.1: Node

5.2 Tree Design

The tree will look like the following —

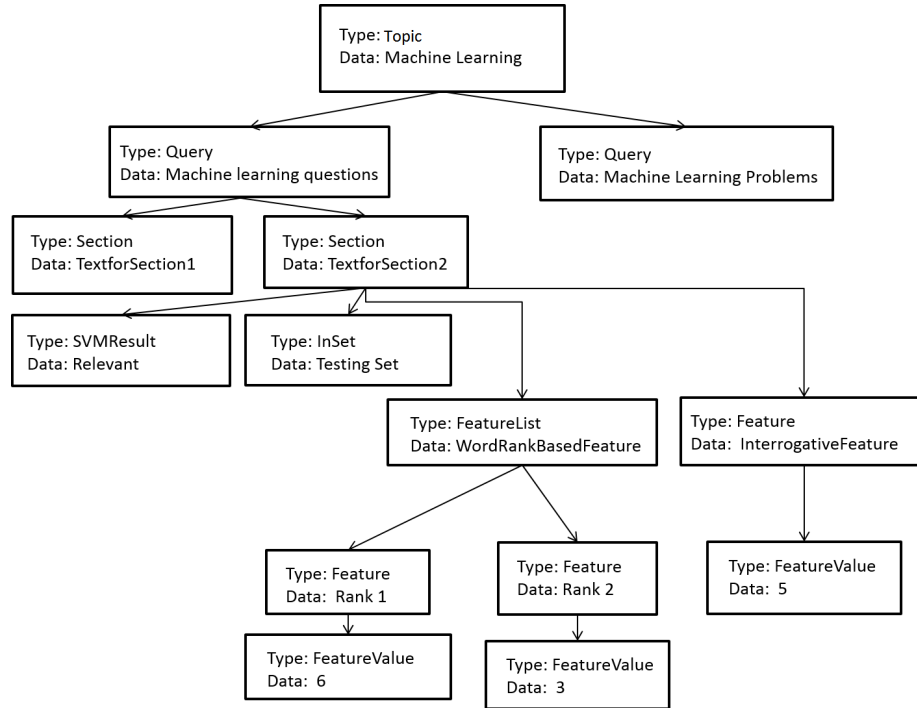


Figure 5.2: The data tree

Each node will have its corresponding data in its child nodes.

As each component is run, it will add or delete nodes as required.

The Data Tree Life–Cycle

5.2.1 Query Generation

The Query Generator generates a set of *queries* to search for from the given search *topic* and creates a set of corresponding nodes as children to the root node, as shown.

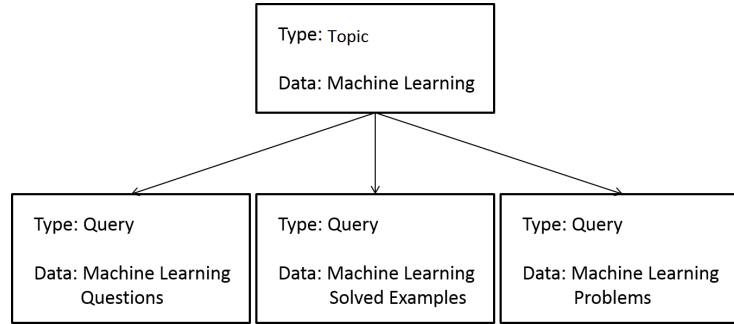


Figure 5.3: Query Generation Data

5.2.2 Link Generation

For each of the *queries* generated in the previous step, a set of links corresponding to the search results for that *query* is generated as children of each *query* node.

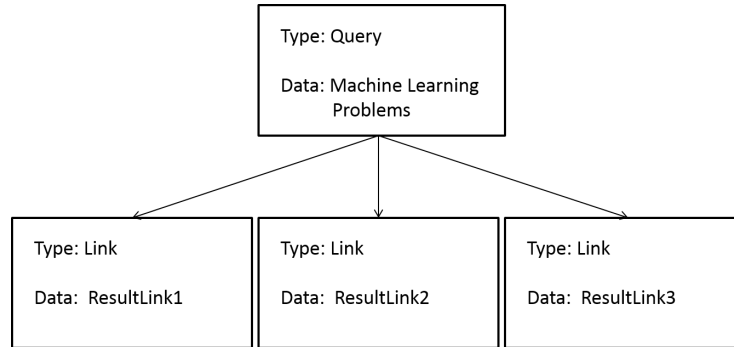


Figure 5.4: Link Generation Data

5.2.3 Section Generation

The *Section Generator* parses each of the links generated in the previous step, removing unnecessary data and then divides each such page into *sections*, each of which may contain a question relevant to the topic at hand.

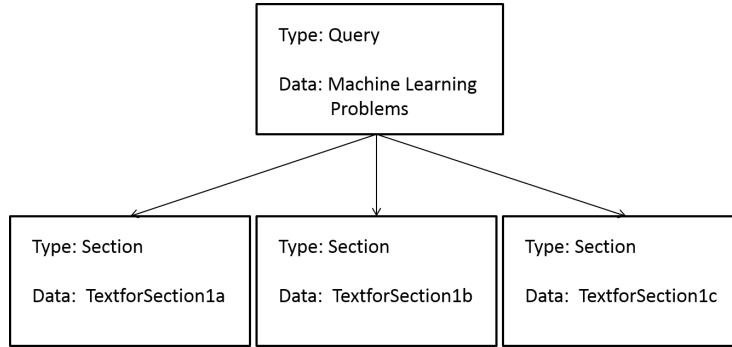


Figure 5.5: Section Generation Data

5.2.4 Labelling

Sections intended to be part of the training or testing set will be labelled as relevant or not relevant. This is indicated by a corresponding child node for that *section* node, as shown.

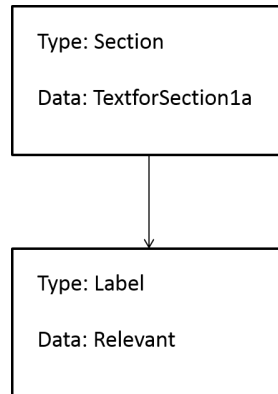


Figure 5.6: Labelling Data

5.2.5 Feature Generation

The features of a *section* are similarly represented by corresponding child nodes of each feature category.

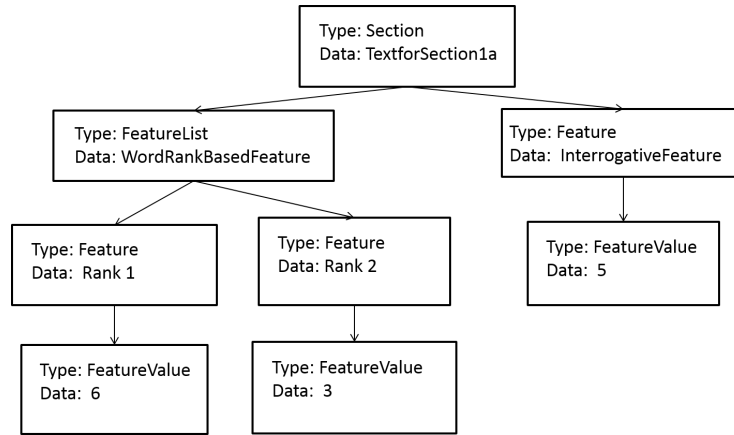


Figure 5.7: Feature Data

5.2.6 SVM Classification

The results of the Support Vector Machine classification and the set to which the result belongs are also represented as shown.

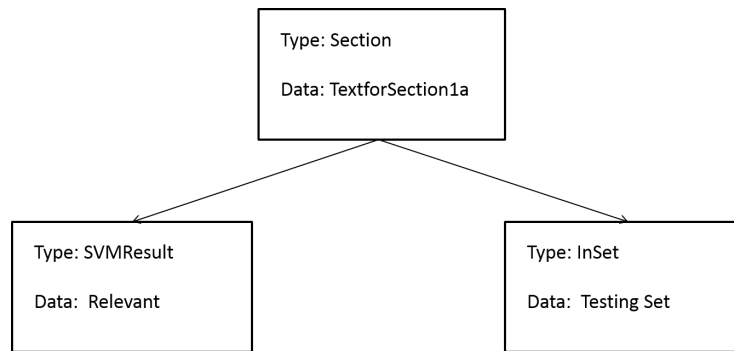


Figure 5.8: SVM Data

Chapter 6

Data File Design

We will be using the *XML* file format for the data file since the tree-like structure of the *XML* format is highly suitable for representing a tree data structure [9].

The file, *data.xml*, will be a representation of the data stored in a tree format.

Each node of the tree will be represented as an element¹ in the *XML* file delineated by the tags of the node-type. Each child-node in the tree will be represented as child-elements of the corresponding parent node.

The key features of this design are its performance, extensibility and transparent nature which allows us to analyse the file at any point in the application's life-cycle.

The Java *JAXB*² feature is capable of loading the data from the file directly into a tree format as per the class definitions. This will allow us to easily read, write or update the data file as required.

¹A logical document component which either begins with a start-tag and ends with a matching end-tag. The characters between the start- and end-tags, if any, are the element's content and may contain other elements, which are called child elements. Eg. `<Type>Data</Type>`.

²*Java Architecture for XML Bindings* is capable of writing/reading annotated object data to/from an *XML* file.

```

1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <data>
3      <topic>
4          <text>operating systems</text>
5          <query>
6              <text>operating systems questions</text>
7              <linkList>
8                  <link>http://www.techinterviews.com/operating-system-questions</link>
9                  <link>
10                     http://www.careercup.com/page?pid=operating-system-interview-question
11                     s</link>
12                     <link>http://www.computerhope.com/os.htm</link>
13                     <link>http://www.indiabix.com/technical/operating-systems/</link>
14                     <link>
15                         http://windows.microsoft.com/en-us/windows-vista/32-bit-and-64-bit-wi
16                         ndows-frequently-asked-questions</link>
17                 </linkList>
18                 <sectionList>
19                     <section>
20                         <text>Operating system questions | TechInterviews Search Tech
21                         Interviews Tech Interviews Prepare for job interviews with the
22                         questions and answers asked by high-tech employers Skip to
23                         content * .NET * C++ * Database * General * Hardware * Java *
24                         Networking * Puzzles * SAP ABAP * Testing * Unix/Linux * VB *
25                         Web dev * Windows Hardware , Unix/Linux , Windows &gt;&gt;
26                         Operating system questions « 8086 interview questions Jakarta
27                         struts questions» Operating system questions By admin | January
28                         17, 2005 * What are the basic functions of an operating system?
29
30                         - Operating system controls and coordinates the use of the hardware among the
31                         various applications programs for various uses.
32                     </text>
33                     <manualLabel>1</manualLabel>
34                     <svmLabel>0</svmLabel>
35                     <interrogativeFeature>0.0</interrogativeFeature>
36                     <wordAbsenceFeature>0.0</wordAbsenceFeature>
37                 </section>
38                 <section>
39                     <text>Operating system acts as resource allocator and manager.
40                     Since there are many possibly conflicting requests for resources the operating
41                     system must decide which requests are allocated resources to operating the computer

```

Figure 6.1: data.xml example

Chapter 7

Config File Design

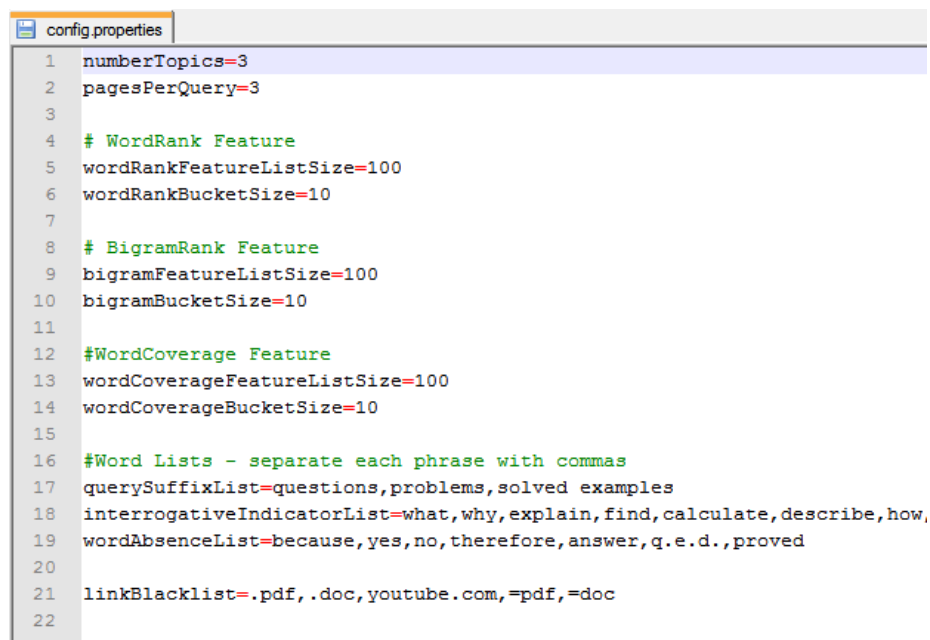
The file *config.properties* will have all the configurable parameters in the Java *Properties* format.

The values will be loaded from this file every time the application runs. Thus we need not re-compile the entire application just to change the parameters.

The parameters can be changed in a simple manner using any text-editor.

Thus the user is free to change the parameters as per his/her requirements.

This will help reduce wastage of time during testing and analysis of parameters by eliminating needless re-compilations.

A screenshot of a text editor window titled 'config.properties'. The editor shows 22 lines of code in a Java Properties format. Lines 1-2 are 'numberTopics=3' and 'pagesPerQuery=3'. Lines 4-6 are for 'WordRank Feature' with 'wordRankFeatureListSize=100' and 'wordRankBucketSize=10'. Lines 8-10 are for 'BigramRank Feature' with 'bigramFeatureListSize=100' and 'bigramBucketSize=10'. Lines 12-14 are for 'WordCoverage Feature' with 'wordCoverageFeatureListSize=100' and 'wordCoverageBucketSize=10'. Lines 16-19 are for 'Word Lists' with values: 'questions,problems,solved examples', 'what,why,explain,find,calculate,describe,how,', 'because,yes,no,therefore,answer,q.e.d.,proved', and '.pdf,.doc,youtube.com,=pdf,=doc'. Line 21 is 'linkBlacklist=.pdf,.doc,youtube.com,=pdf,=doc'. Line 22 is empty.

```
1 numberTopics=3
2 pagesPerQuery=3
3
4 # WordRank Feature
5 wordRankFeatureListSize=100
6 wordRankBucketSize=10
7
8 # BigramRank Feature
9 bigramFeatureListSize=100
10 bigramBucketSize=10
11
12 #WordCoverage Feature
13 wordCoverageFeatureListSize=100
14 wordCoverageBucketSize=10
15
16 #Word Lists - separate each phrase with commas
17 querySuffixList=questions,problems,solved examples
18 interrogativeIndicatorList=what,why,explain,find,calculate,describe,how,
19 wordAbsenceList=because,yes,no,therefore,answer,q.e.d.,proved
20
21 linkBlacklist=.pdf,.doc,youtube.com,=pdf,=doc
22
```

Figure 7.1: config.properties

Chapter 8

SVM Package Identification

We have identified the *LIBSVM* [10] library to implement the Support Vector Machine.

We are using *LIBSVM* because it has functions for the *Radial Basis Kernel* [11].

This kernel nonlinearly maps samples into a higher dimensional space so it, unlike the linear kernel, can handle the case when the relation between class labels and attributes is nonlinear [11].

LIBSVM is a mature SVM library for Java and has good performance and high accuracy [11].

Chapter 9

Section Generation Heuristic

Paragraph and sentence boundary detection is done using the *SentParDetector* class from the *Spiatools* library by Scott Piao [19].

It is a highly accurate sentence and paragraph breaker, employing heuristic rules for identifying boundaries.

Sentences returned by the *SentParDetector* class having length more than 400 characters are again attempted to be broken down using the standard *BreakIterator* class provided by Java.

As questions are likely to be contained in fewer than four sentences, every paragraph returned by the *SentParDetector* class containing six or more sentences is broken up into half, and the check is repeated on the individual paragraphs too.

Chapter 10

Frequency List Generation

The *FrequencyListGenerator* class generates bigram and unigram word frequencies for *query objects* and *section objects*.

The text content from the corresponding *object* is broken into tokens using the *StandardTokenizer* class from the *Apache Lucene* library [16]. *StandardTokenizer* is a grammar-based tokenizer constructed with *JFlex* [17]. This class implements the *Word Break* rules from the *Unicode Text Segmentation* algorithm, as specified in Unicode Standard Annex #29 [18].

10.1 Unigram Frequency Generation

The tokens generated by the *StandardTokenizer* are iterated over. Each token is mapped to a hashtable using the token string as the key, and setting the frequency of occurrence of the token in the text as the hash value. The hashmap is sorted in the decreasing order of token frequencies, and the corresponding *object* word frequency list is set as this hashmap.

10.2 Bigram Frequency Generation

ShingleFilter class from *Apache Lucene* library helps in constructing shingles (token n-grams) from a token stream. The bigram frequency generator uses the *ShingleFilter* class to generate bigram tokens from the tokens generated by the *StandardTokenizer*. These bigram tokens are then mapped to a hashtable in a similar manner to unigram frequency generation to obtain bigram token frequencies, and this hashmap is set as the bigram frequency list of the corresponding *object*.

Chapter 11

Implementation and Analysis

The entire program was successfully implemented in Java as per the proposed design.

11.1 Training Data Generation

The Training Data was generated by running *EQML1* for 3 topics — ‘*operating systems*’, ‘*data structures and algorithms*’, ‘*computer architecture*’. This resulted in a total of 6467 *sections* being generated which were saved in *data.xml*.

- 5223 *sections* were marked irrelevant.
- 1243 *sections* were marked relevant.

EQML2 was used to generate the SVM Feature files by first loading the *data.xml* file and then calculate the features based on various combinations of the configurable parameters.

Since we have 4 times the number of irrelevant *sections* vs. relevant *sections*, we use a weightage of 4 for relevant *sections* and weightage of 1 for irrelevant *sections* while training the SVM to avoid unbalanced data bias [13].

11.2 Performance Measure

Let x_1, x_2, \dots, x_n be the testing data and $f(x_1), f(x_2), \dots, f(x_n)$ be the decision values by the SVM. If true labels (manual labels) are denoted by y_1, y_2, \dots, y_n then accuracy can be thought of as the percentage of correctly predicted data vs. the total testing data [11].

$$t_i = \begin{cases} 1 & \text{if } f(x_i) = y_i \\ 0 & \text{if } f(x_i) \neq y_i \end{cases}$$

$$Accuracy = \frac{\sum t_i}{n} \times 100\%$$

Cross-validation is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set [15].

It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice.

Cross-validation is important in guarding against testing hypotheses suggested by the data (*“Type III errors”*) [15].

In *k-fold cross-validation*, the original sample is randomly partitioned into k equal size subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining $k - 1$ subsamples are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results from the folds then can be averaged to produce a single estimation.

The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once [15].

5-fold Cross Validation Accuracy was chosen as the performance measure.

11.3 Feature Analysis

11.3.1 Procedure

For each configuration of features —

1. Features were generated by configuring the *config.properties* file
2. The features were scaled¹ to $[0, 1]$ using *svm_scale*
3. Optimal C and γ SVM parameters were determined using *grid.py*
4. *5-fold Cross Validation Accuracy* was determined

¹Scaling is done to avoid the problem of features having greater numeric range dominating the results [12].

11.3.2 Observations

Word Rank Based Features

The Word Rank Feature set size was varied while keeping the other features constant.

- Word Rank Bucket Size = 1
- Bigram Rank Features = 150
- Bigram Rank Bucket Size = 1
- Word Coverage Based Features = 150
- Word Coverage Bucket Size = 5

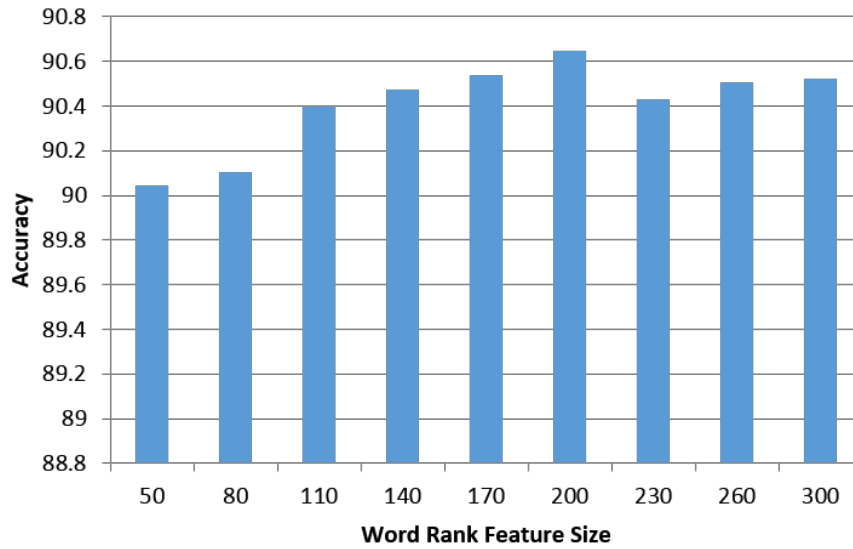


Figure 11.1: Word Rank Features

The accuracy increased with the set size upto a point and then varied. This leads us to believe that the first 150 word ranks are significant beyond which the data becomes less relevant.

Bigram Rank Based Features

The Bigram Rank Feature set size was varied while keeping the other features constant.

- Word Rank Features = 150
- Word Rank Bucket Size = 1
- Bigram Rank Bucket Size = 1
- Word Coverage Based Features = 150
- Word Coverage Bucket Size = 5

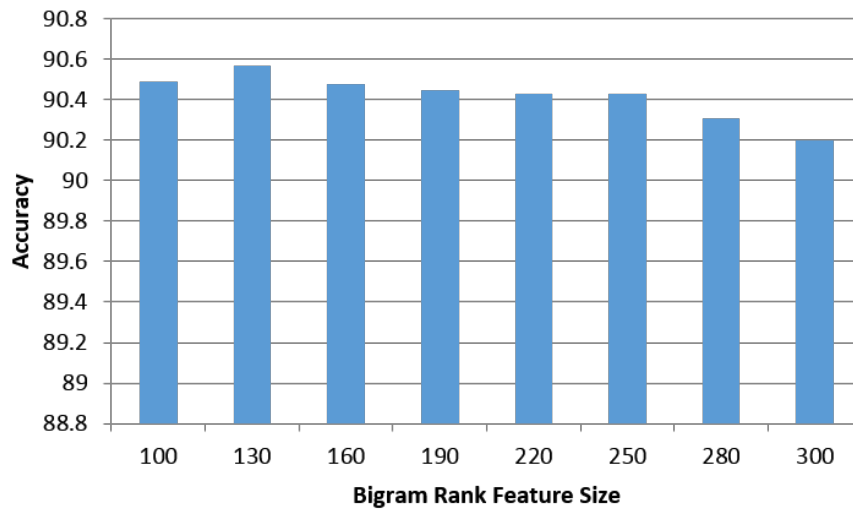


Figure 11.2: Bigram Rank Features

The accuracy was higher around the 130–150 mark but beyond that the accuracy dropped. This is probably due to the first few bigrams capturing data about the topic but beyond that random bigrams make the data too noisy to matter.

Word Coverage Based Features

The Word Coverage Feature set size was varied while keeping the other features constant.

- Word Rank Features = 150
- Word Rank Bucket Size = 1
- Bigram Rank Features = 150
- Bigram Rank Bucket Size = 1
- Word Coverage Bucket Size = 5

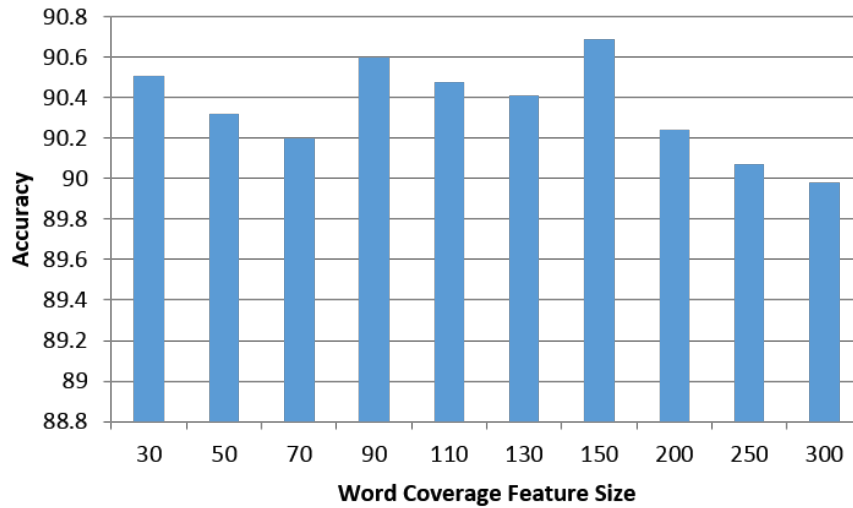


Figure 11.3: Word Coverage Features

There was significant variation in accuracy as the number of features changed. The maximum accuracy was observed at 150 features.

Optimal Feature Configuration

Based on the insights obtained, all the features were varied and the optimal accuracy was determined.

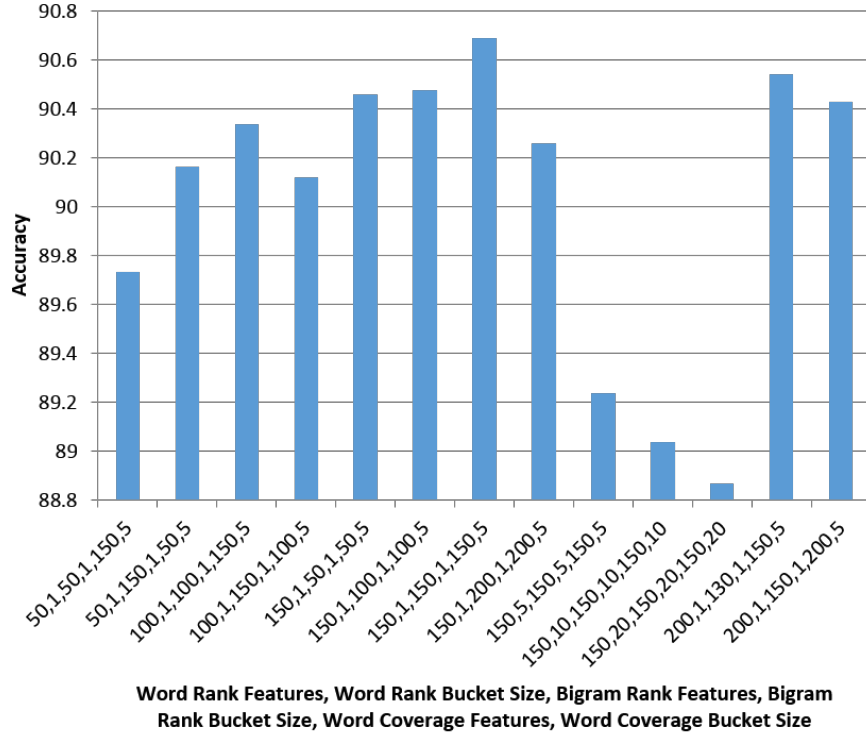


Figure 11.4: Optimal Feature Configuration Detection

Optimal Configuration — Accuracy 90.69%

- Word Rank Features = 150
- Word Rank Bucket Size = 1
- Bigram Rank Features = 150
- Bigram Rank Bucket Size = 1
- Word Coverage Features = 150
- Word Coverage Bucket Size = 5

Optimal SVM parameters

- $C = 8.0$
- $\gamma = 0.125$

This feature set was used to generate the model for the Real Run.

11.4 Training Set Size Analysis

11.4.1 Procedure

The effect of Training Set size on the accuracy was determined by —

1. Training Set was generated using stratified subsampling² of the Data Set
2. The features were scaled³ to $[0, 1]$ using *svm_scale*
3. Optimal C and γ SVM parameters were determined using *grid.py*
4. SVM Model was generated using the Training Set
5. Accuracy was calculated by predicting the Data Set

The optimal feature configuration determined earlier was used for this analysis.

11.4.2 Observation

We see that the accuracy increases exponentially with training set size.

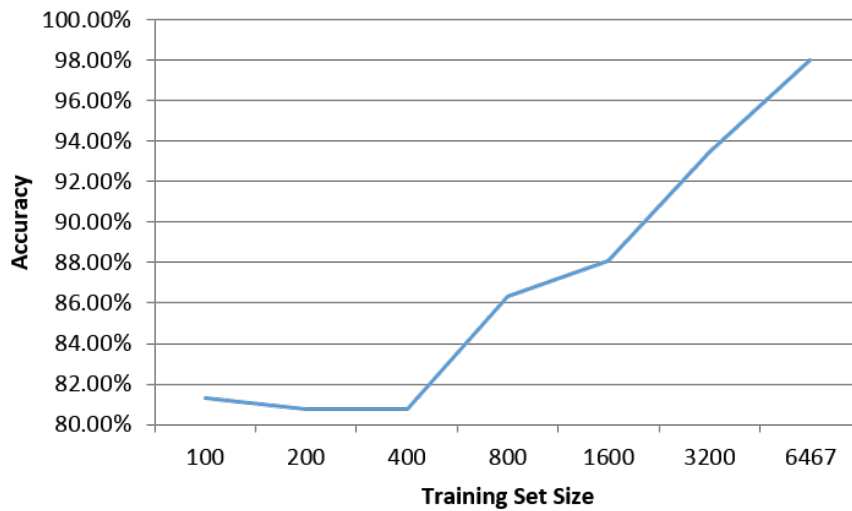


Figure 11.5: Learning Curve

Thus we choose to make the final model with the full 6467 samples as the Training Set.

Higher number of samples is avoided to keep the training time low as training time is proportional to $(SetSize)^2$ [11] and to avoid the problem of over-fitting⁴.

²Stratified sampling using proportionate allocation uses a sampling fraction in each of the strata that is proportional to that of the total population. For instance, if the population consists of 60% in the male stratum and 40% in the female stratum, then the relative size of the two samples (three males, two females) should reflect this proportion [20].

³Scaling is done to avoid the problem of features having greater numeric range dominating the results [12].

⁴Overfitting occurs when a statistical model describes random error or noise instead of the

11.5 Proposed Features Analysis

11.5.1 Procedure

The effect of our proposed features was studied by —

1. Configurations of the Training File were generated by selecting combinations of the 3 classes⁵ of features⁶
2. The features were scaled⁷ to $[0, 1]$ using *svm_scale*
3. Optimal C and γ SVM parameters were determined using *grid.py*
4. 5-fold Cross Validation Accuracy was determined

11.5.2 Observation

The Interrogative feature succeeds in identifying whether a *section* is a question but is not capable of determining irrelevant questions by itself.

Using the Interrogative feature with the *SQUINT* features results in a 90.55% accuracy and adding the Word Absence features results in accuracy of 90.69%.

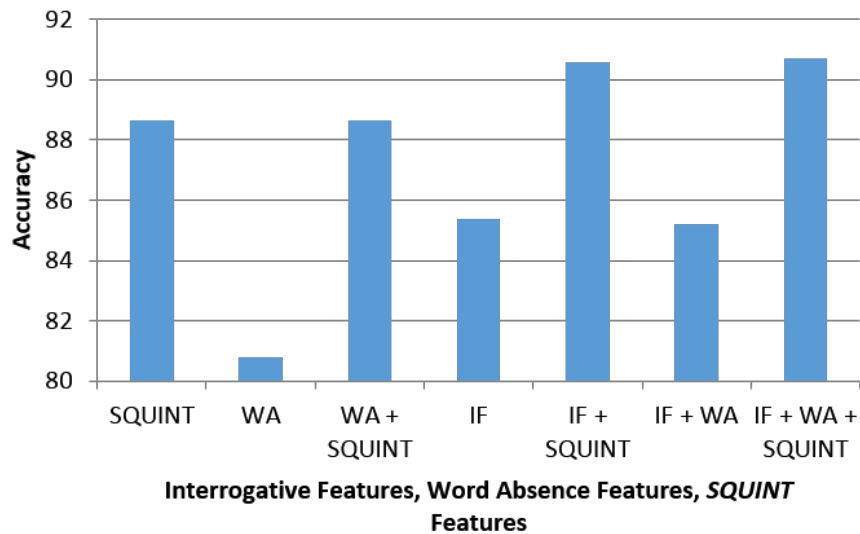


Figure 11.6: Proposed Features

Thus it is optimal to use all the features.

underlying relationship. Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. A model which has been overfit will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data [21].

⁵Interrogative Features, Word Absence Features and *SQUINT* Features

⁶The trivial case with no features was discarded

⁷Scaling is done to avoid the problem of features having greater numeric range dominating the results [12].

11.6 Real Run

On giving a topic as input, the program searches the web for relevant pages, retrieves these pages, breaks them into *sections*, generates the requisite features and then the *svm_predict* module is called to get the predictions of whether the *sections* are relevant or not.

The SVM uses the optimal model generated using our manually labelled data with the optimal feature set.

The *sections* which are predicted to be relevant are stored in a file, *realOutput.txt*, as output and this contains the required list of questions relevant to the topic given as input.

```
96
97 1. What precisely defines a strategic situation?
98 Why is rational choice more complicated in strategic situations?
99
100 2. What is a Nash equilibrium?
101 Are Nash equilibria necessarily Pareto optimal?
102 Why or why not?
103
104 3. What does one need to know in order to define a "game"?
105
106 b.
107 What would game theorists predict will happen?
108 What do you think actually happens?
109 Why?
```

Figure 11.7: Sample Output for Topic – Game Theory

Chapter 12

Conclusion and Future Work

Our proposed method of using a SVM to classify relevant questions from the internet seems successful.

Future work may comprise of using different types of classifiers for the Machine Learning component like Neural Networks.

Our solution discards the information when a *section* is relevant to the topic but is not a question as it is marked as not-relevant. A multi-class SVM classification approach may be used to capture this data into four classes — Not Relevant Not Question, Not Relevant Question, Relevant Not Question, Relevant Question.

Other features to better analyse whether a *section* is a question or not may also be explored to improve the accuracy of the classification.

References

- [1] SQUINT - SVM for Identification of Relevant Sections in Web Pages for Web Search, **Riku Inoue, Siddharth Jonathan J.B., Jyotika Prasad**, *Department of Computer Science, Stanford University*
- [2] The Perfect Search Engine is Not Enough : A Study of Orienteering Behavior in Directed Search, **Jaime Teevan, Christine Alvarado, Mark S. Ackerman and David R. Karger**, *Proceedings of the SIGCHI conference on Human factors in computing systems. pp. 415-422, April 2004.*
- [3] Wikipedia article on Machine Learning, http://en.wikipedia.org/wiki/Machine_learning
- [4] The Elements of Statistical Learning: Data Mining, Inference, and Prediction, **Trevor Hastie, Robert Tibshirani, Jerome Friedman**
- [5] Text Categorization with Support Vector Machines: Learning with Many Relevant Features, **Thorsten Joachims**, *Universitat Dortmund, Informatik LS8, Baroper Str. 301, 44421 Dortmund, Germany*
- [6] Wikipedia article on Support Vector Machine, http://en.wikipedia.org/wiki/Support_vector_machine
- [7] Machine Learning, http://en.wikipedia.org/wiki/Machine_Learning
- [8] Machine Learning Course on Coursera, <https://class.coursera.org/ml-2012-002/class/index>
- [9] XML, <http://en.wikipedia.org/wiki/XML>
- [10] LIBSVM, <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>
- [11] LIBSVM, **Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin**, April 15, 2010.
- [12] A Practical Guide to Support Vector Classification, **Chih-Jen Lin**, *Department of Computer Science, National Taiwan University, Talk at University of Freiburg, July 15, 2003.*
- [13] An approach for classification of highly imbalanced data using weighting and undersampling, **Ashish Anand, Ganesan Pugalenth, Gary B. Fogel, P. N. Suganthan**, *Springer Journal.*
- [14] RBF SVM Parameters, http://scikit-learn.org/dev/auto_examples/svm/plot_rbf_parameters.html

- [15] Cross-validation (statistics), [http://en.wikipedia.org/wiki/Cross-validation_\(statistics\)](http://en.wikipedia.org/wiki/Cross-validation_(statistics))
- [16] Apache Lucene — a high-performance, full-featured text search engine library, **Jakarta, Apache**, 2004.
- [17] JFlex — the fast scanner generator for Java, **Klein, Gerwin and Rowe, Steve and Décamps, Régis**, 2001, <http://www.jflex.de/>
- [18] Unicode Standard Annex# 29, Unicode Text Segmentation, **Davis, Mark**, *Technical Report 29, September 2009*, <http://www.unicode.org/reports/tr29>
- [19] A Highly Accurate Sentence and Paragraph Breaker, **Scott Piao**, 2008, http://text0.mib.man.ac.uk:8080/scottpiao/sent_detector
- [20] Stratified Sampling, http://en.wikipedia.org/wiki/Stratified_sampling
- [21] Overfitting, <http://en.wikipedia.org/wiki/Overfitting>