A Focused Web Crawler Driven by Self-Optimizing Classifiers

Master's Thesis
Dominik Sobania

Original title: A Focused Web Crawler Driven by Self-Optimizing Classifiers
German title: Fokussiertes Webcrawling mit selbstorganisierenden Klassifizierern

Master's Thesis

Submitted by Dominik Sobania
Submission date: 09/25/2015

Supervisor: Prof. Dr. Chris Biemann
Coordinator: Steffen Remus

TU Darmstadt
Department of Computer Science,
Language Technology Group

Erklärung

Hiermit versichere ich, die vorliegende Master's Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 25. September 2015 Dominik Sobania

Zusammenfassung

Üblicherweise laden Web-Crawler alle Dokumente herunter, die von einer begrenzten Menge Seed-URLs aus erreichbar sind. Für die Generierung von Corpora ist eine Breitensuche allerdings nicht effizient, da uns hier nur ein bestimmtes Thema interessiert. Ein fokussierter Crawler besucht verlinkte Dokumente, die von einer Entscheidungsfunktion ausgewählt wurden, mit höherer Priorität. Dieser Ansatz fokussiert sich selbst auf das gesuchte Thema.

In dieser Masterarbeit beschreiben wir einen Ansatz für fokussiertes Crawling, der als erster Schritt für die Generierung von Corpora genutzt werden kann. Basierend auf einem kleinen Satz an Textdokumenten, die das gesuchte Thema definieren, erstellt eine Pipeline, bestehend aus mehreren Klassifizierern, die Trainingsdaten für einen Hyperlink-Klassifizierer – die Entscheidungsfunktion des fokussierten Crawlers. Für die Optimierung der Klassifizierer benutzen wir einen evolutionären Algorithmus für die Feature Subset Selection. Die Chromosomen des evolutionären Algorithmus basieren auf einer serialisierbaren Baumstruktur. Diese Baumstruktur enthält die Features, die zugehörigen Parameter und den Klassifikationsalgorithmus.

Der implementierte fokussierte Crawler übertrifft einen auf Breitensuche basierenden Crawler. Um dies zu beweisen untersuchen wir die gefunden Dokumente eines fokussierten Crawlers, dessen Klassifizierer darauf trainiert ist Hyperlinks zu erkennen, die zu medizinischen Dokumenten führen und vergleichen die Ergebnisse mit den gefundenen Dokumenten eines Crawlers basierend auf Breitensuche und den identischen Seed-URLs. Klassifizierer mit komplexen Features bremsen jedoch den fokussierten Crawler aus. Trotzdem haben wir ein Feature gefunden, das Einfachheit und Qualität vereint: Tokens extrahiert aus einer URL. Diese Tokens können durch das Aufteilen der URL an Punkten, Bindestrichen und Schrägstrichen extrahiert werden.

Abstract

Standard web crawlers download all web documents reachable from a set of seed URLs. A breadth-first search like this is ineffective for corpus generation because in this case we are only interested in a single topic. A focused crawler visits linked web documents, selected by a decision function, with a higher priority. So this approach focuses itself on the topic of interest.

In this thesis we introduce a focused crawling approach that can be used as a first step of corpus generation. Based on a small set of input documents, which defines the topic of interest, a classifier pipeline creates step by step the training data for a hyperlink classifier – the focused crawler's decision function. For the optimization of the classifiers we use an evolutionary algorithm for feature subset selection. Therefore, we introduce a serializable chromosome representation – a tree structure. This tree structure contains features, its parameters and the classification algorithm.

The focused crawler implemented for this thesis outperforms the breadth-first crawler. To prove that, we evaluate the crawler's output of a focused crawl with a decision function, trained to detect hyperlinks regarding the medicine topic, and compare it to the output of a breadth-first crawler starting from the same seed URLs. However, a classifier with many complex features slows down the focused crawler. Nevertheless, we have found a feature set that combines convenience and quality: tokens from an URL. These tokens are extracted by splitting the URL at dots, hyphens and slashes.

Contents

List of Figures

List of Tables

1 Introduction

Corpora, large collections of texts, are the foundation of many applications in *natural language processing* and *machine learning*. They cover e.g. a language's structure, word co-occurrences and allow to count word frequencies. Furthermore, they can be used to train and test machine learning algorithms for text classification [16].

In this thesis we use a *focused crawling* approach as a first step of corpus generation. We investigate an architecture based on a classifier pipeline, for step by step generation of training data, for a classifier that predicts the class of a hyperlink. The focused crawler's decision is based on this classifier. Thus, the user gives the system a set of documents. These document collection defines the topic of interest, e.g. a few megabytes of medicine related documents define medicine as the topic of interest. The pipeline converts these documents into a form that is suitable to train the classifier that decides if a hyperlink leads to a document fitting the topic of interest, e.g. medicine, or not.

Furthermore, we analyze an evolutionary algorithm's selection process for features and classification algorithms, which is used to optimize the pipeline's classifiers. Evolutionary algorithms can be compared to the biological evolution – survival of the fittest. In every generation, different classifiers compete against each other and the "children" of the best classifiers have a higher chance to reach the next generation [39]. Finally, we combine the trained classifier with a crawling system. Documents that belong to the topic of interest with a certain probability are crawled with a higher priority.

The term focused crawler is introduced by Chakrabarti et al. in 1999 [10]. They use a large and detailed taxonomy to train a classifier. The topic of interest is used as the positive class and all other topics in the taxonomy are used as the negative class for training. The classifier serves as a decision function during a web crawl. If the source document is positive, with respect to a certain topic, all hyperlinks of this document are crawled with a higher priority. The decision function is the main distinguishing feature of a focused crawler compared to a standard crawler. Later, in 2002, Chakrabarti et al. included features like tokens in the hyperlink's neighborhood and other features in the decision function [11].

Chakrabarti et al. use their focused crawler to cover the texts of very specific topics. This is why they need this detailed taxonomy. In difference to Chakrabarti et al. we are not interested in small and specific topics, but in a large collection of texts of a certain topic. Furthermore, we do not want to maintain such a large taxonomy.

In our focused crawling approach the topic of interest is defined by the data, added to the system by the user. The focused crawler's decision function is a *hyperlink classifier*. The crawler prioritizes the extracted hyperlinks with support of this classifier. Therefore, it sends the HTML content containing the hyperlink, the hyperlink's URL itself and the source document's URL to the hyperlink classifier. When the classifier's answer is positive, the hyperlink gets a higher priority compared to the other hyperlinks.

The hyperlink classifier is trained with positive and negative HTML data. For feature subset selection we use an *evolutionary algorithm*, which will be explained in Chapter 4. However, the training of the hyperlink classifier is a problem at this point because the user's input contains only positive data and maybe no HTML markup. Therefore, we add a *standard binary classifier* to the system.

The binary classifier is trained with positive and negative data. It supports plain text and HTML content. Again, the feature subset selection is supported by an evolutionary algorithm. The main

purpose of this classifier is to build a train and test set for the hyperlink classifier. Therefore it selects positive and negative documents containing HTML markup from a short general crawl. However, the binary classifier needs negative training data as well. Therefore, the classifier pipeline is extended by a filter function – an *unary classifier*.

The unary classifier is trained with positive data only. So it can be used to generate a training set for the binary classifier. Therefore, the unary classifier iterates over the documents of a short general crawl and if a document is not classified as positive by the unary classifier it can be added to the negative set. After that, the binary classifier can be trained with the positive user's input and the created negative set. In difference to the other classifiers, the unary classifier is not supported by an evolutionary algorithm because this algorithm would need at least negative test data which is not available at this point.

1.1 The Thesis's Structure

This thesis is subdivided into two parts. In the first part we will explain the fundamentals, everything which is important for the understanding of the system, and in the second part we will show the system's evaluation results.

Chapter 2 describes how a web crawler works and the term focused crawler is introduced and explained in detail. In Chapter 3 we will explain the fundamentals of natural language processing and machine learning. Therefore, a few algorithms will be illustrated. In addition to that, the WEKA and UIMA frameworks are described. After that, Chapter 4 explains how evolutionary algorithms work. Furthermore, it will be explained how evolutionary algorithms are used for feature subset selection in the implemented system.

The second part starts with Chapter 5. Here, the architecture of the implemented system is described in detail. Additionally, the individual classifiers are explained and we will present the used features. In Chapter 6 we will show the evaluation results of every single classifier and analyze the performance of the complete focused crawler.

2 Introduction to Web Crawling

Every major search engine we use everyday is based on a web page index. These indices are built with *web crawlers*, which are programs that walk around the *World Wide Web* by "clicking" on every hyperlink they can find and collect the information found. However, building a large web page index is not the only application for web crawlers.

2.1 Purposes of Web Crawling

Web crawlers are the basic tool for building web page indices which can be used as a foundation for the implementation of a search engine [35]. Starting from a few seed URLs the web crawler downloads every page and extracts all hyperlinks found on every visited page and continues by downloading new pages. So it is possible to find every reachable document starting from the seed URLs.

On the other hand, there are millions of pages that are not linked to other pages or secured by a password. The pages, in the so-called *deep web*, cannot be indexed by a web crawler. This poses a problem because the biggest part of the web cannot be found by information retrieval systems, which is a big loss of information [44]. Corporate intranets or even social networks – like Facebook – can be seen as part of the deep web. They form a web inside the web.

Nevertheless, web crawling is a good approach for building a web page index and it is used by all of the famous search engines like Google, Yahoo! or Bing. Even their crawlers are well known, have names of their own and can be detected by their user agent, e.g. for a website's visitor statistics. The most important ones are called Googlebot and Bingbot [58].

To perform information retrieval techniques on a web page index, a well-known data structure called *inverted index* can be used. Here, a list of words is linked to the documents containing the word. This makes AND operations on word level and full text search much easier [30].

Furthermore, web crawling can be used to increase the usability of websites. The maintenance of large websites is a complex job, especially if a lot of people work together. Large websites contain many internal and external hyperlinks, but the structure of a website is subject to constant change and the web also changes its structure every day. However, broken links are a problem on large websites because they cannot be detected manually because this takes too much time and is too expensive.

A web crawler can solve this problem [29]. It can be limited to a specific top level domain, e.g. a corporate website, and detect pages with an HTTP response code greater than or equal to 400 (e.g. 404 is the HTTP code for "page not found") [4]. After that, pages with detected errors can be repaired.

In this thesis we use an efficient type of web crawling as a first step of *corpora generation*. A corpus is a collection of texts. These corpora are useful for research in the fields of natural language processing and machine learning. The internet is a good source for corpora generation, because it contains texts of all conceivable topics. The web crawler downloads the web content and in the end the appropriate content can be combined to build a corpus.

Although the idea of web crawling seems simple, the implementation of a web crawler comes with many challenges. Figure 2.1 shows a simplified sketch of a crawling architecture, based on the book "Mining the Web" by Soumen Chakrabarti [9] that will be explained in this section.

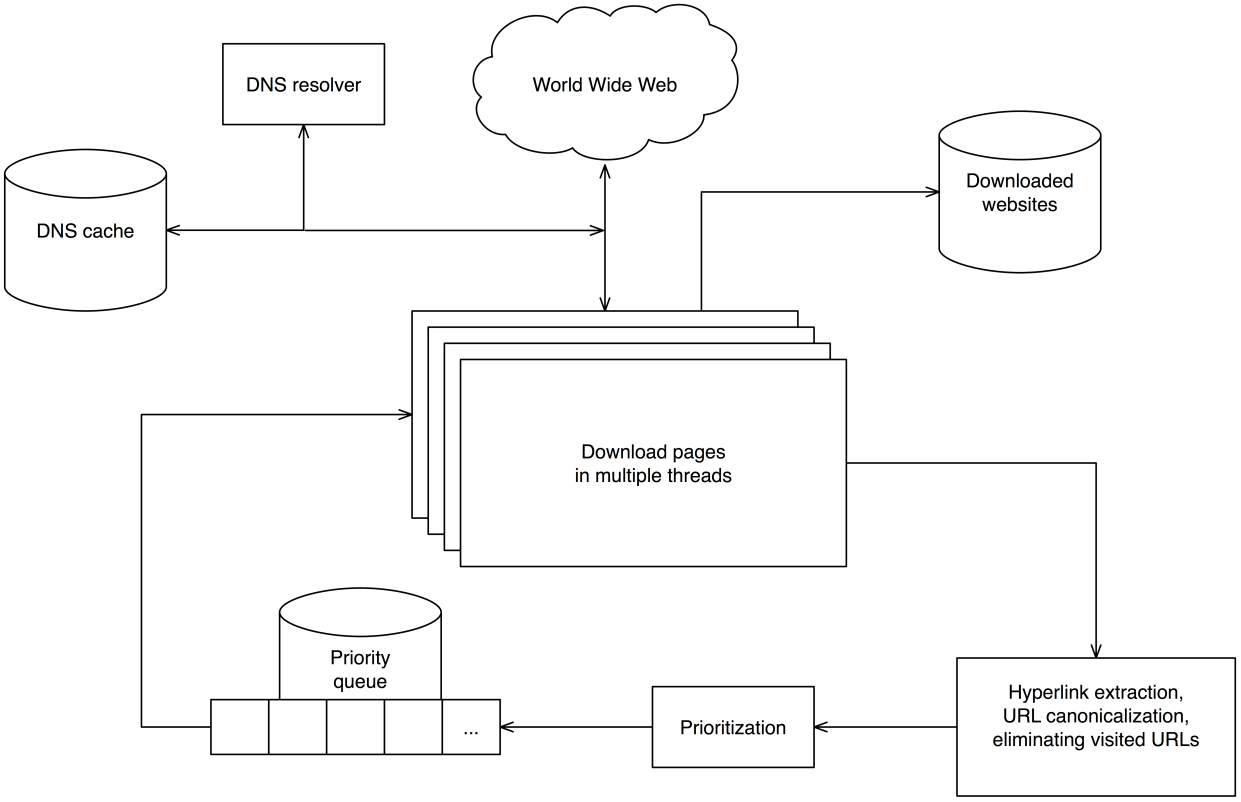

Figure 2.1: A simplified crawling architecture based on "Mining the Web" by Soumen Chakrabarti [9].

The crawling process starts with a small set of seed URLs in the priority queue. These pages are downloaded from the web in the page fetching section, illustrated in the center of Figure 2.1. This part can be realized in multiple threads to increase efficiency. The downloaded pages can be saved on the hard disk or integrated in a web page index.

The crawler needs the IP address of the server before a web document on it can be downloaded. To resolve the domain names into IP addresses it uses DNS (*Domain Name System*) [41]. The web crawler requests the IP address from a service. However, it is very inefficient to make a service call every time before downloading a document. That is why resolved names are saved in the DNS cache for a certain amount of time. If the DNS cache contains a domain's IP address it is not necessary to make an expensive request to the DNS service.

Every downloaded web document is analyzed and the containing hyperlinks are extracted. The already-visited URLs are ignored. The remaining URLs are prioritized and added to the priority queue. After that, the circle starts again until the web crawler is stopped or the priority queue is emptied.

2.2.1 URL Canonicalization

Every HTML document may contain a finite number of hyperlinks connecting to other documents. That is the reason why web crawling is possible. But for crawling it is a problem that HTML allows relative hyperlinks like `../example.html`. These links do not include protocol and domain information. Furthermore, it is possible to use absolute URLs and leave out the protocol information (e.g. `//www.example.org`). This is helpful if the website is available over HTTP and HTTPS, but a crawler cannot save an URL like this without rewriting.

To solve these problems the URL can be rewritten such that it contains all possible elements, which are the protocol definition, server location, port, directory, file and the query parameters, even if they are quite obvious. The following example shows an URL with all possible elements [5]:

`https://www.example.org:80/directory/index.html`

It is completely written in lowercase and it also contains the port number. Now it can be compared to other URLs to recognize duplicates, which will be explained in Section 2.2.2.

2.2.2 Eliminating Visited URLs and Detection of Duplicates

Popular web documents are linked by various websites, but it is inefficient to download these documents more than once. In addition to that, we do not want duplicates in our output. Thus, in a professional web crawler already-visited URLs are filtered. Therefore, the URLs must be canonicalized, as explained in Section 2.2.1. Otherwise a comparison is impossible.

Before downloading a web document, its URL can be hashed and compared to already-visited URLs saved in a hash table. If the hash table contains the URL it will be skipped. On the other hand, it is not only interesting to skip already downloaded documents, but also documents that are similar to downloaded documents.

Even for building corpora based on a web crawl it is useful to skip documents with similar or copied content, like Wikipedia articles copied to gain content for a website. But how it is possible to detect similar documents?

To detect similar documents we cannot simply compare the texts of two documents character by character because even a small modification is a difference. However, with an approach called *shingling* [7] it is possible to detect near duplicates (not used in the implemented system but could be used on the data in a post-processing step). We interpret the text as sequences of n words, known as n-grams, e.g. with $n = 8$. After that, we can compute the Jaccard coefficient with the following formula:

$$J(doc_1, doc_2) = \frac{|S(doc_1) \cap S(doc_2)|}{|S(doc_1) \cup S(doc_2)|}.$$

It is the ratio of the n-gram's intersection and union. A higher value indicates a higher similarity. With a threshold it is possible to sort near duplicates out.

2.2.3 Politeness and the Robots.txt

When a crawl is running, it is important to be polite. That means the crawler should not send too many requests to the same server in a short period of time. The crawler has to use mechanisms to avoid DoS

attacks (*Denial of Service*) [56]. To solve this problem, it is a possible approach to provide a separate queue for every server to assert a break of a few seconds between the requests to the same server.

Another technique belonging to politeness is the *robots.txt* (*robots exclusion standard*) [32]. The robots.txt is written by the server's administrator and is located in the domain's root directory. It defines rules for web crawlers and allows or disallows to visit parts of the website or certain documents. The following list shows a few example rules:

```
User-agent: *
Disallow: /

User-agent: Googlebot
Disallow: /hidden

User-agent: *
Allow: /dir/example.html
Disallow: /dir
```

Web crawlers are matched by their user agent. The first example disallows the whole page to every crawler. The second example disallows the access to one folder, but this rule is only active for Googlebot. The last example disallows the whole directory `dir` except the file `example.html`.

The consideration of the robots.txt is optional for every web crawler. That means there is no official law but it is advisable for web crawlers to respect this standard.

2.3 Comparison of Standard and Focused Crawlers

In the next paragraphs we will introduce the definition of a *focused crawler* and compare it to a standard crawler. When we use the term standard crawler we mean the crawler sketched in Section 2.2. It starts with a few seed URLs and visits all hyperlinks in the downloaded documents. Figure 2.2 shows a standard crawler with one seed URL (breadth-first).

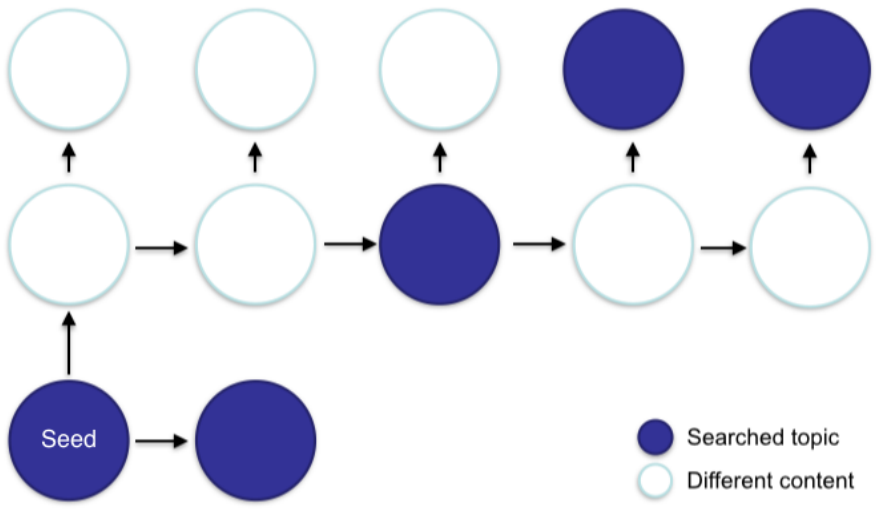

Figure 2.2: A simple standard crawler with one seed URL (the blue circle on the left).

The picture illustrates a standard crawler's performance for corpus generation. The blue circles represent documents belonging to the topic of interest. The white circles represent negative documents. The example crawl starts with one positive seed (the blue circle on the left) and visits all reachable documents.

This is good for building a general web page index, as explained in Section 2.1, but quite inefficient if we are only interested in a single topic. As we can see, the crawl contains a lot of negative documents. Downloading negative documents takes much time and the crawler expands into uninteresting web areas.

A focused crawler tries to focus itself on the topic of interest. It follows hyperlinks only if they are evaluated as positive by a *decision function*. Figure 2.3 illustrates the results of an optimal focused crawler.

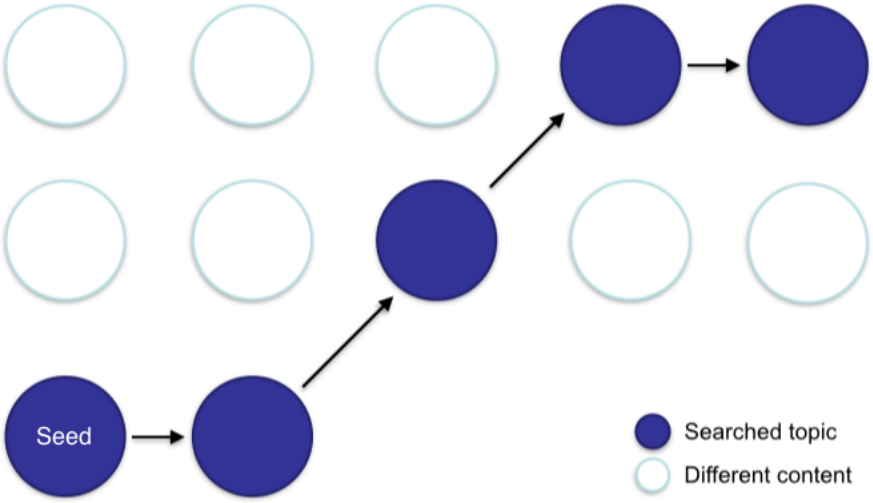

Figure 2.3: An optimal focused crawler with one seed URL (the blue circle on the left).

Again, it starts with one positive seed URL (on the left side), but in contrast to the standard crawler only positive documents are downloaded. However, an optimal result like this is only possible with a simple decision function, like only following URLs from a specific domain, e.g. `*.com`, `*.co.uk` or `*.de`. When we are e.g. interested in documents regarding to a specific topic, as in this thesis, the decision function is much more complex, because we need a classifier which decides to follow an URL or not [51].

2.4 Framework: Heritrix

There are a lot of open source web crawlers available on the web which can be used and extended for one's own crawling purposes and experiments – *Heritrix*[1] is a powerful example. It is developed and used by the Internet Archive[2] for their archiving work [24].

The architecture follows the principles explained in Section 2.2. Furthermore, the crawler provides a *REST* interface and a graphical user interface for easy interaction and integration.

Internally, Heritrix is divided into three processing chains [54], which contain the crawler's processors. The processors are responsible for the most of the crawler's tasks. The following list describes the three processing chains:

- Candidate chain: The processors of this chain are executed for URLs before they are added to the queue and before downloading.
- Fetch chain: This chain's processors are executed for URLs when they are downloaded – e.g. different extractors for hyperlinks in HTML, CSS or JavaScript [55].
- Disposition chain: The processors of this chain are executed for URLs when the other chains are ready – e.g. statistics or the generation of the output files (WARC) [53].

The chains are always executed in the previously described order. The order of the processors's execution inside a chain is defined, and can be edited, in the `crawler-beans.cxml`. Additionally, that is the place where custom processors, e.g. for changed hyperlink prioritizing or filtering, can be added.

[1] `https://webarchive.jira.com/wiki/display/Heritrix/Heritrix` (checked 08/16/2015)
[2] Internet Archive: `https://archive.org/index.php` (checked 08/16/2015)

3 Short Introduction to Natural Language Processing and Machine Learning

Products and software based on the techniques of natural language processing (NLP) become more and more a part of our everyday life, and there are many more than Siri, Google Now or Microsoft's Cortana. In this chapter we give an introduction to natural language processing and machine learning.

3.1 Natural Language Processing and its Applications

The origins of computer-based natural language processing lie in the 1950s. The *turing test* is a well-known example of a test for artificial intelligence of a computer program. In this test, a person has to decide if he/she is talking with a human being sitting in front of a different terminal or a computer program [57]. This test is still relevant today, e.g. the *Loebner prize* rewards chatterbots for a human-like communication [48].

Another famous example for early NLP is *ELIZA* by Joseph Weizenbaum. ELIZA is a simple chatterbot simulating a psychotherapist. It is based on keyword matching and rule-based content transformation. However, the key feature of ELIZA: It keeps the user (the patient) going on with the communication. Therefore, the program asks questions and reformulates the user's input. The following shortened example is taken from the article introducing ELIZA by Joseph Weizenbaum (1966) [60]:

```
User:  Men are all alike.
ELIZA: IN WHAT WAY
User:  They're always bugging us about something or other.
ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE
User:  Well, my boyfriend made me come here.
ELIZA: YOUR BOYFRIEND MADE YOU COME HERE
```

A state-of-the-art example is *Watson*, developed by IBM. Its basic idea is to provide a semantic search engine which improves the results compared to todays keyword-based systems. It became known to a wider public in 2011 when the system participated in the game show Jeopardy. Watson played against two of the most successful players of the show and won with advance [36]. This victory can be compared to the chess game between Garri Kasparow, the former chess world champion, and Deep Blue, a system developed by IBM, in 1997 [23]. Both systems were successful in a field which was dominated by human intelligence.

Today, Watson is used for various applications like healthcare or business decision systems [43] – e.g. the Memorial Sloan Kettering Cancer Centre[1] used Watson to calculate a person's risk of lung and breast cancer [1]. Furthermore, IBM provides with the *AlchemyAPI*[2] a cloud-based service for a lot of natural language processing tasks, like translation or question answering [22].

Many researched tasks in the field of natural language processing are sub-tasks which can be combined to large applications like machine translation or automatic summarization systems. These sub-tasks are e.g. word segmentation, part-of-speech tagging, parsing or named entity recognition [25].

[1] https://www.mskcc.org (checked 08/19/2015)

[2] `http://www.alchemyapi.com` (checked 09/20/2015)

3.2 A Short Introduction to Machine Learning

Machine Learning is one of the most powerful tools in computing of the last decades and became more popular in the recent years. In this thesis the focus is on a subarea of machine learning – *supervised text classification*. Supervised classification means that the complete training data is labeled [40]. In the following sections we will explain the foundations of the main algorithms used in this thesis.

3.2.1 Instance-Based Learning

One of the simplest learning approaches is instance-based learning. Here, no real model is learned, the labeled training examples are simply stored. A basic instance-based algorithm is the *Rote learner*: It classifies only previously seen examples. This leads to a high accuracy for examples seen by the algorithm but it is unusable for a system when there are a lot of new examples to classify.

A well-known instance-based learning algorithm is the *k-nearest neighbors algorithm*. As mentioned before, the training examples are simply stored. To classify an unseen example the algorithm has to calculate the distance to every training example. The unseen example's class corresponds to the majority class of the k nearest neighbors. Figure 3.1 shows an example of the k-NN.

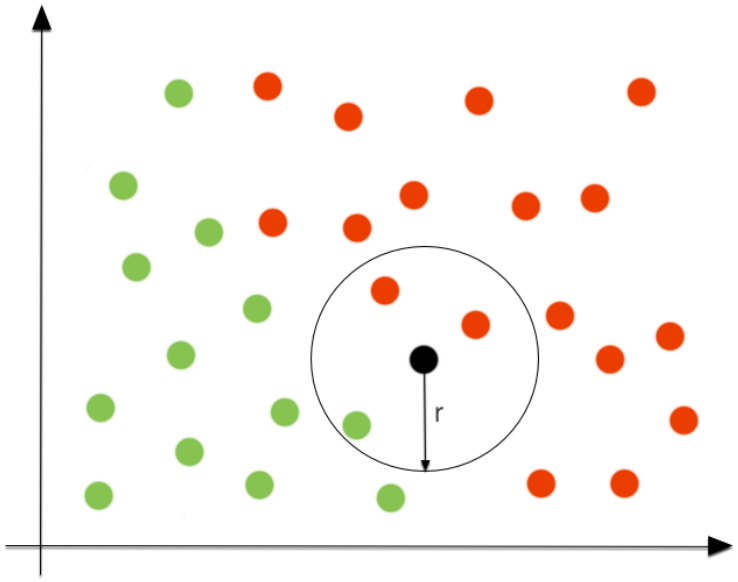

Figure 3.1: An example for the k-NN with an unlabeled example (black circle).

In this example $k = 3$. The nearest neighbors of the unseen example are two red examples and one green example. So the unseen example is classified as red. The choice of the value k is very important because a small value leads to a classifier which is susceptible to noise in the data and with a large value it is possible that too many other classes affect the classification result [40].

A key factor of every k-NN algorithm is the distance function. A commonly used distance function is the *Euclidean distance*. Another distance function is the *Manhattan distance* [59]: It is the sum over all attribute distances. Lastly, we will explain the *cosine similarity* [20]:

$$CosineSimilarity(A,B) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}}.$$

It is the angle's cosine value of the two vectors representing the two instances. The cosine similarity is a value between 0 and 1 for vectors containing word frequencies.

3.2.2 Decision Tree Learning

Decision tree learning is an approach which learns an abstract but human-readable data structure from the training data – a simple *decision tree*. Figure 3.2 shows a simple decision tree that could be used for trip planning. Every node represents a decision required for the final classification result. The classes are represented by the tree's leaves. In Figure 3.2, the resulting class is positive if it is weekend and the weather is good.

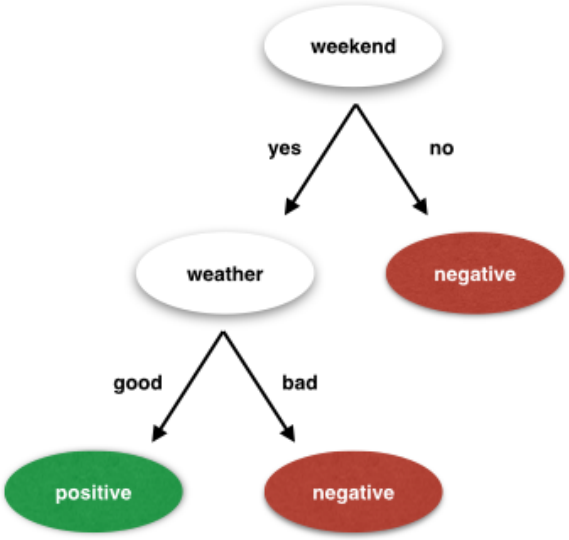

Figure 3.2: A simple decision tree for trip planning.

Top-down induction of decision trees is defined recursively. At the beginning, we have a set of training examples and a set of possible features. The purpose of a decision tree is to order the examples and the algorithm should start with the best ordering features to reach a fast convergence. So a decision tree algorithm has its own feature subset selection.

However, to find the best ordering features we have to rank all available features. Therefore, we can use an entropy-based (*entropy* is a measure for unorderness, e.g. the entropy is one if the classes have the same distribution) measure like the *information gain* based on the input training set S and a single feature F:

$$InformationGain(S,F) = Entropy(S) - AverageEntropy(S,F).$$

The average entropy is defined by the following formula:

$$AverageEntropy(S,F) = \sum_i \frac{|S_i|}{|S|} \cdot Entropy(S_i).$$

Lastly, the entropy for two classes is defined by the following formula:

$$Entropy(S) = -p_{pos} \cdot \log_2(p_{pos}) - p_{neg} \cdot \log_2(p_{neg}).$$

The information gain must be calculated for every possible feature or attribute. The attribute with the highest value will be selected. The information gain is maximized by minimizing the average entropy. This may be critical, e.g. when there are attributes with a lot of possible values, like IDs or timestamps, because this leads to a high average entropy. So, information gain has a bias towards multivalued attributes. This can be corrected by an appropriate pre-processing or with a different metric, like *gain ratio* or *gini index* [40].

3.2.3 Support Vector Machines

Support vector machines [14] are the most recent but most of the time also the most successful learning algorithms mentioned in this chapter.

A support vector machine tries to find a good *hyperplane* [6] which separates the training examples into two classes. A hyperplane is a geometric construct that is one dimension below the dimension of the space around it. In the two-dimensional example, illustrated in Figure 3.3, the hyperplane is a line. Furthermore, the selected hyperplane is the one with the largest margin between the classes on both sides of the hyperplane. The hyperplane with the maximum margin is defined by so-called *support vectors*, which are the points on the margin lines. In Figure 3.3 the support vectors are the two red circles on the margin line on the right side and the two green circles on the left side.

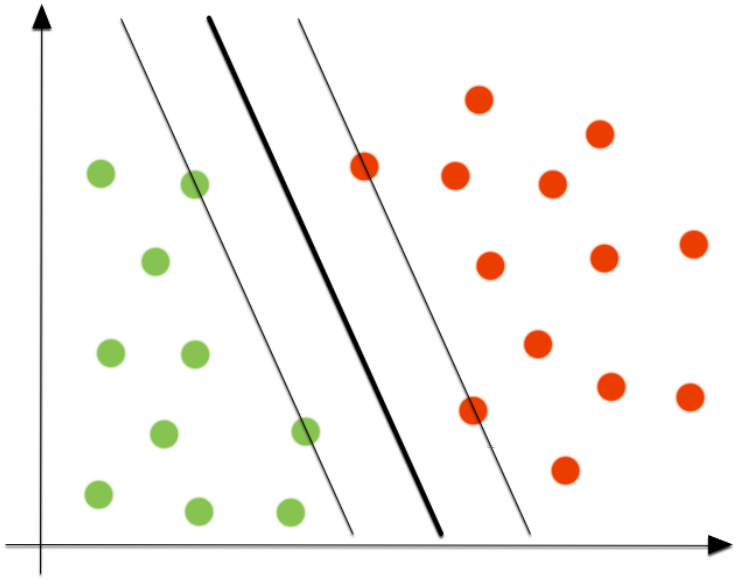

Figure 3.3: An example for the splitting of an SVM (the bold black line represents the hyperplane).

Certainly, it is not always possible to separate the classes linearly. Therefore, the SVM transforms the space into a much higher dimensional space such that a separation is possible.

3.3 Frameworks

In everyday software development stability and the development time are very important. To reduce the required time and reach the stability goal we can use well-tested frameworks. In this section we will describe the two essential frameworks used in the software implemented for this thesis.

3.3.1 UIMA

The *Unstructured Information Management Architecture*[3] (UIMA) is a tool usable for many NLP problems. Now, it is an Apache project but originally it was developed by IBM. One of the key concepts is its pipeline based workflow [13].

The internet and many data collections provide a lot of unstructured data, e.g. data that is not available in relational databases or in an XML format. With UIMA it is possible to enrich the unstructured data step-by-step with annotations in a pipeline. Therefore, a simple building block called *analysis engine* can be used. Every analysis engine should encapsulate a single task, like tokenization or part-of-speech tagging. Furthermore, subsequent annotators can be based on previous annotations, e.g. POS tagging can be based on tokenization.

Every document and its annotations are stored in the *common analysis structure* (CAS) which could be visualized as a container for all the information. Furthermore, a simple pipeline contains not only annotation engines but also a *collection reader*. It is used to read e.g. text collections and write it to the common analysis structure for further processing in the pipeline.

In the software implemented for this thesis we use UIMA to annotate e.g. part-of-speech tags. Furthermore, every frequency measuring unit and every feature extractor is encapsulated in an analysis engine. The frequency measuring units count e.g. the frequency of every token in the training data. The feature extractors use the frequency information to generate the feature sets for every document. This information can be used to generate the train and test data for the machine learning framework, introduced in Section 3.3.2.

3.3.2 WEKA

The *Waikato Environment for Knowledge Analysis*[4] (WEKA) is a machine learning framework developed by the University of Waikato in Hamilton, New Zealand [19]. It contains a large collection of preprocessing tools, classification, regression and clustering algorithms. Furthermore, WEKA can be controlled by a graphical user interface and programmatically in one's own source code.

WEKA introduces the *attribute relation file format* (ARFF) to represent the instances of a train or test data collection in a text file. It is divided into two parts: A section for the header information and a section for the data. The header defines the attributes and data types. The data section contains simple lists of the instances's attribute values including the class label.

In the software implemented for this thesis we use some of WEKA's classification algorithms. This is explained in detail in Section 5.3.2.

[3] `https://uima.apache.org` (checked 09/20/2015)
[4] `http://www.cs.waikato.ac.nz/ml/weka/` (checked 09/20/2015)

4 Introduction to Evolutionary Algorithms

Using *evolutionary algorithms* is one possible approach to find good solutions for problems where an exact solution is hard to find in practice, e.g. because of the algorithm's runtime. In this chapter a basic evolutionary algorithm and its adjustment for feature subset selection is described.

4.1 An Approach Inspired by Biology

A prime example for an observable evolution in a very short time is the *peppered moth* (lat. *biston betularia*). The peppered moth lives mainly in mixed forests where we can also find a lot of birch-trees. Furthermore this insect exists in two different main colors – black and white – and that is the connecting factor to an effect called *industrial melanism*, but below we will call it *evolution*.

Figure 4.1: A white example of the peppered moth [42].

Today, about 95 percent of these moths are white and 5 percent are black. In the early 19th century the distribution was more or less the same. But in the mid-19th century when the industrial revolution was on its peak in England it influenced not only the populations everyday life, but also the environment and animals including the peppered moth. In this time there was a lot of pollution in the air and the trees, especially the white parts of the birch-trees, got darker. This effect destroyed the camouflage of the white ones because of the high contrast and in the following years the mentioned distribution of white and black moths switched, because the white ones were easier to catch by their natural enemies.

In the mid-20th century the distribution switched again because of new environmental laws. The air got cleaner and the trees recovered from the pollution. And the most impressive fact is that: Everything happened in less than 100 years [34].

4.2 Evolutionary Algorithm

In the following sections a basic evolutionary algorithm is explained step by step. At the end of each step the implementation of the evolutionary algorithm for feature subset selection, written for this thesis, is illustrated.

4.2.1 A Rough Sketch of the Algorithm

There are a lot of different versions of evolutionary algorithms. Every sub-function of such an algorithm may differ a little bit in the multitude of applications depending on the application's domain. So developers have many options for fine-tuning.

The following algorithm shows a basic version of an evolutionary algorithm [40] to give a first overview and to make it easier to understand the explanation in the following sections.

1. Generate a random sample of *chromosomes*.
2. Calculate the fitness for every chromosome with some heuristic.
3. Select chromosomes for the next generation.
    a) Chromosomes with a higher fitness value have a higher probability to be selected.
4. Perform *crossover* and *mutation* on the chromosomes.
5. If the termination condition is not fulfilled goto Step 2.

4.2.2 About Chromosomes and Generations

In case of evolutionary algorithms the terminology is influenced and adapted from biology. We have a limited set of chromosomes (also called individuals) that exist within a generation. Furthermore the chromosomes consist of a set of *genes* which carry parts of the information.

Every chromosome contains a complete description of a possible solution for the considered problem. This may be e.g. a simple string or tree representation.

Consider the famous *knapsack problem*: A knapsack, or most of the time a container in real-world problems, has a weight limit. Furthermore, a set of items is given. Every item has a weight and a certain value (e.g. 3 kilograms and 200 euros). The challenge is it to maximize the value in the knapsack without violating the weight limit [37].

The knapsack problem is an *NP-hard* (Non-deterministic Polynomial-time hard) optimization problem and part of Richard Karp's list of 21 NP-complete problems [26]. That means, that there is no known algorithm, which can find an exact solution in *polynomial time*, because this would be the proof that $P = NP$. For most applications of this problem, it is sufficient to have a very good approximation – there is no need for the exact solution – and evolutionary algorithms can be used to find an approximation for the knapsack problem.

The design of a chromosome is an important part in the design process of an evolutionary algorithm. As mentioned above a chromosome contains a possible problem solution. In case of the knapsack problem the data structure must contain every item with its value. The items are called genes. The knapsack is represented by the chromosome itself, it is the container for the items. It is possible to use a string representation here.

Imagine a knapsack with a weight limit of 5 kilograms and the following items:

- Laptop, 2 kilograms, 1500 euros
- Tablet computer, 0.7 kilograms, 800 euros
- Mobile phone, 0.2 kilograms, 300 euros
- Camera, 1.2 kilograms, 700 euros
- Telephoto lens, 0.9 kilogram, 400 euros
- Books, 2.5 kilograms, 120 euros
- Office supplies, 0.2 kilograms, 25 euros

It is not possible to put all items in the knapsack, so a selection is needed. The string representation of a chromosome could look like the following line:

- `Laptop;2;1500|Books;2.5;120|OfficeSupplies;0.2;25`

This is not the best solution but it is a valid candidate and a possible chromosome, which can be improved during the evolution. The data structure for a chromosome for feature subset selection [52] is more complex than the one used in the knapsack example. It is illustrated in Figure 4.2:

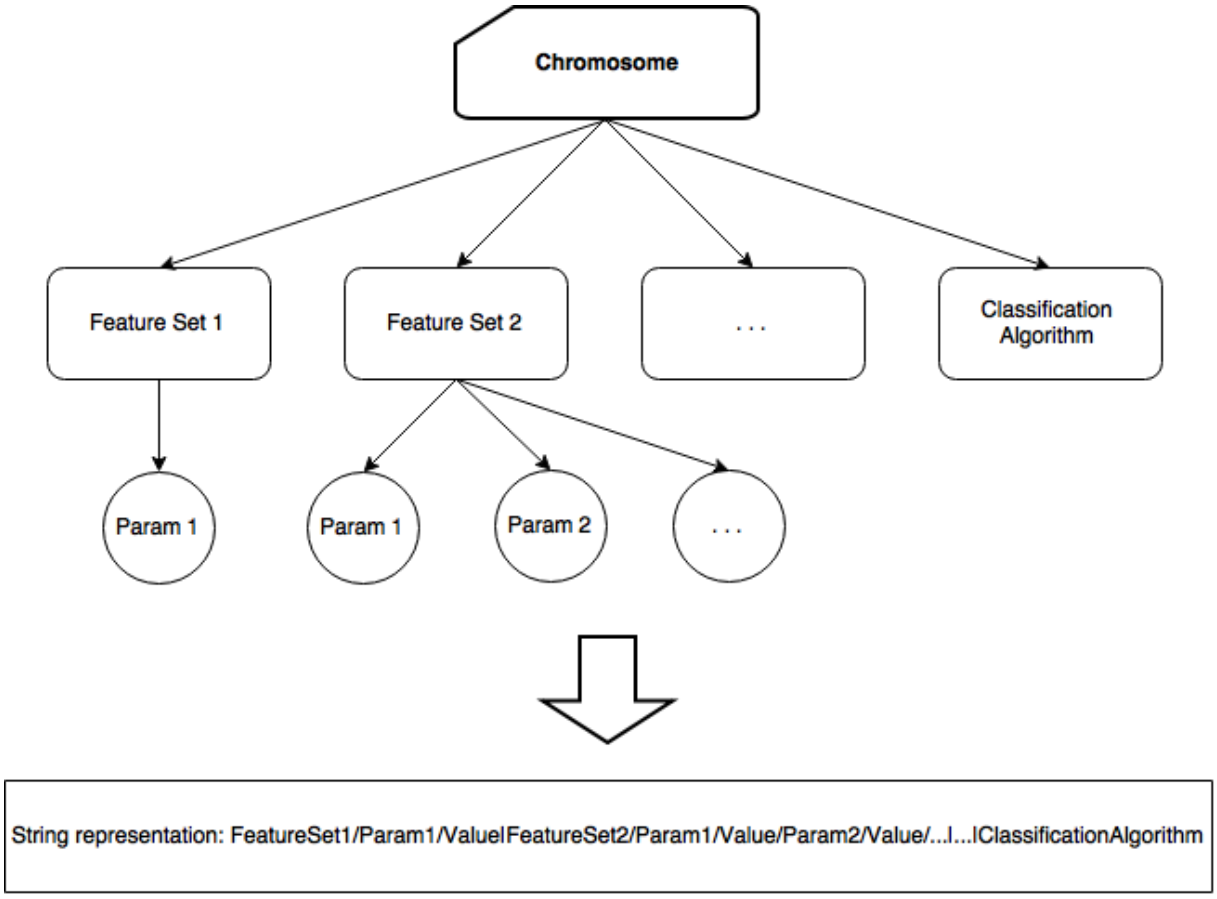

Figure 4.2: General structure of a chromosome for feature subset selection.

There are a lot of different feature sets (like *n-grams*, *POS tags*, etc.) to build a feature subset for classification algorithms. Furthermore, feature sets have properties, e.g. the *n* in n-gram. So the chromosome has to cover a finite number of feature sets, their parameters and the classification algorithm.

In the picture above the tree structure is illustrated. The chromosome represents the whole feature subset and each gene is a feature set except the last gene at the end because it covers the used classification algorithm. For an easier handling, the tree structure is serialized into a string as in Figure 4.2 and analogously to the knapsack example. Again, the chromosome contains exactly one possible solution.

An evolutionary algorithm improves the solution in its generations. A generation consists of a finite number *n* of chromosomes. The chromosomes in the first generation are generated randomly. In case of feature subset selection, first, a random number in the range from one to the number of possible feature sets is generated to determine the number of feature sets for every single chromosome in the first generation. After that, the chromosome with the random number of feature sets and a random classification algorithm is generated. Furthermore, the parameters of the feature sets are also generated randomly but here the ranges are defined manually (e.g. 1 to 5 for the *n* in n-gram).

Sometimes the previously described approach leads to a lot of chromosomes with a high number of feature sets in the first generation. To prevent that and to guarantee a good mixture of the number of feature sets per chromosome, the maximum value for the random process increases from two to the number of possible feature sets over the number of chromosomes in the first generation. The result of this strategy is that the chromosomes, which are generated first, have less feature sets.

After the first generation is generated, every chromosome is assigned a *fitness value*, which will be explained in the next section, calculated by a specific function. The fitness value is important for the selection process because chromosomes with a higher value have a higher probability to reach the next generation. The selection process chooses *n* chromosomes, because every generation has the same size, and it is possible that a chromosome is selected more than once for the next generation.

The calculation of the fitness value and the selection process are repeated in every generation as described in the algorithm overview in Section 4.2.1. After that, some functions called mutation and crossover, explained later in Section 4.2.4, are executed to bring some change into the population.

Further important facts are the number of chromosomes per generation and the number of generations. For many problems, values from 100 to 1000 for the number of chromosomes per generation and 50 to 500 for the number of generations are typical values in literature [33, 17]. These values could be used also for the knapsack problem, but for feature subset selection they had to be reduced because of computational issues regarding the applied fitness function. Here, 32 generations and values from 20 to 30 for the number of chromosomes per generation are used. The selection of these values will be motivated in more detail in the next chapters. There are also various possible termination conditions. The desired effect when using evolutionary algorithms is that the chromosomes get better during the algorithm's runtime, but it is also possible that it reaches the point that there is no further improvement. So it is also an idea to stop the algorithm after that point.

4.2.3 Fitness Function

The fitness function determines a fitness value for each chromosome in a generation. As mentioned before, chromosomes with a higher fitness value have a higher probability to reach the next generation. As an illustrative example, remember the peppered moth: The fitness value here is their ability to hide.

If these insects can be seen easily then they are a simple target for enemies and so there will be no offspring to reach the next generation – a prime example of *natural selection*.

A fitness function for the knapsack problem could be the sum of all values in a valid knapsack (or in a chromosome). If there is a better fitness value, then this solution is better for the knapsack problem.

For feature subset selection there are some candidates for the fitness function: accuracy, precision, recall and f-measure. For every function a complete training and evaluation process is required. In the system implemented for this thesis, the F-measure is used because of its combination of precision and recall:

$$F_\beta = (1+\beta^2) * \frac{precision * recall}{(\beta^2 * precision) + recall}.$$

We set $\beta = 1$ in the implemented system. A big advantage of this formula is that its focus can be moved towards recall or precision. Higher values for β puts the focus on recall and lower values on precision [18].

4.2.4 Crossover and Mutation

Chromosomes which are selected for the next generation, are candidates for crossover and mutation with a certain probability. These functions are the essential part of all evolutionary algorithms because they make the evolution possible.

First, let us have a look on the crossover function. There are some possibilities to implement this function, e.g. the *one-point crossover* [47], which is used in the implemented system and illustrated in Figure 4.3.

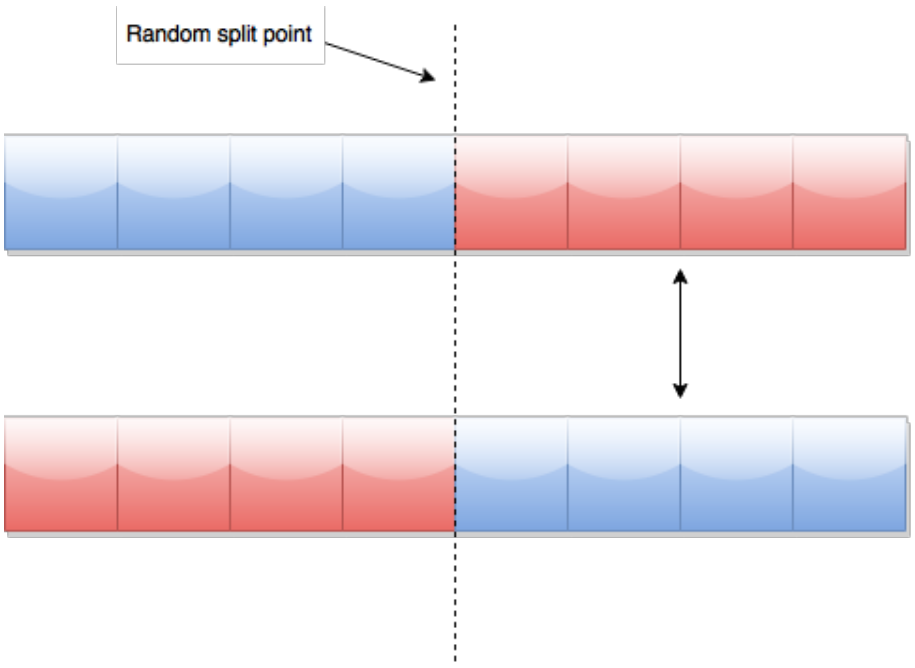

Figure 4.3: One-point crossover.

For two chromosomes a random split point is selected and the genes on the right or left side of the split point are exchanged between the chromosomes. It is also conceivable to perform the exchange of the genes in a diagonal way if the gene order is important. But for feature subset selection a diagonal exchange is not possible because the classification algorithm must be at the end of the chromosome (as described in Figure 4.2).

Another crossover function is the *n-point crossover* (illustrated in Figure 4.4) which is similar to the one-point crossover. The difference is that there are multiple split points.

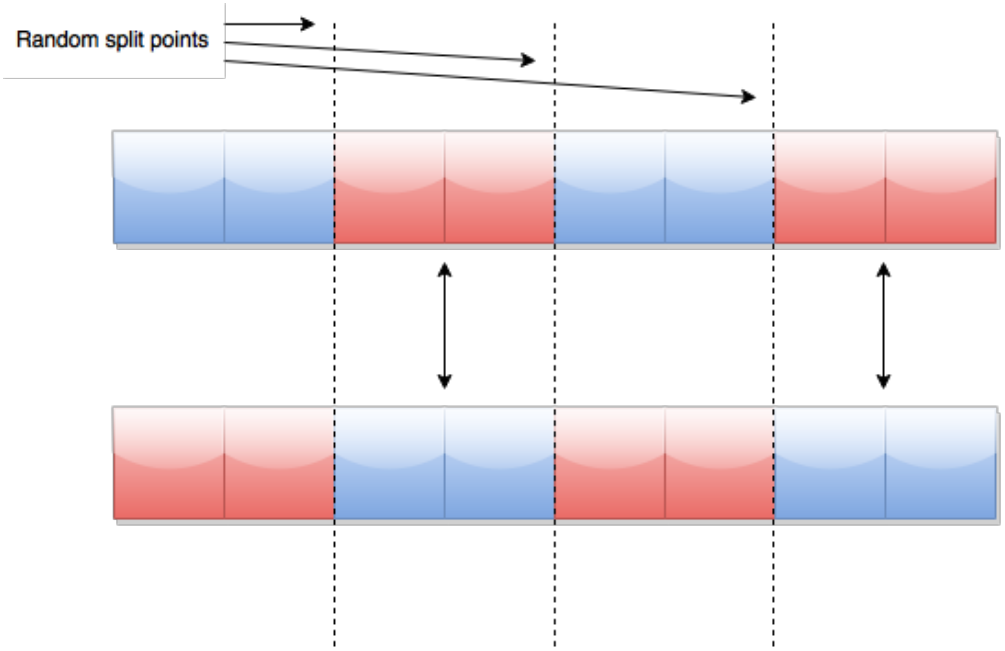

Figure 4.4: N-point crossover.

The mutation function is executed after the crossover. It iterates over all chromosomes and in each chromosome over all genes. With a certain probability a gene is exchanged or deleted such that the chromosome is still valid.

Typical values for the crossover probability are 0.6 to 0.7 and 0.001 to 0.01 for mutation probability. In the implemented system 0.6 is used for crossover probability and values from 0.01 to 0.03 for the mutation probability [46]. The mutation probability is higher just to compensate that there are less generations and chromosomes per generation. Results for these values will be shown in the following chapters [39].

Finally, evolutionary algorithms are a good approach for optimization problems, e.g. if it is to expensive to use an exact algorithm. Nevertheless, just like the biological evolution in nature, evolutionary algorithms are a bit opaque. Sometimes it is surprising that these algorithms work well for a problem. It needs some experience and a few experiments to find good values for the crossover probability, the mutation probability and the number of chromosomes per generation.

5 System Architecture

The architecture of a system is one of the major decisions in software development. It is, amongst other things, responsible for a fast implementation and a good interaction with other software components.

This chapter starts with a general overview and proceeds with a deep dive into the single components of the classification system and the focused crawler.

5.1 The Focused Crawler's Task

To understand the focused crawler's architecture it is essential to understand its purpose for this thesis. The purpose of the system is illustrated in Figure 5.1.

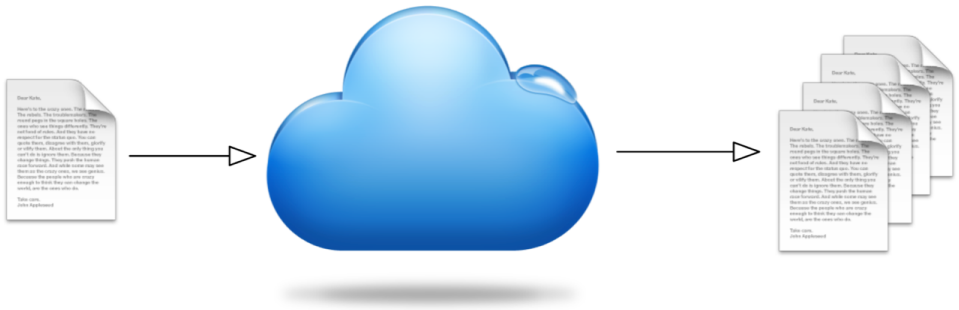

Figure 5.1: An abstract view on the system's architecture: We have a small set of input documents and after the focused crawl a large collection of documents in the output.

From a small set of input data, which contains just a few megabytes of text from a single domain, a large set of output data should be generated. To reach this goal, an extended focused crawler is used with some fundamental changes compared to the focused crawler introduced by Chakrabarti et al. in 1999 [10].

As explained in Chapter 1, Chakrabarti et al. are using a large taxonomy as training data for their focused crawler. The searched topic is the positive class and all other topics in the taxonomy are used as the negative class. A major problem is that it is hard to build and maintain such a large taxonomy. Furthermore, an architecture as illustrated in Figure 5.1 would be impossible with this approach, because it does not work when there is only positive input data ("positive" means data from the domain of interest here). In addition to that, Chakrabarti et al. are using their crawler not only to generate large amounts of data from the same domain, but also for collecting very special topics which are not covered very well in the World Wide Web. So we have to find another way to train a classifier for our focused crawler with only a positive input set.

The approach used in the implemented focused crawler is to build a negative training set for a binary classifier at runtime without any user interaction. With the trained binary classifier it is possible to build the train and test data for a hyperlink classifier which decides if a hyperlink leads to a positive (in-domain) or a negative (out-of-domain) website. To reach this goal with low user interaction, a classifier pipeline is used which is explained in Section 5.3.

5.2 Introduction to REST Services

The focused crawler's architecture makes use of the *Representational State Transfer* (REST) paradigm over HTTP. This is an architectural style, introduced in 2000 by Roy Fielding in his doctoral thesis [15], based on the foundations of the World Wide Web, which allows to communicate over HTTP's methods (e.g. GET, POST etc.) [4] and other already defined and common standards.

The basic idea of REST is it to adapt the human way to interact with the web and its services. The only software needed is a simple web browser which supports HTTP, HTML and a few further standards. For the user there is no difference in doing online banking or visiting a web encyclopedia. And REST is designed for machine-to-machine communication.

The most essential facts of REST are summarized in the following list:

- REST works with *resources*, an abstraction layer which encapsulates a service or an entity, and every resource has at least one representation.
  - A REST service can provide the most appropriate file type, e.g. plain text, XML or JSON, as representation.
- Every service has a unique address, URI (Uniform Resource Identifier).
- A REST service is stateless, which means that a client's request includes all required data (a big plus for scalability).
  - But it is also possible that a resource changes its state because it is triggered by a client's request.
- The range of methods is limited. When HTTP is used there are e.g. GET, POST, PUT and DELETE, but this is enough to implement commonly used CRUD systems (Create, Read, Update and Delete) [3].
- A REST service should be "discoverable". That means that it is sufficient to know the URI of the service's starting point, every other resource should be reachable over hyperlinks or a similar concept.

Fielding describes this architectural style in his thesis from a very abstract perspective. So, from concept it is possible to use other protocols than HTTP, which fulfill the mentioned requirements, but in practice mostly HTTP is used today.

5.3 Main Components

As described in Section 5.1, the system should crawl a large amount of documents of a single topic, defined by the user's small set of input data, with minimal user interaction. Therefore, the web crawler visits hyperlinks, which lead to documents of the searched topic with a high probability, with a higher priority. So the web crawler is focused to the topic of interest and the amount of downloaded irrelevant data is reduced.

To make the goal of a minimal required user interaction possible and to support the web crawler with hyperlink predictions, the previously mentioned classifier pipeline is used. The pipeline's component diagram is illustrated in Figure 5.2.

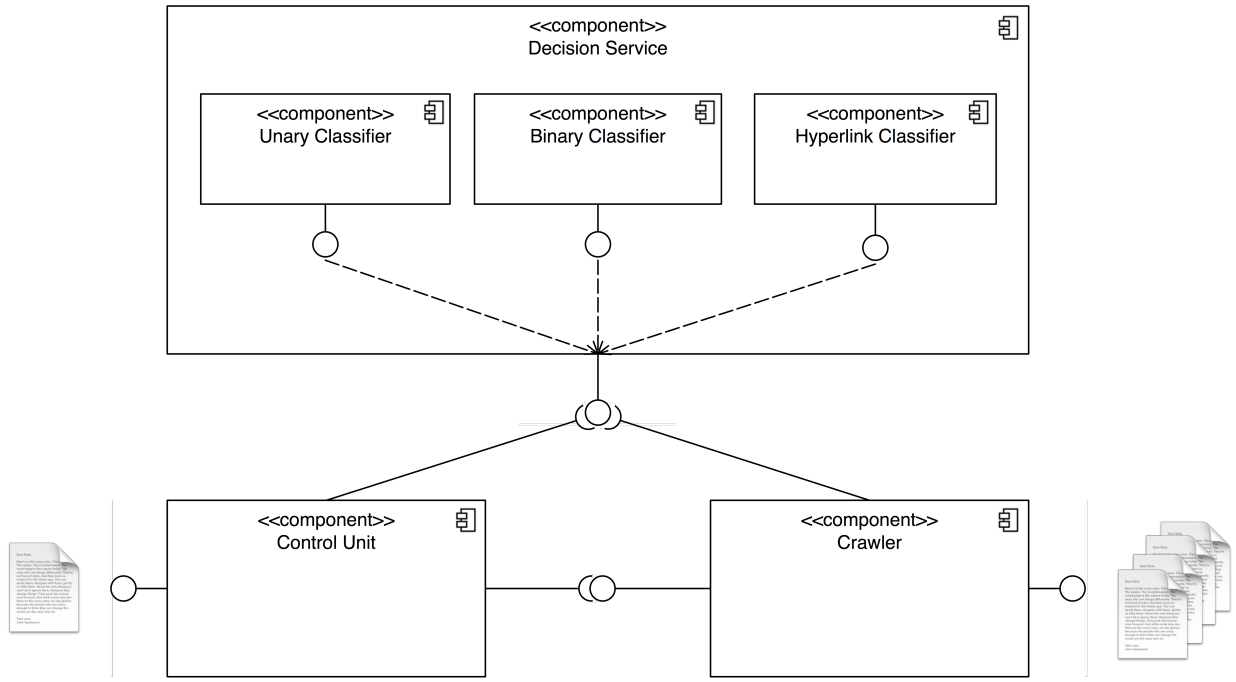

Figure 5.2: A component diagram of the focused crawler.

The most important part of this pipeline is the *hyperlink classifier*. Given an HTML document with the marked hyperlink of interest, it decides if the hyperlink leads to a positive or a negative website, with respect to the searched topic. The classifier is trained with a positive and a negative set of labeled HTML data. This is a problem because the user's input contains only positive data and it is also possible that the user's input consists only of plain text. So the system needs a mechanism to create a train set for the hyperlink classifier.

Therefore, the pipeline is extended with a *binary classifier*. It handles both, plain text and HTML documents. So it can be used by a mechanism which builds a train set for the hyperlink classifier, e.g. with documents found by a short general crawl, e.g. with a standard breadth or depth first web crawling strategy. The binary classifier gets the target document of a hyperlink and assigns the class used for labeling the hyperlink in the train set. Just as the hyperlink classifier, the binary classifier is trained with positive and negative examples. So the previously mentioned problem is only solved in part. It is possible to handle HTML and plain text input now, but the binary classifier also needs negative examples for training.

That is why the pipeline is extended with a *filter function* – an *unary classifier*. It is trained with only one class. From a general crawl, all data which cannot be classified by the unary classifier can be used as a negative set.

As one can see in Figure 5.2, the previously described architecture is extended by a component called control unit. It manages the execution of the different classifiers and the web crawler over REST calls. So the crawler and the classifier pipeline are loosely coupled components communicating over REST. The main advantages of this loose coupling are that the components can be easily exchanged or updated, e.g. when there is a new Heritrix version, and they can be reused in another context. In addition to that Heritrix also uses REST for its internal communication, so it is simply good style to be consistent.

Furthermore, Heritrix offers a web user interface, and with REST it is very simple to provide such an interface for the classifier pipeline too. The user interface shows the different trained classifiers and gives important information about the classifier's settings and evaluation results (cf. Figure 5.3).

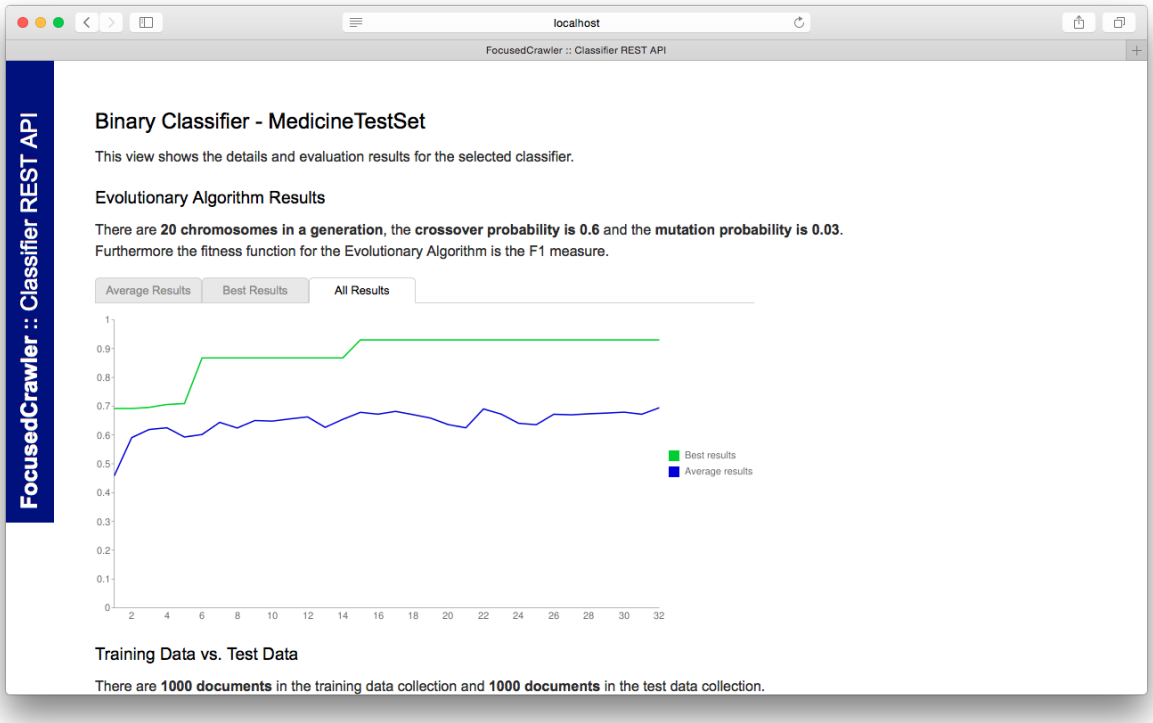

Figure 5.3: The classifier pipeline's user interface with evaluation results.

The classifier pipeline's REST service uses XML as the representation format which can be also accessed with a web browser. However, the presentation as an XML tree is not designed for user friendly readability, but with *XSLT* (*Extensible Stylesheet Language Transformation*) [27] it is possible to add a template to the XML tree and convert it into HTML and enrich it with CSS and JavaScript. All major modern web browsers support this without plugins [38].

5.3.1 Unary Classifier

The main purpose of the unary classifier is, to filter the data, collected by a short general crawl, to generate a negative text collection with a maximum purity. With this negative train and test collection we can train and evaluate the binary classifier, because a binary classifier's accuracy is much better most of the time. We will point that in Chapter 6.

Basically, the system uses the *LibSVM*[1] – as a WEKA extension – which provides an algorithm for unary classification with the parameter `-S 2` (the parameter enables the unary classification mode of the LibSVM with this setting). It is trained with the positive input data.

The function that generates the negative text collection tests every document from a short general crawl with the unary classifier. If the document is not classified as positive, it will be part of the negative text collection.

[1] `https://www.csie.ntu.edu.tw/~cjlin/libsvm/` (checked 09/20/2015)

It is not possible to use an evolutionary algorithm for feature subset selection here, because that requires a test set with positive and negative data. And if negative data is available, the unary classifier would be unnecessary. Therefore, the unary classifier uses predefined feature sets.

We use surface features here, because the algorithm is used as a first filter, like the most frequent tokens, n-grams or tokens with a special part of speech tag. The experiments and the evaluation results of the unary and the other classifiers are presented in Chapter 6.

5.3.2 Binary Classifier

We use the binary classifier to build a train and test set for the hyperlink classifier. As explained before, the data of a general crawl is used. A function in the control unit selects one hyperlink per document randomly and visits the linked document with *Jsoup*[2], an HTML parser library for Java. The returned document is classified by the binary classifier and the hyperlink in the source document is marked and labeled accordingly for the hyperlink train set.

The binary classifier, and also the unary classifier, handle HTML documents and plain text. Therefore *Boilerpipe*[3] [31], a library for content extraction, is used to extract the plain text if the document contains HTML. So the classifier is concentrated on the main content.

In contrast to the unary classifier, the binary classifier can be trained with or without the support of an evolutionary algorithm. If it is trained without the evolutionary algorithm the features are the 100 most frequent tokens with stop-word filtering and the classification algorithm is the LibSVM in binary mode.

If the classifier is trained with evolutionary algorithm support one of the following classification algorithms – provided by WEKA – is used per chromosome:

- Decision Stump
- IBk (k-NN implementation)
- J48 (C4.5 implementation)
- LibSVM
- Naïve Bayes
- OneR
- Random Tree
- ZeroR

In Chapter 3 we explained only some of the most important classification algorithms. The algorithm pool contains a few more algorithms, e.g. ZeroR, a classifier that predicts always the majority class [61]. Therefore, we do not expect ZeroR to be a good classifier for text classification. In this regard, we will compare the performance of different algorithms in Section 6.3.2 and in Section 6.4.1.

Every classifier may consist of n feature sets out of a feature pool. Each document runs through a UIMA pipeline where the selected feature extractors are executed. In a chromosome every feature set is unique, e.g. it is not possible that a chromosome contains the 100 most frequent tokens and the 200 most frequent tokens at the same time. The possible feature sets are explained in the following paragraphs.

[2] `http://jsoup.org` (checked 09/20/2015)
[3] `https://github.com/kohlschutter/boilerpipe` (checked 09/20/2015)

The n most frequent tokens work as explained before. Furthermore, there is a concept we call "center tokens". This feature extractor collects tokens that occur in a minimum number of documents, but do not occur in too many documents – so there is also an upper bound. In addition to that, there are n-gram feature extractors for $n \geq 2$ for the previously mentioned extractors. So it is possible that there are e.g. the n most frequent tokens and the n most frequent n-grams in a chromosome.

The last feature extractor based on simple tokens collects all tokens which are longer than a threshold n. Furthermore, there are a couple of length feature extractors which are all based on a simple binary decision: Is the counted value above or below a certain threshold? These feature extractors count the characters, the tokens or the sentences.

For part-of-speech tagging the system uses the *OpenNLP POS tagger*[4] and there are a lot of feature extractors based on part-of-speech tagging. These feature sets are the most frequent part-of-speech n-grams, adjectives, adverbs, nouns and verbs.

Lastly, the system contains a couple of Boolean ratio feature extractors. These feature extractors measure for each document e.g. the ratios of adjectives, adverbs, pronouns, verbs, numbers or even emoticons. We use a variable threshold to convert the ratios into a Boolean value.

5.3.3 Hyperlink Classifier

The hyperlink classifier is the core component of the whole software. It is responsible for the prediction of every hyperlink, if it redirects to a positive or negative document regarding to the current crawling topic. And for every hyperlink the crawler sends the HTML document with the marked hyperlink to the classifier and prioritizes the link based on the classification result.

As mentioned above, the classifier is trained with a labeled set of positive and negative data, and in each document the considered hyperlink is marked. Furthermore, the hyperlink classifier is based on the binary classifier. When the evolutionary algorithm is active, it selects the classification algorithm out of the same algorithm pool. The pool of feature extractors is a bit larger here, because it contains all the feature extractors from the binary classifier and some added feature sets especially for properties of the considered hyperlink.

The feature sets from the binary classifier are included to give the evolutionary algorithm the chance to include also the classification of the source document and not only hyperlink information. With respect to the higher number of possible feature sets, the number of chromosomes per generation is increased from 20 (used by the evolutionary algorithm for binary classification) to 30.

In contrast to the other implemented classifiers, the hyperlink classifier only handles HTML data because it is required for hyperlink extraction. Internally it uses the extracted plain text for the feature extractors inherited from the binary classifier and the HTML code for the hyperlink feature extractors. For the hyperlink extraction Jsoup is used again. In the following paragraphs the feature sets added for hyperlink classification are explained.

As in the binary classifier the focus is on tokens. We consider the most frequent tokens of the HTML title, the tokens of the text in the hyperlink's neighborhood and the tokens of the hyperlink's label. Furthermore, something we call "URL tokens" is included. This means the URL of the source document or the target document is split at all dots, hyphens and slashes to get the tokens. When this is done, the most frequent tokens form the feature set.

[4] `http://opennlp.apache.org` (checked 09/20/2015)

Another feature extractor counts the occurrence of the considered hyperlink in a document. It considers only the hyperlink's URL and not its label. A related feature extractor identifies the hyperlink's position in the document. A further feature extractor looks at the URL's length.

In addition to that, the system contains a feature extractor that considers the HTML tags and attributes, like IDs and classes, in the hyperlink's neighborhood [45]. Finally, there is a feature extractor that compares the source document's URL with the URL of the target document. It is a Boolean feature with a threshold. If the compared URLs are similar both documents are probably part of the same website and may share the same topic.

Using the tokens in the hyperlink's neighborhood, its label and its URL as features is not a new approach. It is used by Chakrabarti et al. in 2002 [11] and Cho et al. [12], but they used different strategies for feature subset selection and fine-tuning. The focus in this thesis is the automatic generation of a training set and the fine-tuning of the hyperlink classifier by evolutionary algorithms.

5.3.4 Focused Crawler

The focused crawler is an extended Heritrix crawler which makes use of the previous mentioned components – directly and indirectly. It is based on the principles explained in Chapter 2. Whenever the crawler extracts a hyperlink from a fetched document it sends the document with the marked hyperlink to the hyperlink classifier. After the processing the hyperlink classifier returns a positive or a negative response. If it is classified as positive it gets a higher priority such that it is crawled before less important hyperlinks.

Heritrix is not made for focused crawling, at least for our strict definition, so it has to be extended. The first problem is that the HTML code is no longer available in memory when Heritrix considers the URLs. That is why the `CandidatesProcessor`, which is positioned in the disposition chain, is extended, because it is the last chance to save the HTML code in the `CrawlURI` object for further use during the crawling procedure.

Now the HTML code is also available in the candidate chain and we can add our new extension, where we combine Heritrix with the hyperlink classifier. When a hyperlink is classified as positive we update its scheduling directive and use `HIGH` as scheduling constant. If the hyperlink is not positive then its priority is not touched. Figure 5.4 illustrates why negative hyperlinks are not filtered out.

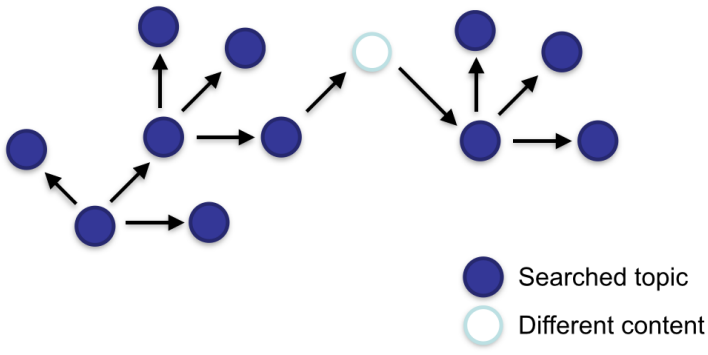

Figure 5.4: Focused crawling over a combining negative example.

Negative links are not crawled with a higher priority because it is possible that meshes of positive documents are not connected directly. So the crawler has the chance to find more positive documents over a few intermediary negative documents [10].

6 Experiments and Evaluation

In this chapter we will present and discuss the experiments and its evaluation for all trained classifiers and lastly for the focused crawler. Particular attention is given to the results of the used evolutionary algorithm for feature subset selection.

6.1 Crawling Topics and Corpora

For the evaluation of the focused crawler and its components we use three different real world topics: medicine, finance and technology. Furthermore, we analyze the classifiers performance on language prediction.

During development we used manually collected documents from the web. These collections were small-sized and is was a lot of work to collect and assess the documents. Nevertheless, they had a high content quality.

However, for the final corpora we used a different and more efficient approach – the *BootCaT method* [2]. This method starts from a set of words, tuples or even triples representing the topic of interest. Every item in this set is a query for a search engine. The best results for every query are collected.

The size of the corpus depends on the size of the word set. In the corpora generated for this thesis, we chose appropriate words from the convenient development sets. Based on this word lists we generated enough triples to reach the desired corpus size. The original BootCaT method provides also a different approach to generate a larger corpus: Later word lists can be generated using previously downloaded web documents. This enables a system that can be interrupted when it reaches the desired corpus size.

The method's quality is based on the quality of the used search engine – a better ranking of the results leads to a better corpus. Generally, using a web search engine is always a black box. The ranking cannot be reproduced after a certain amount of time [28].

In the corpora generated for this thesis, we used Microsoft Bing because of their API and the amount of free quota[1]. However, every search engine and even custom information retrieval systems are conceivable. The latter makes it possible to influence e.g. the documents ranking.

Lastly, the collected documents must be cleaned. Here we used the boilerpipe library to extract the content of each document. Furthermore, we checked the topic of each document manually.

[1] `http://datamarket.azure.com/dataset/bing/search` (checked 09/02/2015).

6.2 Filtering with the Unary Classifier

We use the unary classifier to provide very clear negative training data to the binary classifier. Therefore the unary classifier builds a negative training set from a standard Heritrix crawl in the size of the positive training set given by the user. In this section we will investigate the nature of web data, used as a negative training set, and the performance of different approaches of unary classifying.

6.2.1 Natural Purity of Web Data

A unary classifier is usually not as good as a classifier trained with examples from all classes but with good features it is much better than a simple baseline, e.g. predicting always the majority class like ZeroR [61]. Nevertheless, we can use it as a filter to improve the negative training set's purity, because if we randomly download web documents it is most likely that the majority of these data is negative data with respect to a predefined positive topic.

To prove that, we made a short general crawl with `http://www.dmoz.org`, a large web directory, as seed URL. We randomly selected 500 files out of the 3.48 gigabyte crawl data. After that, we sorted and counted these files manually with respect to the defined topics finance, medicine and technology. Table 6.1 shows the results.

Table 6.1: Filtering with the unary classifier: Natural purity.

Corpus	Positive files	Percentage
Finance	5	99.0%
Medicine	4	99.2%
Technology	16	96.8%

In the 500 randomly selected files are only between 5 and 16 positive files dependent on the considered topic. That proves the assumption because about 98% of the data is negative data with respect to the predefined positive topic. So even if it is not possible to filter the negative data's purity with a unary classifier, it is possible to use a standard crawl's data for a negative set without filtering.

6.2.2 Evaluation of the LibSVM Used for Unary Classification

We check the unary classifier's performance on all three training topics and for different feature sets. In this section we consider the LibSVM. As mentioned before, it is not possible to use an evolutionary algorithm in the productive system because this requires at least negative test data, and the user's input contains only positive data.

The biggest challenge is to find a simple feature subset that fits on every imaginable topic. Therefore, we tried different feature sets and feature set combinations to find a classifier for the three considered topics.

We reached the best performance on all three considered topics with the center tokens using a top border of 30 and a bottom border of 3. As explained before, center tokens are tokens that occur in

a minimum number of documents, but do not occur in too many documents. Table 6.2 shows the LibSVM's results using center tokens.

Table 6.2: Results of the unary classifier on different corpora.

Corpus	Accuracy	Precision	Recall	F1-measure
Finance	0.3555	0.3934	0.533	0.4527
Medicine	0.343	0.3842	0.521	0.4423
Technology	0.3645	0.395	0.51	0.4452

If we look at the results we cannot expect the classifier to be a good filter. The F1-measure is between 0.44 and 0.45 as a result of an average recall and a poor precision. To find better feature sets for the LibSVM we used the evolutionary algorithm for every considered topic. Figure 6.1 shows the generations of the evolutionary algorithm on the medicine development set.

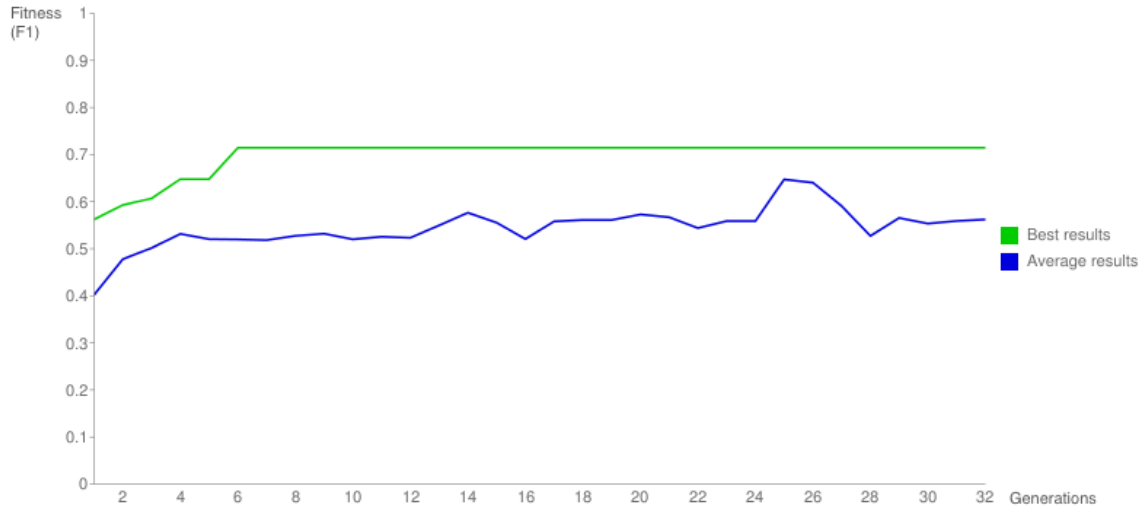

Figure 6.1: Results of the evolutionary algorithm with $p_m = 0.03$ on the medicine development set.

We see that the evolutionary algorithm found a good classifier in the sixth generation. It is based on tokens longer than 16 characters, character n-grams, POS n-grams, adverbs and the verb ratio. With 0.7143, the F1-measure is better than the F1-measure in Table 6.2. In addition to that, even the accuracy is better with 0.7246. We reached similar results on the other topics, but it was not possible to find a common combination of feature sets that outperforms the center token's results on all considered topics.

Nevertheless, we tried the LibSVM's filter ability in an experiment simulating its practical application. Therefore, we created a document collection containing 2% positive data, with respect to the considered topic, and 98% negative data (based on the results presented in Section 6.2.1). This collection consists of 2000 documents. After that, the we used the classifier to generate a negative set containing 100 documents. Therefore, a function iterates over the shuffled 2000 documents and if the classifier predicts the document not as positive, the document is added to the negative set.

We repeated this experiment 10 times for every topic to prevent irregularities because of the shuffling step. Table 6.3 shows the results for the considered topics.

Table 6.3: Results of the unary LibSVM in practice.

Corpus	Errors (AVG)	Variance	Purity (neg.)
Finance	7.6	10.49	92.4%
Medicine	0.0	0.0	100.0%
Technology	7.4	5.16	92.6%

During the generation of the negative set, the unary classifier classified most examples as positive. So there were a lot of false positives, but false positives are no issue here because only documents classified as negative were copied to the negative set.

Table 6.3 shows that it works well with the trained classifier on the medicine topic. There are no false negatives and the purity of the resulting negative set is 100%. On the finance corpus we have an average error of 7.6 and on the technology corpus we have an average error of 7.4. The purity is below 93% on both corpora. That is not a suitable result because the initial purity was 98%. However, this is not a major problem for the complete architecture because we could use the data collected by a short general web crawl without filtering (see Section 6.2.1).

6.2.3 Evaluation of a Clustering Approach Used for Filtering

Using the LibSVM with center tokens does not work well to filter positive documents on all considered topics. This is why we tried a second approach beside unary classification: An approach based on *k-means clustering* [21].

K-means clustering uses the complete document collection, represented by its feature vectors, as input and it divides these documents into k clusters. Compared to the classification algorithms, used mainly in this thesis, k-means clustering is an unsupervised approach. The algorithm has to divide the documents without a learned model. Furthermore, the algorithm is not based on any labeled training examples. The basic k-means clustering algorithm is sketched in the following list:

1. The algorithm chooses randomly k cluster centers.
2. Every example is added to the closest cluster center.
3. The positions of the cluster centers are calculated again.
    a) Based on the examples in the current clusters.
4. Every example is added to the closest new cluster center.
5. Goto Step 3 until convergence.

This algorithm can be used for the filtering challenge. We have 98% negative and 2% positive data and want to filter the positive data to generate a completely negative set. We apply the k-means clustering algorithm with a certain k and hopefully all positive documents are in the same cluster. In practice, the positive documents are distributed over several clusters because the negative documents are very noisy documents collected from the web.

As mentioned before, most of the time the positive documents are distributed over several clusters. That is why we cannot simply identify the positive cluster. Nevertheless, we can try to identify the cluster with the highest purity, with respect to the negative documents.

Therefore, we calculate the centroid of the positive train set. After that, we calculate the Euclidean distances of the positive training centroid to all the other cluster centers (WEKA provides k-means

clustering and allows to access a cluster's centroid). Finally, we choose the cluster with the largest distance to the training data.

For the experiments we used 1000 positive documents for the cluster representing the training data, and 1000 test documents which contained 98% negative and 2% positive data. The documents were represented by vectors containing the 250 most frequent tokens of the training data. Table 6.4 shows the results for different values of k.

Table 6.4: Results of the clustering approach for different values of k.

Corpus	k	Positive	Negative	Purity (neg.)
Finance	1	20	980	98.00%
	2	5	177	97.25%
	3	1	135	99.26%
	4	0	55	100.00%
	5	0	50	100.00%
Medicine	1	20	980	98.00%
	2	11	871	98.75%
	3	2	108	98.18%
	4	2	104	98.11%
	5	0	147	100.00%
Technology	1	20	980	98.00%
	2	4	146	97.33%
	3	0	57	100.00%
	4	0	57	100.00%
	5	0	57	100.00%

The first line (for $k = 1$) of every topic shows the start distribution with a purity of 98%. For $k > 2$ the approach reaches a purity over 98% for all three considered topics and even 100% on the technology topic. For $k = 5$ the purity is 100% for every considered topic – the optimal result. In contrast to the LibSVM approach we do not have a fixed number of documents in the negative set. The number of documents in the result set depends on the size of the cluster with the largest distance to the training data. Usually, a larger k leads to a smaller result set (but not always, see $k = 5$ on the medicine topic). However, when generating a negative set, it is an advisable approach to iterate over different values of k and choose the result set generated by the largest k that contains an appropriate number of documents.

Lastly, the data show another interesting aspect. The LibSVM approach, previously explained in Section 6.2.2, tries to separate the documents into two classes, but it works only on the medicine topic. We do not have the same data set nor the same features in the clustering experiment but we can still compare the results because we have the same data distribution. For $k = 2$ the approach does not improve to purity of the negative set – only on the medicine topic we see a small improvement. The experiment proves that we need more that one split through the data set.

6.3 Text Classification with the Binary Classifier

We investigate the binary classifier's performance on the text sets already mentioned above – finance, medicine, technology. Mainly, we will focus on the medicine topic in most detail. Additionally, we analyze the evolutionary algorithm's behave, e.g. the fitness development, the selected features or the selected algorithms.

6.3.1 Performance on the Different Topics

The crossover probability p_c, the mutation probability p_m and the number of chromosomes per generation have a high influence on the result of an evolutionary algorithm. We set the number of chromosomes per generation to 20 because this number is only slightly greater than the number of possible feature sets in the system. Furthermore, we set $p_c = 0.6$ because it is the common value found in literature (explained in Section 4.2.4). In addition to that, in practice it is impossible to check all combinations of p_c and p_m. Lastly, we checked different values for p_m represented in Table 6.5.

Table 6.5: Evolutionary algorithm results for the medicine corpus.

p_m	Best result	Worst result	Average	Standard deviation
0.01	0.96970	0.95630	0.96293	0.0067
0.02	0.971	0.9382	0.95833	0.01763
0.03	0.9686	0.9604	0.96413	0.00415

Table 6.4 shows the best, worst and the average F1-measure for nine tests of the evolutionary algorithm for feature subset selection on the medicine corpus. We made three tests for every mentioned mutation probability value to find the best value for p_m for our evolutionary algorithm. Furthermore, the table contains the standard deviation of the results for every value of p_m.

The evolutionary algorithm has the best result for $p_m = 0.02$ but this line contains also the worst result and the worst average result. The results for $p_m = 0.03$ are slightly better than the results for $p_m = 0.01$ but overall they are similar.

However, even if $p_m = 0.02$ contains the best result, it is not a good idea to choose this value for further use because it contains also the worst result. Therefore, we introduced the standard deviation: We are interested in a reliable evolutionary algorithm and we define reliability in this case with results which have a low standard deviation.

That is why we have chosen $p_m = 0.03$ for further use because it leads to the lowest standard deviation. Furthermore, this value leads to the best average result.

Firstly, we analyze the binary classifier's results on the medicine corpus. We used 20 chromosomes per generation, a crossover probability of 0.6 and a the previously determined mutation probability of 0.03. Figure 6.2 shows the progress of the evolutionary algorithm over 32 generations. The green line represents the best and the blue line the average results.

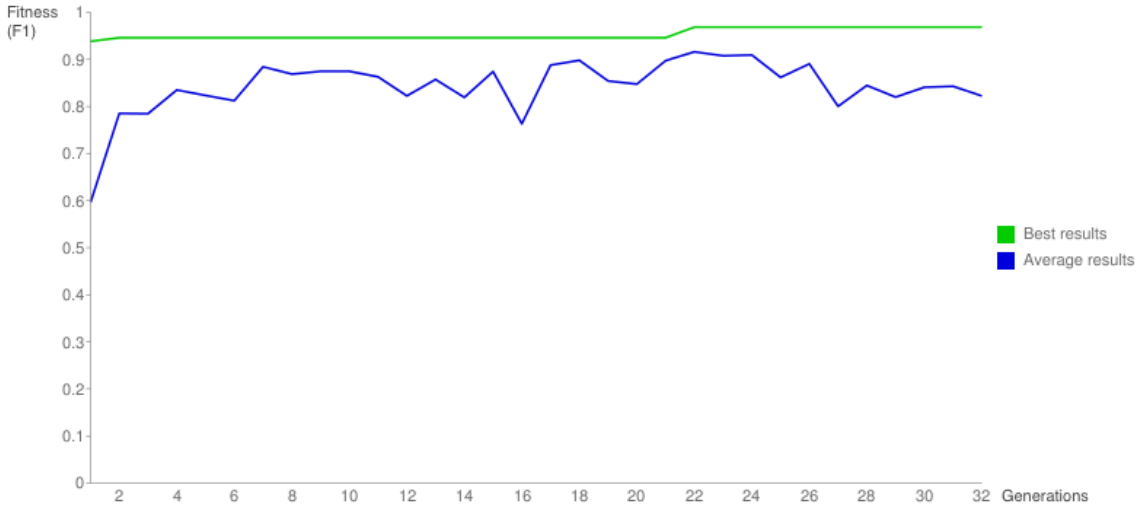

Figure 6.2: Results of the evolutionary algorithm with $p_m = 0.03$ on medicine corpus.

The average results start at an F1-measure of 0.596, increase by 0.3 in the first 20 generations. After that, the value decreases slightly by about 0.1. The average results have the fastest growth in the first eight generations. The best results start at an F1-measure of 0.939 and increase to 0.946 in the second and to 0.969 in the 22nd generation.

The evolutionary algorithm uses the F1-measure as fitness function and Figure 6.2 shows only the F1-measure but also the other commonly used measures are interesting. Figure 6.3 shows a diagram which contains the accuracy, precision, recall and F1-measure on the medicine test set. The training set consists of 2000 documents where 1000 documents are positive and 1000 are negative. The test set consists also of 2000 documents with the same distribution of positive and negative documents.

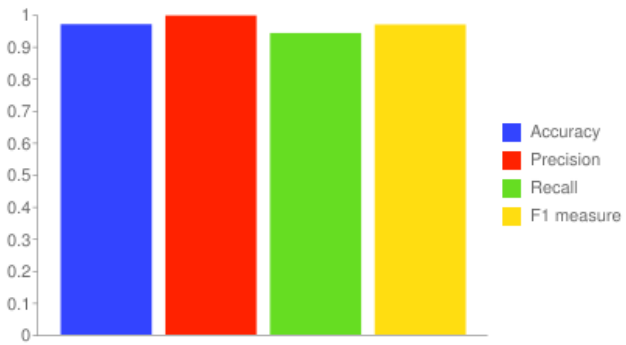

Figure 6.3: Results of the found classifier on medicine corpus. We used an evolutionary algorithm with $p_c = 0.6$ and $p_m = 0.03$.

The found classifier has an accuracy of 0.970, a precision of 0.997, a recall of 0.942 and an F1-measure of 0.969 on the training data. This means that the classifier classifies 97% of the training examples correctly. The selected classifier is the LibSVM. The selected features sets are the 303 most frequent nouns from the training set, the pronoun ratio and a length feature which checks if the content is longer than 39878 characters.

However, the test examples have a high influence on the result of the evolutionary algorithm because the algorithm's selection process depends on the fitness function which is calculated on the test examples. So, overfitting on the test data could be conceivable. To eliminate this apprehension, we tested the trained medicine classifier on an unseen test set. The new test set consists of 400 documents where 200 documents are positive and 200 are negative. Figure 6.4 shows the results on this test set.

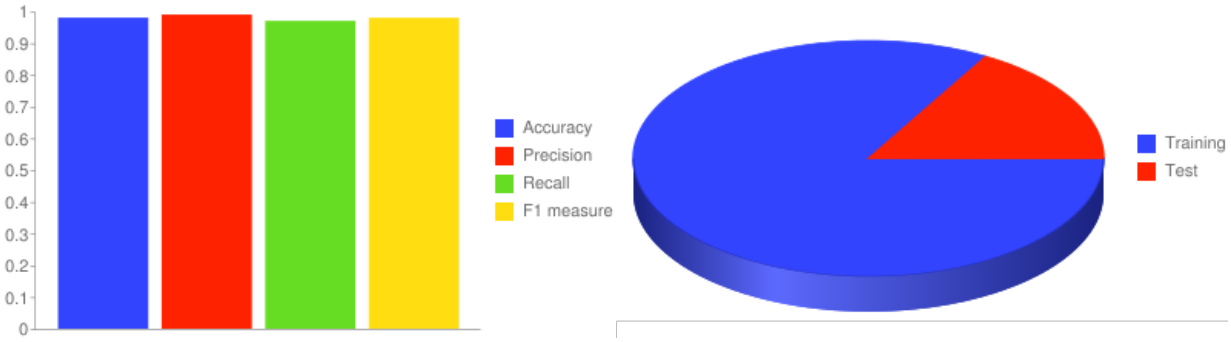

Figure 6.4: The classifier's performance on an unseen test set containing 400 documents. The train set contains 2000 documents.

The classifier has an accuracy of 0.98, a precision of 0.99, a recall of 0.97 and an F1-measure of 0.98 on the unseen test set. Compared to the results on the test set used with the evolutionary algorithm, the results are even slightly better. So, in this case we cannot observe any overfitting. Nevertheless it is imaginable, that a smaller training and especially a smaller test set may lead to overfitting with a higher probability. Figure 6.5 compares the results of a classifier trained and tested on a smaller set. Furthermore, we evaluated this classifier with the previously mentioned unseen test set.

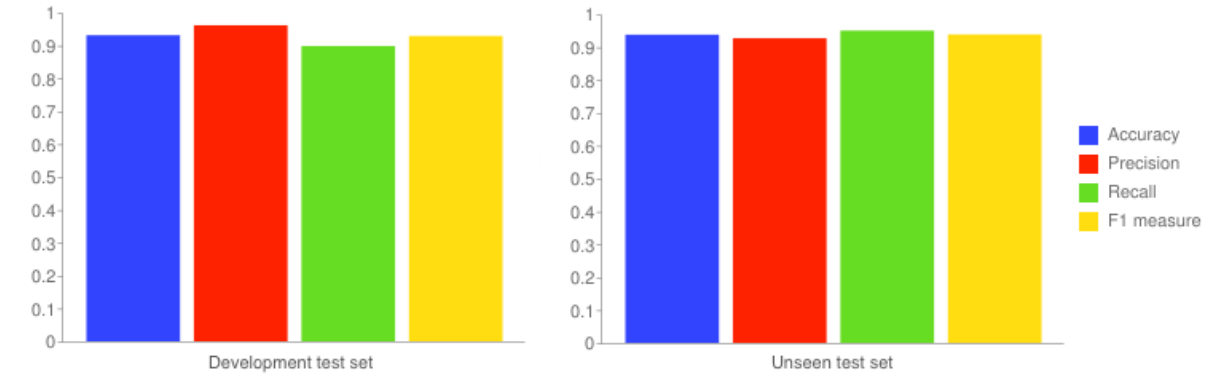

Figure 6.5: It works also with a smaller set of training data: 666 documents in the train set and 334 documents in the development test set. The unseen test set contains 400 documents.

The medicine development test set consists of 334 documents and the classifier was trained by 666 documents. Both, the train and the test set contain 50% positive and 50% negative documents. The evolutionary algorithm reached an accuracy of 0.931 and a precision of 0.962, a recall of 0.898 and an F1-measure of 0.929 on this test set.

On the unseen test set, the same classifier has an accuracy of 0.938 and a precision of 0.927, a recall of 0.95 and an F1-measure of 0.938. Compared to the classifier created with the larger train and test sets, these values are slightly below the values mentioned before, e.g. the classifier created with the larger train set reaches an F1-measure of 0.98 on the unseen test set. However, even with smaller train and test sets we cannot observe any overfitting.

In addition to the medicine topic, we analyze the evolutionary algorithm's behavior on the finance and the technology topic too. Figure 6.6 shows the evolutionary algorithm's progress over 32 generations on the finance set on the left and the technology set on the right side. Again, we set $p_m = 0.03$, $p_c = 0.6$ and we used 20 chromosomes per generation. Furthermore, we used the same set size and the same distribution as for the train and test sets used for the medicine topic.

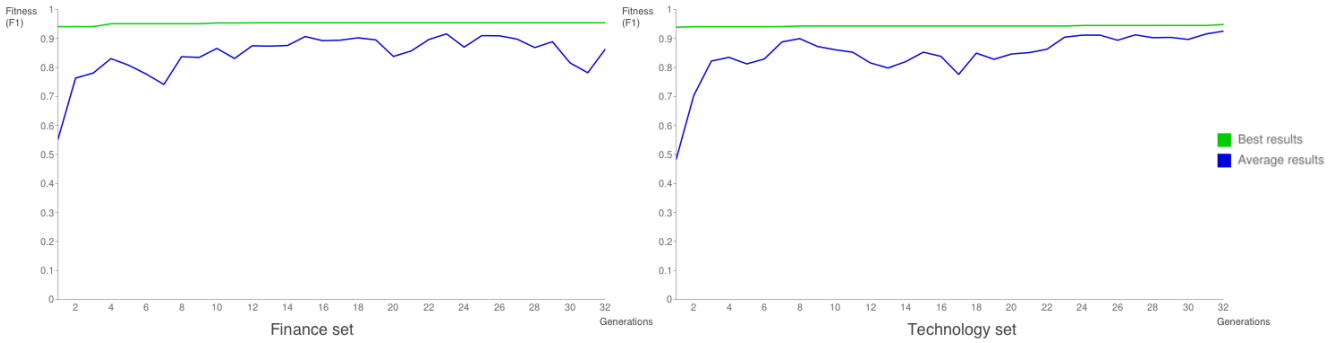

Figure 6.6: The evolution on the finance (left) and the technology set (right). We set $p_m = 0.03$.

The best F1-measure on the finance set starts on a high level, and it increases only in generation four to the final classifier with an F1-measure of 0.954. The progress of the average results can be compared to the medicine results. They start at 0.551 and increase to 0.907 in generation 15. After that, the average results keep their level except some local minima.

The classifier, created by the evolutionary algorithm, has an accuracy of 0.955, a precision of 0.972 and a recall of 0.937. It is based on the LibSVM and uses the following feature sets: the pronoun ratio, money signs (e.g. \$ or €), a number ratio, a question ratio, the 1529 most frequent tokens and the 678 most frequent adverbs.

The best F1-measure on the technology set starts also on a high level. It starts at 0.939 and increases slightly multiple times to 0.948. With 0.481, the average results start on a lower level than on the medicine or finance set. In the end, the average results converge to the best results with an F1-measure of 0.926 in generation 32. This classifier has an accuracy of 0.9485, a precision of 0.9489 and a recall of 0.948. Again, it is based on the LibSVM. Furthermore, it uses the following feature sets: The 659 most frequent nouns, the number ratio and the 334 most frequent tokens.

6.3.2 Classification Algorithms Chosen by the Evolutionary Algorithm

The three considered classifiers use the LibSVM as classification algorithm because the evolutionary algorithm has selected a chromosome containing the LibSVM because of its fitness function. As described in Section 5.3.2, the evolutionary algorithm for binary classification has the possibility to choose an algorithm out of a classification algorithm pool. In this section, we will compare the performance of the different classifiers. Table 6.6 shows the number of the classification algorithms, selected by the evolutionary algorithm, for different F1-measure intervals.

Table 6.6: Best classification algorithms generated by the evolutionary algorithm (binary classification).

Corpus	F1-measure	LibSVM	Naïve Bayes	J48	Other
Finance	≥ 0.80	22	87	1	16
	≥ 0.85	20	87	1	3
	≥ 0.90	15	84	1	0
	≥ 0.95	6	0	0	0
Medicine	≥ 0.80	22	160	4	29
	≥ 0.85	22	159	4	8
	≥ 0.90	22	142	2	1
	≥ 0.95	4	0	0	0
Technology	≥ 0.80	4	199	62	20
	≥ 0.85	3	188	48	4
	≥ 0.90	1	130	17	0
	≥ 0.95	0	0	0	0

The results, presented in Table 6.6, show that all classifiers with an F1-measure ≥ 0.95 are based on the LibSVM. Even the technology classifier is based on the LibSVM with an F1-measure of 0.948. The naïve Bayes classifier is slightly below the LibSVM, but the evolutionary algorithm produces more suitable classifiers based on naïve Bayes than on the LibSVM.

The other classifiers are far behind the performance of the LibSVM and naïve Bayes for binary classification. Only on the technology set, J48 works well too. Later, we will compare the classification algorithms chosen by the evolutionary algorithm for hyperlink classification. Furthermore, we will compare the hyperlink results to the binary results.

6.3.3 Performance in Language Detection

The Focused Crawler should also work for building text collections of different language domains. Even if language detection is not the main focus of this thesis, it is useful to analyze the feature and algorithm selection of the evolutionary algorithm on different languages compared to the selection on our three topics.

Again, we used 20 chromosomes per generation and we set $p_m = 0.03$ and $p_c = 0.6$. The train set consists of 200 English documents (positive) and 150 German documents (negative). The test collection has the same distribution.

Language detection is a simple classification task. The classifier reaches an F1-measure of 1.0. Mainly, it uses word-based features. The feature sets are: the 1820 most frequent n-grams ($n = 3$), the adjective ratio and the 860 most frequent adverbs. Furthermore, the resulting classifier is based on naïve Bayes as classification algorithm and not on LibSVM as the previously described topic classifiers. However, the complete results of the evolutionary algorithm show that the choice of the classification algorithm is not important here. It is possible to reach an F1-measure ≥ 0.95 with almost every algorithm in the classification algorithm pool.

6.3.4 Performance with an Unfiltered Train and Test Set

As described in Section 6.2.1, the data collected by a standard web crawl, starting from a general set of seed URLs, contains mostly negative data, with respect to the topic of interest. We observed a negative set's purity of about 98%.

It is conceivable, that it is also possible to use a binary classifier trained on a pure positive and impurified negative set. To prove that, we used the evolutionary algorithm to find a suitable feature subset and an algorithm for a binary classifier for the medicine set. We used 20 chromosomes per generation and we set $p_c = 0.6$ and $p_m = 0.03$. Furthermore, the train set consists of 1000 positive and 1000 negative documents. The test set has the same distribution. Additionally, for this experiment it is important that the negative sets include 20 positive documents to guarantee the required impurity. Figure 6.7 shows the progress of the evolutionary algorithm.

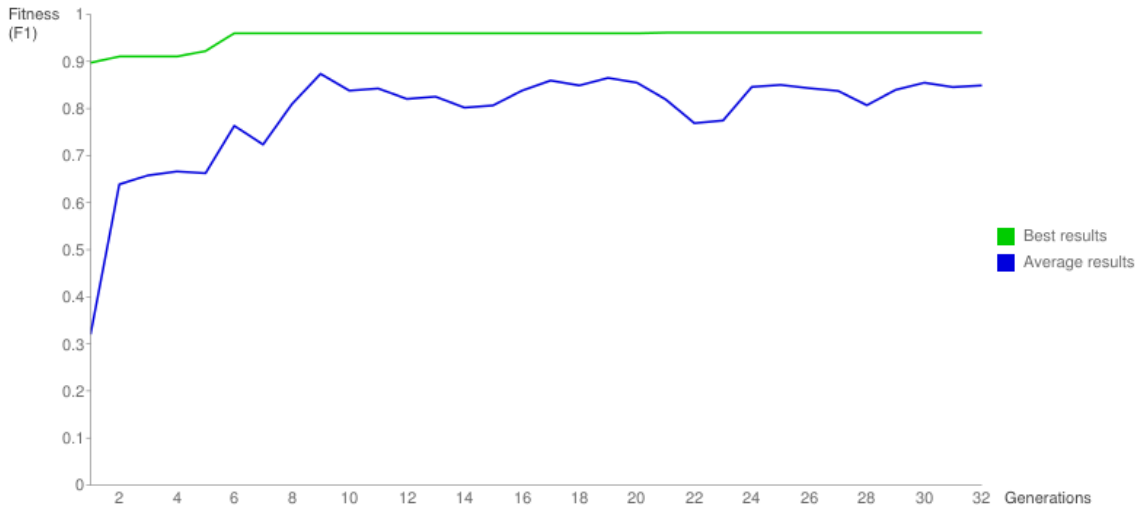

Figure 6.7: Results of the evolutionary algorithm with $p_m = 0.03$ on the medicine set with a 98% pure negative and a 100% pure positive set.

The evolutionary algorithm reaches the best F1-measure of 0.96 in generation six. Compared to the other investigated evolutionary algorithm runs for binary classification, the average results of this run start at a lower level – with an average F1-measure of 0.32. The average results achieve its peak in generation nine with an average F1-measure of 0.87.

The classifier, created by the evolutionary algorithm, is based on the LibSVM (like all evaluated binary classifiers in this thesis, except the binary classifier for language detection). Furthermore, it uses the following feature sets: An adverb ratio, the 452 most frequent nouns, the 35 most frequent character

n-grams ($n = 2$), an emoticon ratio, the number of sentences in a text and the 102 most frequent tokens. Figure 6.8 shows the accuracy, the precision, the recall and the F1-measure on the impure medicine test set (left) and the classifier's results on an unseen test set (right).

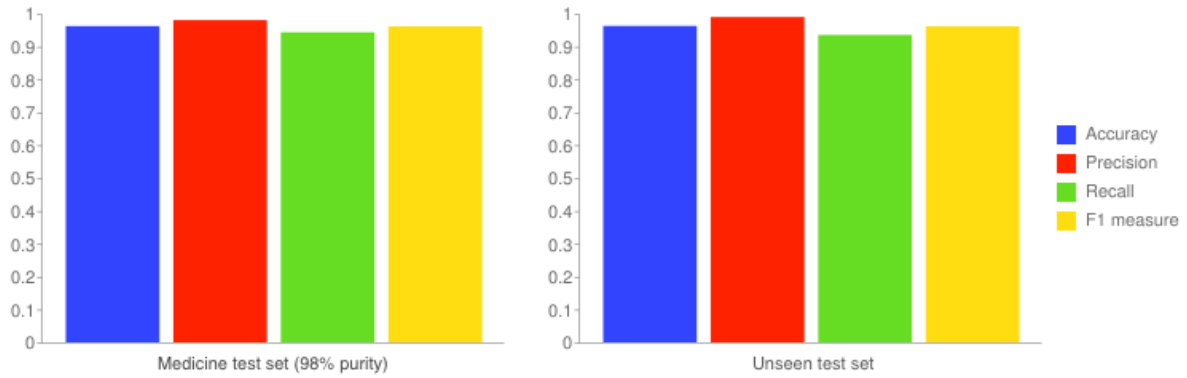

Figure 6.8: Results of the classifier trained on the medicine set with 98% purity (left) and the results on an unseen test set (right).

The classifier has an accuracy of 0.96, a precision of 0.98 and a recall of 0.94 on the test set used by the evolutionary algorithm. The unseen test set consists of 200 positive and 200 negative documents. On this set, the classifier has an accuracy of 0.96, a precision of 0.99, a recall of 0.94 and an F1-measure of 0.96.

The results are on the same level as the observed results of the classifiers based on a 100% pure train and test set. Thus, it is possible to use the data collected by a standard web crawl, starting from a general set of seed URLs, as negative set. Furthermore, it is also possible to use the binary classifier for the generation of the hyperlink classifier's train and test set because the results indicate no difference to the classifiers evaluated above.

The hyperlink classifier's performance is essential for the overall performance of the focused crawler. The hyperlink classifier is based on the technology of the binary classifier. In fact, internally it is a simple binary classifier. Furthermore, it uses the same feature and algorithm pools, but the feature pool is expanded to include special hyperlink features, e.g. parts of a document's URL or tokens in a hyperlink's neighborhood.

Again, we start with the analysis of the medicine set. The training collection contains 610 positive an 724 negative documents. The test data collection has the same distribution. We used a larger set of negative documents to support classification algorithms which choose the majority class in case of poor coverage, because the web contains more negative documents, with respect to the considered topic.

Figure 6.9 shows the evolutionary algorithm's progress with 30 chromosomes per generation. We increased the number of chromosomes per generation because of the larger feature pool compared to the binary classifier. Furthermore, we set $p_m = 0.03$ and $p_c = 0.6$.

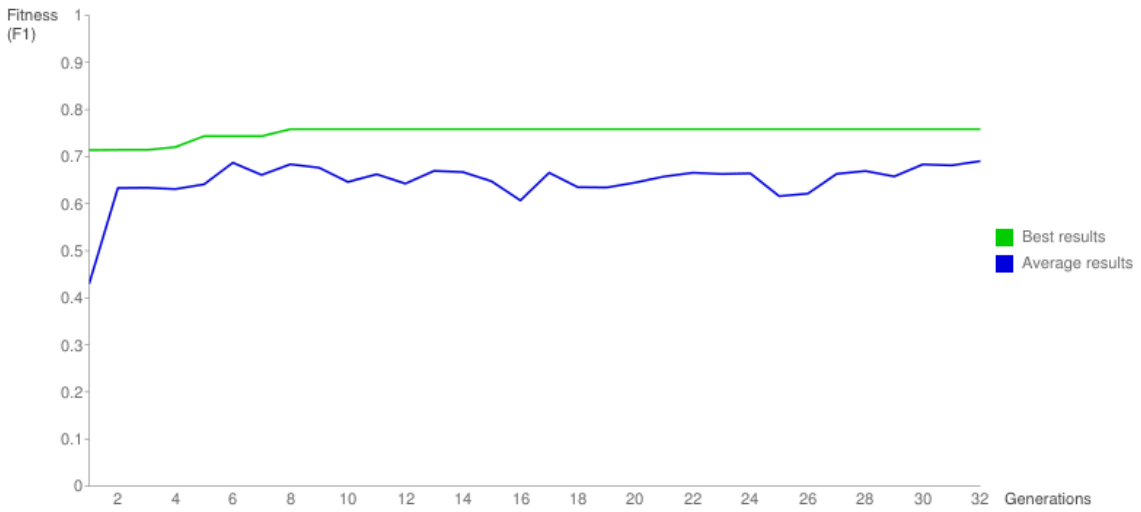

Figure 6.9: Results of the evolutionary algorithm with $p_m = 0.03$ on the medicine hyperlink set.

The best F1-measure in generation one is 0.714 and it increases to the final result of 0.758 after eight generations. This means that the evolutionary algorithm converges quickly and in practice it is not necessary to use 32 generations.

The behavior of the average results can be compared to the average results' behavior of the evolutionary algorithm for binary classification. The results start at a low level and increase quickly. After that, it goes on with a lateral movement.

The classifier, created by the evolutionary algorithm, is based on the J48 classification algorithm. Furthermore, it uses the following feature sets: The source document's adjective ratio and the 406 most frequent URL tokens of the source document. So, the classification is only based on the source document. In addition to that, the feature sets focus the document's URL and not its content.

Figure 6.10 shows the classifier's results on the medicine hyperlink test set. This classifier has an accu-

racy of 0.747, a precision of 0.674, a recall of 0.866 and an F1-measure of 0.758.

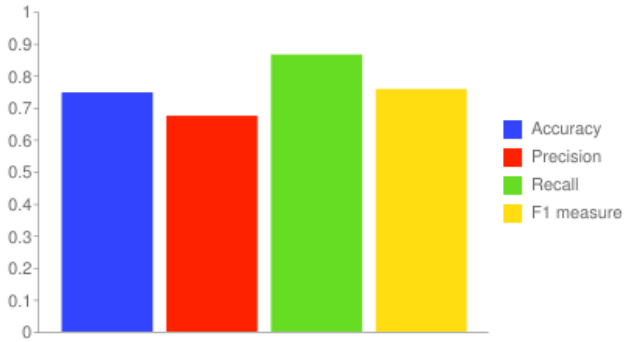

Figure 6.10: Results of the found classifier on the medicine hyperlink set. We used an evolutionary algorithm with $p_c = 0.6$, $p_m = 0.03$ and 30 chromosomes per generation.

Figure 6.11 shows the the evolutionary algorithm's progress on the finance hyperlink set, on the left side, and on the technology hyperlink set, on the right side. The finance hyperlink train set consists of 444 positive and 454 negative documents. The technology hyperlink train set consists of 549 positive and 594 negative documents. Both test sets have the same distribution. Again, we used 30 chromosomes per generation and we set $p_m = 0.03$ and $p_c = 0.6$.

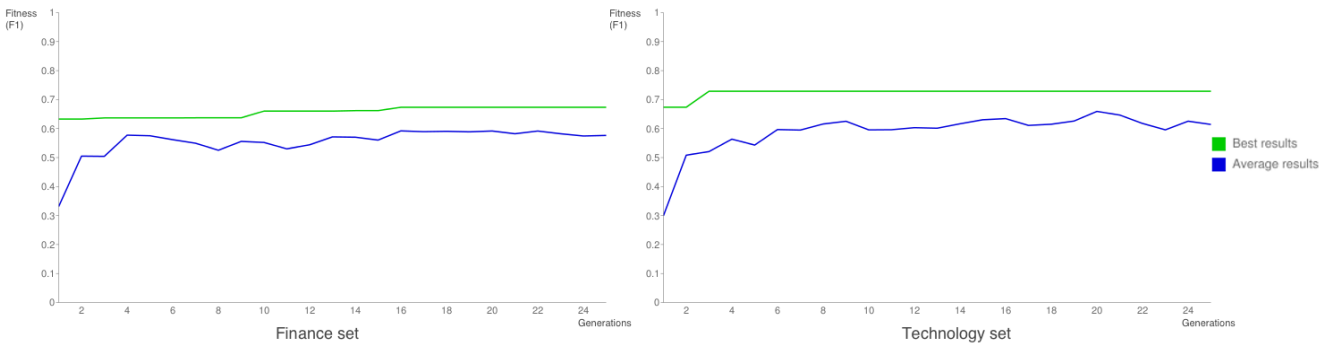

Figure 6.11: The evolution on the finance (left) and the technology hyperlink set (right). We set $p_m = 0.03$.

Both runs can be compared to the medicine run; they converge in the first 20 generations. The finance hyperlink classifier reaches an F1-measure of 0.673, an accuracy of 0.607 and a precision of 0.571 and a recall of 0.820. Is is based on IBk as classification algorithm and is uses the following feature sets: Center tokens using a top border of 35 and a bottom border of 6, various money signs, the count of the considered URL in the source document and the most frequent URL tokens from the source document.

The technology hyperlink classifier has an accuracy of 0.701, a precision of 0.647, a recall of 0.834 and an F1-measure of 0.729. It uses only hyperlink features: The 269 most frequent hyperlink label tokens and the most frequent source and target URL tokens. In addition, the used classification algorithm is naïve Bayes.

6.4.1 Classification Algorithms Chosen by the Evolutionary Algorithm

In contrast to the evolutionary algorithm for binary classification, the LibSVM seems not to be the best classifier for hyperlink classification. It has been never selected by any tested run of the evolutionary algorithm. Table 6.7 shows the number of classification algorithms, selected by the evolutionary algorithm, for different F1-measure intervals.

Table 6.7: Best classification algorithms generated by the evolutionary algorithm (hyperlink classification).

Corpus	F1-measure	LibSVM	Naïve Bayes	J48	IBk	Other
Finance	≥ 0.55	0	102	6	232	46
	≥ 0.60	0	13	1	92	9
	≥ 0.65	0	0	0	12	0
	≥ 0.70	0	0	0	0	0
Medicine	≥ 0.55	8	124	99	8	23
	≥ 0.60	7	120	98	4	21
	≥ 0.65	7	101	82	2	1
	≥ 0.70	1	20	44	0	0
Technology	≥ 0.55	9	201	124	7	15
	≥ 0.60	0	196	109	3	8
	≥ 0.65	0	20	25	0	4
	≥ 0.70	0	8	0	0	0

The LibSVM is only on the medicine hyperlink set in the highest group. Good classification algorithms for hyperlink classification are naïve Bayes and J48. Only on the finance set IBk performs best, but additionally, the result on the finance set is also the least optimal one. Compared to binary classification, there is no outstanding winner but naïve Bayes and J48 lead to good results. In Section 6.5.1 we will present further evaluations of the hyperlink classifier based on J48.

6.5 Focused Crawler

Now we combine all the components and see how they work together in a real crawl – the hyperlink classifier serves as the decision function for the focused crawler.

Firstly, we analyze the focused crawler on the medicine topic, where the hyperlink classifier is based mainly on the most frequent URL tokens from the source document. Therefore, we crawled 64 gigabytes of data with the implemented focused crawler, a standard crawler (non-focused) and a focused crawler based on a medicine language model. All these crawlers started from the same 25 medicine related seed URLs.

The focused crawler based on a language model, introduced by Remus and Biemann [50, 49], calculates the perplexity for every source document's content and redefines the priority in Heritrix for the contained hyperlinks if necessary. In this thesis we will call it language model crawler. So both focused crawlers are concentrated on different aspects of the source document. Additionally, the language model crawler reached an average download rate of 2 MB/s and a peak download rate of 8 MB/s on a machine with 64 gigabytes of main memory and 24 cores, where the crawler was restricted to 16 gigabytes of main memory and 25 threads, on the educational domain.

Figure 6.12 shows the results of the implemented focused crawler, the standard crawler and the language model crawler. To evaluate the crawler, we selected seven samples containing 100 documents. The samples are generated from the 1st, 2nd, 4th, 8th, 16th, 32th and 64th gigabyte. Furthermore, they are counted by three people and Figure 6.12 shows the average in percent, e.g. a value of 40% means that the sample contains 40% positive documents.

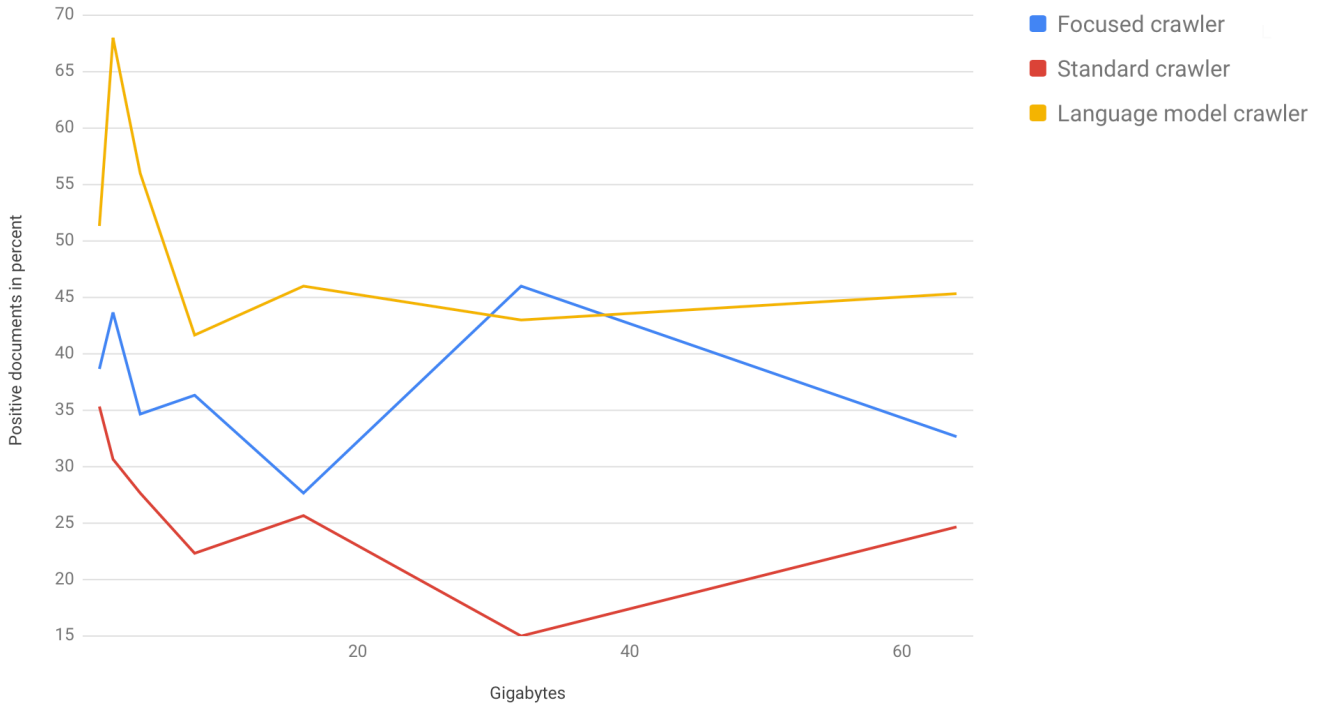

Figure 6.12: Medicine crawl evaluated by three annotators on seven samples containing 100 documents.

In this evaluation, both crawlers, the implemented focused crawler and the language model crawler, beat the standard crawler's results. The standard crawler starts at 35% positive data but after 20 gigabytes, the results are between 15% and 25%.

The implemented focused crawler starts at about 38% positive data and keeps that level except some local minima and maxima. The language model crawler starts at about 52% positive data, increases to 67% positive data and after eight gigabytes, the samples contain about 45% positive data. So in this evaluation, the language model crawler beats the implemented focused crawler. This is not surprising, because the implemented focused crawler is based mainly on a few source URL tokens and the language model crawler makes use of the complete source document. However, these results and the evaluation results of the different hyperlink classifiers prove that it is a good approach to focus the classification on the source document. In addition to that, we observed an average download rate of 2.1 MB/s and a peak download rate of 20 MB/s on a machine with eight gigabytes of main memory and two cores during the medicine crawl with the implemented focused crawler.

Furthermore, the results counted by three annotators have a high standard deviation (from 2.3 up to 24.87), even if the overall result is supported by every annotator. Especially one annotator distorts the results for the language model crawler and puts it on a higher level (here we have a standard deviation between 11.13 and 24.87). That is why we also used a programmatically approach to evaluate the crawlers: perplexity [8]. We used the same control points for the samples but we used larger samples for the calculation. Furthermore, we used a trigram language model based on the 1000 medicine train set's positive documents to calculate the perplexity. Figure 6.13 shows the evaluation results using perplexity and including out-of-vocabulary words. The perplexity is plotted on the y-axis (lower values are better).

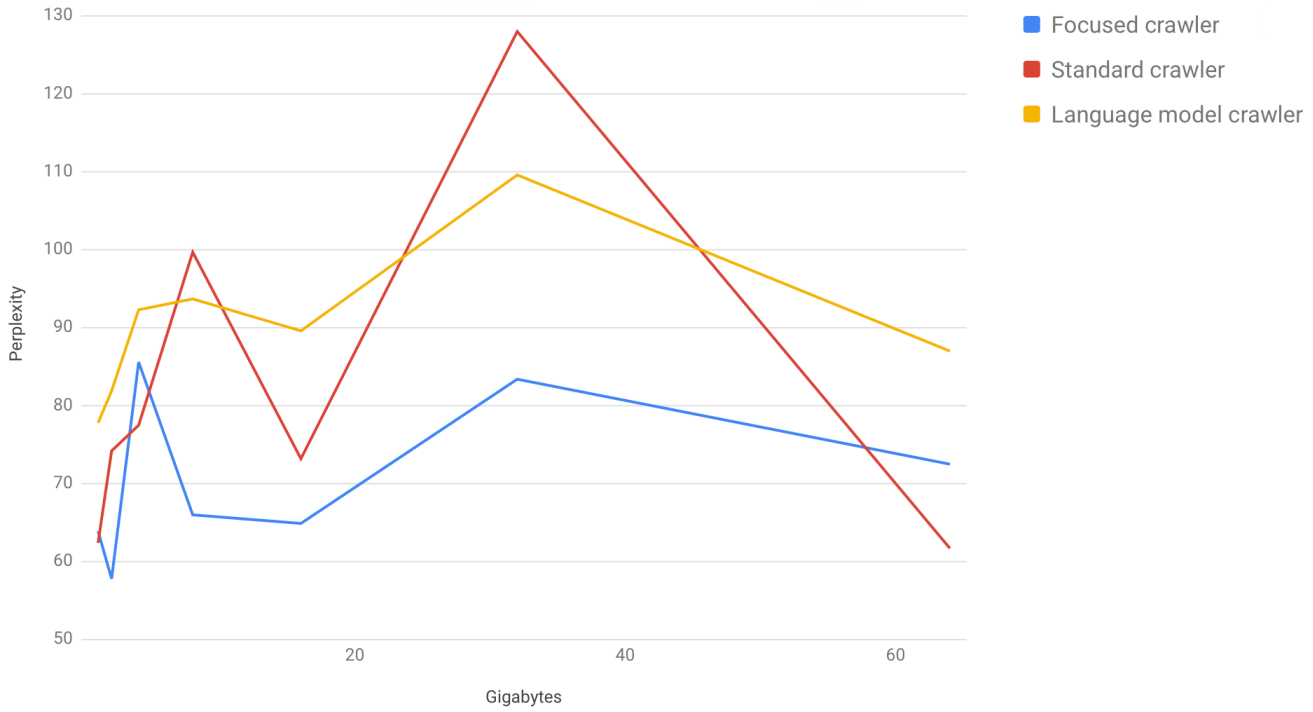

Figure 6.13: Medicine crawl's perplexity on seven samples based on an n-gram language model ($n = 3$) including out-of-vocabulary words.

The standard crawler's results are very agitatedly: They are between 62 and 127. In this evaluation, the implemented focused crawler beats the standard crawler and the language model crawler. The implemented focused crawler has a perplexity of about 75 and the other focused crawler of about 95.

Compared to the evaluation in Figure 6.12, this is a surprising result. But it can be explained by the documents in the samples. The found data can be separated into three parts: medicine/health, general science and other documents. Figure 6.12 shows that the samples of the language model crawler contain about 45% medicine/health data. The implemented focused crawler's samples contain about 38% medicine/health data, but also a large amount of general science data. So the data crawled by the implemented focused crawler contain more scientific speech.

Figure 6.13 shows the perplexity evaluation based on a trigram language model. Furthermore, we did not filter the out-of-vocabulary words. In contrast, Figure 6.14 shows the perplexity evaluation based on the same trigram language model but now, we filtered the out-of-vocabulary words.

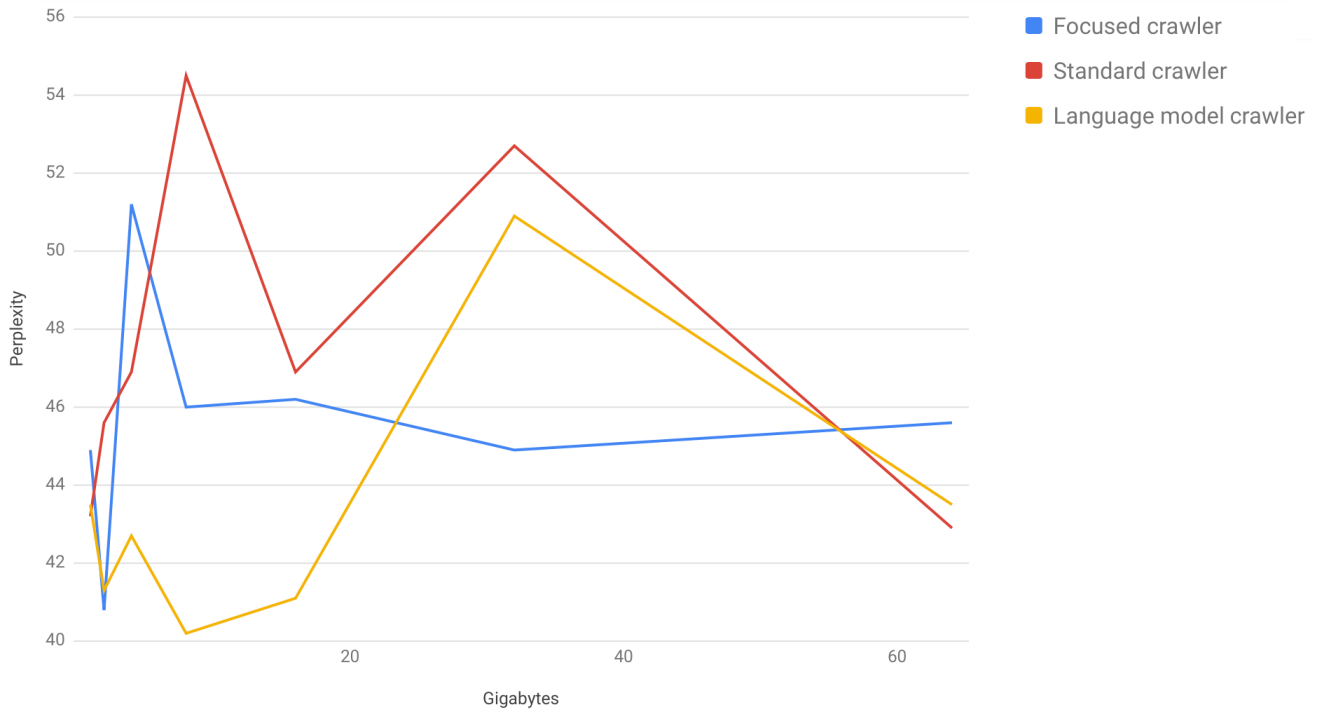

Figure 6.14: Medicine crawl's perplexity on seven samples based on an n-gram language model ($n = 3$) without out-of-vocabulary words.

Again, the standard crawler's results are very agitatedly but this time, the implemented focused crawler's and the language model crawler's values are below the standard crawler's results except some insignificant outliers. The standard crawler's perplexity is between 42.9 and 54.5. The implemented focused crawler has a perplexity of about 45 after eight gigabytes. The language model crawler's results are most of the time below the implemented focused crawler's results beside a large outlier after 32 gigabytes. The language model crawler's average perplexity is 43.3.

The focused crawlers for the finance and technology topics are based on a hyperlink classifier with more complex feature sets, compared to the medicine crawler. That slows down the focused crawler's

speed dramatically. To ensure, that it is not the classification service that slows down the crawler, we implemented the classification functions for finance and technology directly in Heritrix and all needed files for classification were in the main memory. Anyway, the focused crawler's average speed for the finance and technology topic is 8 kB/s. That is why we crawled only 1024 megabytes for these topics. Nevertheless, it is also interesting to analyze the focused crawlers' start. Figure 6.15 shows the results of the implemented focused crawler for the finance (left) and technology (right) topic compared to the standard crawler's results. We chose samples containing 100 documents from packets of 30 megabytes of HTML code after the 256th, 512th and 1024th megabyte.

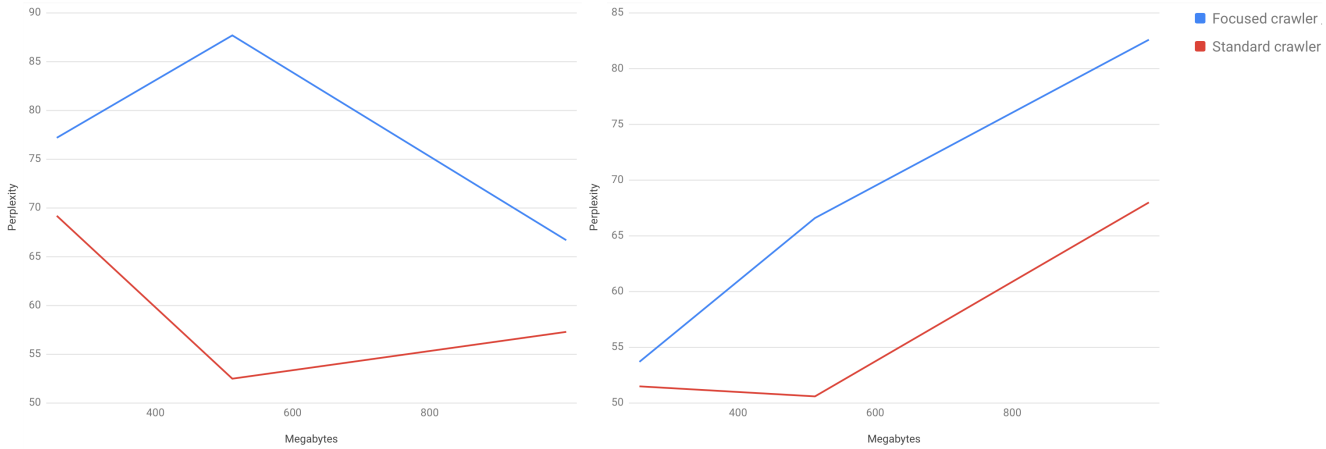

Figure 6.15: Perplexity of the finance (left) and technology (right) crawl on three samples based on an n-gram language model ($n = 3$) including out-of-vocabulary words.

On both topics, the perplexity of the focused crawler is higher than the standard crawler's perplexity in the complete first gigabyte. It is more or less the same on the medicine topic. So for a real evaluation of a focused crawler, we need more data than one gigabyte. However, we have to find an approach that is fast enough for a web crawl and works on all topics. Section 6.5.1 will present an evaluation of URL tokens with J48 as classification algorithm to solve this problem.

6.5.1 Performance with URL Tokens

The analysis in Section 6.5 has shown that complex classifiers slow down the focused crawler's speed. However, the hyperlink classifier for the medicine set works well, does not slow down the focused crawler and it is based primarily on the 406 most frequent URL tokens from the source document and it uses J48, which has its own feature subset selection, as classification algorithm. Furthermore, hyperlink tokens can be extracted very fast and it is not necessary to store the complete source document's HTML code with the URL in the priority queue, which wastes a lot of memory, for later classification.

Table 6.8 shows the evaluation results (accuracy, precision, recall and F1-measure) for hyperlink classifiers on different topics based on the 500 most frequent source and target URL tokens and J48 as classification algorithm.

At first glance, the results are comparable to the previously analyzed hyperlink classifiers based on the evolutionary algorithm for feature subset selection. The F1-measure is only slightly below the result reached by the use of the evolutionary algorithm for all three topics, e.g. there is a difference of 0.0418 on the finance topic and 0.0015 on the medicine topic.

Table 6.8: Results of the hyperlink classifier with URL tokens and J48.

Set	Accuracy	Precision	Recall	F1-measure
Finance	0.637	0.6341	0.6284	0.6312
Medicine	0.7684	0.7284	0.7869	0.7565
Technology	0.6891	0.6601	0.7286	0.6927

In conclusion, it is possible to use the most frequent source and target URL tokens in combination with J48, the most successful algorithm for hyperlink classification in our tests, as classification algorithm. The biggest advantages are the fast feature extraction and the reduced memory requirements. Additionally, it is not necessary to use the evolutionary algorithm for hyperlink classification then, but it was needed to find this feature set.

In this chapter we investigated a filtering approach based on an unary classifier (the LibSVM in its unary classification mode). However, it was not possible to filter a few positive documents from a mainly negative set to reach a high purity, with respect to the negative set. Nevertheless, we tried an approach based on k-means clustering to generate a pure negative set. Therefore, the k-means clustering algorithm separates the test data into k clusters. After that, we calculate the distances from the input data to every cluster and choose the cluster with the largest distance. This cluster can be used as a negative set. This approach works well and it was possible to generate a 100% pure negative set for every considered topic.

We used an evolutionary algorithm to find good feature sets and a classification algorithm for a binary classifier. We used 20 chromosomes per generation because of the number of possible feature sets and a crossover probability of 0.6. Furthermore, we tried different mutation probabilities between 0.01 and 0.03. It turned out that a mutation probability of 0.03 works best for the binary classifier. We reached an F1-measure over 0.9 for all considered topics.

The hyperlink classifier predicts if a hyperlink leads to a positive or a negative document with respect to the considered topic. We used an evolutionary algorithm to optimize this classifier. Here, we used 30 chromosomes per generation because of the larger pool of possible feature sets. We set the crossover probability to 0.6 and the mutation probability to 0.03. It turned out that the most frequent URL tokens are, in combination with J48 as classification algorithm, the best features for the hyperlink classifier. We reached an F1-measure over 0.75 for the medicine topic with the hyperlink classifier.

We analyzed the complete focused crawler in detail on the medicine topic. The focused crawler beats the non-focused crawler and is more or less on the same level as the language model crawler. The implemented focused crawler finds about 38% positive documents (on the medicine topic). However, complex classifiers slow down the focused crawler. Nevertheless, URL tokens can be easily extracted and with J48 as classification algorithm we reached F1-measure values between 0.63 and 0.76 on the considered topics with the hyperlink classifier. Thus, with URL tokens and J48 it is possible to build a fast focused crawler.

7 Conclusion

In this thesis we have shown a focused crawling approach which can be used as a first step of corpus generation. In contrast to Chakrabarti et al. [10], we do not maintain a large and detailed taxonomy to train and test a classifier which serves as a decision function for prioritizing URLs during a web crawl.

In our focused crawling approach the topic of interest is defined by the user's input documents. Chakrabarti et al. uses the topic of interest as the positive class and all other topics in their taxonomy as the negative class to train the classifier. In contrast, we use the user's input as positive set and have to build the negative set.

Therefore, we tried an approach using an unary classifier – the LibSVM in its unary mode – for filtering, but it was not possible to find a static feature subset which works on every of the considered topics. However, we investigated a different approach based on k-means clustering. We use the clustering algorithm to separate the test data into k clusters. After that, we calculate the distances from the input data to every cluster and choose the cluster with the largest distance. This cluster can be used as a negative set. In contrast to the LibSVM approach, the clustering approach works on all three considered topics.

So it is possible to make a short web crawl, starting from a general set of seed URLs, and filter the crawled data to generate a negative set. In addition to that, we have shown that it is also an option to use the general crawl's data without filtering because this data contains mostly negative data, with respect to a predefined positive topic.

The input data and the created negative set can be used to train a binary classifier. To optimize the feature subset for the classifier, we use an evolutionary algorithm. This algorithm is based on a special chromosome structure, containing feature sets, its parameters and a classification algorithm. Additionally, we use the F1-measure as fitness function. We use 20 chromosomes per generation, because of the number of possible feature sets, a crossover probability of 0.6 and we investigated different mutation probabilities. It turned out, that a mutation probability of 0.03 works well on our considered sets. The value is low enough to use the evolutionary algorithm's advantages and high enough to compensate the low number of chromosomes per generation.

The final binary classifier is used to build a train and test set for the hyperlink classifier, which is the focused crawler's decision function. Therefore, the binary classifier labels the target documents of the hyperlinks in the general crawl data. The hyperlink classifier is also based on the evolutionary algorithm for feature subset selection. Furthermore, it uses the feature sets from the binary classifier but also additional features especially for hyperlinks, e.g. URL tokens. URL tokens are extracted from a document's URL by splitting it at dots, hyphens and slashes. It turned out, that URL tokens work well as feature set for hyperlink classification. So it is also possible to use the hyperlink classifier with URL tokens without the evolutionary algorithm.

We investigated the complete focused crawler in detail for the medicine topic with a 64 gigabyte crawl. The crawlers decision function is based on the source document's URL tokens. The focused crawler beats the non-focused crawler's perplexity and is more or less on the level reached by a focused crawler based on a language model. However, it turned out that the focused crawler based on a language model finds more medicine related documents as the implemented focused crawler on a set of small samples counted by three independent annotators. The implemented focused crawler finds

about 38% positive documents and the focused crawler based on a language model finds about 45% positive documents.

For the investigated topics finance and technology, the evolutionary algorithm selected more complex feature subsets: That slowed down the complete focused crawler. So a focused crawler based on URL tokens is recommended because it is successful and much faster.

7.1 Recommended Architecture

Complex feature sets slow down the hyperlink classifier and the complete focused crawler. However, we have shown in this thesis that URL tokens work well in combination with J48 for hyperlink classification. In this section we will describe a focused crawling architecture based on this insights.

The user's input documents define the topic of interest (the positive class). After that we have to generate a negative set. This could be a predefined negative set or documents collected by a short general web crawl. However, we can use the negative set with or without filtering because of the natural purity of web data with respect to the negative set. For filtering, we can use the k-means clustering approach. We select the cluster with the largest distance to the positive data as negative set.

With the user's input and the generated negative set we can train a binary classifier. If a predefined static feature set, e.g. the most frequent tokens, works well with the LibSVM it is not necessary to use the evolutionary algorithm for feature subset selection. However, it can be used optionally to optimize the binary classifier's result.

The binary classifier is used to generate a train and test set for the hyperlink classifier. Therefore, a function iterates over a set of HTML documents (e.g. from a short standard web crawl), visits the hyperlinks of every HTML document and labels the target documents with support of the binary classifier. It is not necessary to store any HTML code in the train and test set because the hyperlink classification is based only on the URLs.

The hyperlink classifier uses the most frequent URL tokens from the source and the target URL. Furthermore, it uses J48 as classification algorithm. The crawler could be connected over REST to the hyperlink classifier or the hyperlink classifier could be implemented directly in Heritrix. Finally, the binary classifier can be used to filter the crawler's output.

7.2 Future Work

In the system implemented for this thesis, a lot of subsystems work together. We investigated different types of classification algorithms, many possible feature sets for binary and hyperlink classification and we used an evolutionary algorithm for feature subset selection. It would be interesting to investigate some parts of this thesis in more detail.

In this thesis we used the evolutionary algorithm to select appropriate features and a classification algorithm. For the classification algorithms we always used their default settings, but algorithms like the LibSVM or J48 offer a powerful set of possible parameter settings. So it could be interesting to use a fixed classification algorithm for the evolution and include the classification algorithm's settings in the chromosome structure in addition to the feature sets.

We found out that it is possible to filter small portions of positive data, with respect to the considered topic, from a negative set with an approach based on k-means clustering. Therefore, we use the

cluster with the largest distance to the user's input, which defines the positive set, as negative set. Even if it is possible to crawl large amounts of data quickly, this approach is a big waste of data because we filter up to 90% of the data. So it would be interesting to investigate if it is e.g. possible to detect further usable clusters in addition to the cluster with the largest distance to the user's input. Furthermore, it could be interesting to use different feature sets, distance metrics or clustering algorithms.

We have shown in this thesis, that URL tokens work well for hyperlink classification. Currently, we split the URLs at dots, hyphens and slashes to extract its tokens. This works well on URLs like `http://www.tu-darmstadt.de/`. Here, it is simple to extract the tokens, e.g. `tu`, `darmstadt` or `de`. However, it is difficult on URLs like `http://www.medicineonline.com` because with this approach it is impossible to extract the tokens `medicine` and `online`. During the implementation of the focused crawler we did not expect that the URL tokens work so well. Therefore, it would be interesting to investigate if an improved URL splitting technique will improve the results of the hyperlink classifier and the evaluation results of the complete focused crawler.

Bibliography

[1] Christine Abueg. IBM Watson To Improve Healthcare. 2015. http://www.healthaim.com/ibm-watson-improve-healthcare/18199 (checked 08/19/2015).

[2] Marco Baroni and Silvia Bernardini. BootCaT: Bootstrapping Corpora and Terms from the Web. In *In Proceedings of LREC 2004*, pages 1313–1316, 2004.

[3] Robert Battle and Edward Benson. Bridging the Semantic Web and Web 2.0 with Representational State Transfer (REST). *Web Semant.*, 6(1):61–69, February 2008. ISSN 1570-8268. doi: 10.1016/j.websem.2007.11.002. Amsterdam, The Netherlands.

[4] Tim Berners-Lee, MIT/LCS, Roy Fielding, UC Irvine, Henrik Frystyk, and MIT/LCS. RFC1945: Hypertext Transfer Protocol – HTTP/1.0. 1996. http://tools.ietf.org/html/rfc1945 (checked 07/01/2015).

[5] Tim Berners-Lee, W3C/MIT, Roy Fielding, Day Software, Larry Masinter, and Adobe Systems. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. 2005. https://tools.ietf.org/html/rfc3986 (checked 07/29/2015).

[6] Albrecht Beutelspacher and Ute Rosenbaum. *Projective Geometry: From Foundations to Applications*. Cambridge University Press, 1998. ISBN 978-0521483643. http://www.maths.ed.ac.uk/~aar/papers/beutel.pdf (checked 08/20/2015).

[7] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic Clustering of the Web. In *Selected Papers from the Sixth International Conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[8] Peter F. Brown, Vincent J. Della Pietra, Robert L. Mercer, Stephen A. Della Pietra, and Jennifer C. Lai. An Estimate of an Upper Bound for the Entropy of English. *Comput. Linguist.*, 18(1):31–40, March 1992. ISSN 0891-2017. Cambridge, MA, USA.

[9] Soumen Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kauffman, 2002. ISBN 1-55860-754-4. http://www.cse.iitb.ac.in/~soumen/mining-the-web/ (checked 07/29/2015).

[10] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused Crawling: A New Approach to Topic-specific Web Resource Discovery. In *Proceedings of the Eighth International Conference on World Wide Web*, WWW '99, pages 1623–1640, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[11] Soumen Chakrabarti, Kunal Punera, and Mallela Subramanyam. Accelerated Focused Crawling Through Online Relevance Feedback. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 148–159, New York, NY, USA, 2002. ACM. ISBN 1-58113-449-5. doi: 10.1145/511446.511466.

[12] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. Efficient Crawling Through URL Ordering. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 161–172, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[13] Apache UIMA Development Community. UIMA Overview and SDK Setup. 2011. http://uima.apache.org/d/uimaj-2.4.0/overview_and_setup.html (checked 08/22/2015).

[14] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-78019-5.

[15] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. University of California, Irvine.

[16] Association for Computational Linguistics. ACL SIGLEX Resource Links. `http://www.clres.com/corp.html` (checked 08/16/2015).

[17] Stanley Gotshall and Bart Rylander. Optimal Population Size and the Genetic Algorithm. University of Portland. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.105.2431&rep=rep1&type=pdf` (checked 07/27/2015).

[18] Cyril Goutte and Eric Gaussier. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *in: Proceedings of the 27th European Conference on Information Retrieval*, pages 345–359, 2005.

[19] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. SIGKDD Explorations, Volume 11, Issue 1. `http://www.cs.waikato.ac.nz/ml/weka/citing.html` (checked 09/24/2015).

[20] Eui-Hong Han, George Karypis, and Vipin Kumar. Text Categorization Using Weight Adjusted k-Nearest Neighbor Classification. 2001. `http://link.springer.com/chapter/10.1007/3-540-45357-1_9` (checked 08/20/2015).

[21] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. 1979. Journal of the Royal Statistical Society. Series C (Applied Statistics) Vol. 28, No. 1 (1979), pp. 100-108. Blackwell Publishing.

[22] IBM. Watson Services. . `http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/services-catalog.html` (checked 08/19/2015).

[23] IBM. Deep Blue. . `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/` (checked 08/19/2015).

[24] Paul Jack and Noah Levitt. Heritrix. 2014. `https://webarchive.jira.com/wiki/display/Heritrix/Heritrix` (checked 08/05/2015).

[25] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.

[26] Richard M. Karp. Reducibility Among Combinatorial Problems. *R. E. Miller und J. W. Thatcher, Complexity of Computer Computations*, 1972. New York, USA.

[27] Michael Kay and Saxonica. XSL Transformations (XSLT) Version 2.0. 2007. `http://www.w3.org/TR/xslt20/` (checked 07/02/2015).

[28] Adam Kilgarriff. Googleology is Bad Science. *Comput. Linguist.*, 33(1):147–151, March 2007. ISSN 0891-2017. doi: 10.1162/coli.2007.33.1.147. Cambridge, MA, USA.

[29] Bastian Kleineidam. LinkChecker – Check Websites for Broken Links. `http://wummel.github.io/linkchecker/` (checked 08/05/2015).

[30] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Addison-Wesley, 1998. ISBN 978-0201896855.

[31] Christian Kohlschütter, Peter Fankhauser, and Wolfgang Nejdl. Boilerplate Detection Using Shallow Text Features. In *Proceedings of the Third ACM International Conference on Web Search and Data Mining*, WSDM '10, pages 441–450. ACM, 2010. ISBN 978-1-60558-889-6. doi: 10.1145/1718487.1718542. New York, NY, USA.

[32] Martijn Koster and WebCrawler. A Method for Web Robots Control. 1996. `http://www.robotstxt.org/norobots-rfc.txt` (checked 07/29/2015).

[33] Sushil J. Louis and Gregory J. E. Rawlins. Predicting Convergence Time for Genetic Algorithms. In *Foundations of Genetic Algorithms 2*, pages 141–161. Morgan Kaufmann, 1992.

[34] Michael E. N. Majerus. Industrial Melanism in the Peppered Moth, Biston Betularia: An Excellent Teaching Example of Darwinian Evolution in Action. 2008. Springer Science + Business Media, LLC.

[35] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN 978-0521865715.

[36] John Markoff. Computer Wins on Jeopardy!: Trivial, It is Not. 2011. `http://www.nytimes.com/2011/02/17/science/17jeopardy-watson.html` (checked 08/19/2015).

[37] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. J. Wiley and Sons, Inc., 1990. ISBN 978-0-471-92420-3. New York, NY, USA.

[38] Microsoft. Example of xsl:include. 2012. `https://msdn.microsoft.com/en-us/library/ms256125.aspx` (checked 07/02/2015).

[39] Melanie Mitchell. An Introduction to Genetic Algorithms. *A Bradford Book The MIT Press*, 1996. Cambridge, Massachusetts, USA.

[40] Thomas Mitchell. *Machine Learning*. Mcgraw-Hill, 1997. ISBN 978-0-0711-5467-3.

[41] Paul Mockapetris and ISI. RFC 1034: Domain Names – Concepts and Facilities. 1987. `https://tools.ietf.org/html/rfc1034` (checked 07/29/2015).

[42] Naturalhistoryman. Peppered Moth Biston Betularia. July 2012. Image: `https://www.flickr.com/photos/naturalhistoryman/7502672010/in/photostream/` (checked 06/27/2015). Creative Commons License: `https://creativecommons.org/licenses/by-nc-nd/2.0/` (checked 06/27/2015).

[43] BBC News. IBM Supercomputer Watson Aims to Help Businesses. `http://www.bbc.com/news/technology-29226737` (checked 08/19/2015).

[44] Pierluigi Paganini. The Good and the Bad of the Deep Web. 2012. `http://securityaffairs.co/wordpress/8719/cyber-crime/the-good-and-the-bad-of-the-deep-web.html` (checked 07/28/2015).

[45] Gautam Pant. Deriving Link-context from HTML Tag Tree. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 49–55, New York, NY, USA, 2003. ACM. doi: 10.1145/882082.882094.

[46] Adriana Pietramala, Veronica L. Policicchio, Pasquale Rullo, and Inderbir Sidhu. A Genetic Algorithm for Text Classification Rule Induction. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *ECML/PKDD (2)*, volume 5212 of *Lecture Notes in Computer Science*, pages 188–203. Springer, 2008. ISBN 978-3-540-87480-5.

[47] Riccardo Poli and W. B. Langdon. Genetic Programming with One-Point Crossover and Point Mutation. In *Soft Computing in Engineering Design and Manufacturing*, pages 180–189. Springer-Verlag, 1997.

[48] Loebner Prize. Home Page of The Loebner Prize in Artificial Intelligence. `http://www.loebner.net/Prizef/loebner-prize.html` (checked 08/19/2015).

[49] Steffen Remus. Unsupervised Relation Extraction of In-Domain Data from Focused Crawls. In *Proceedings of the Student Research Workshop at the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 11–20, Gothenburg, Sweden, April 2014. Association for Computational Linguistics.

[50] Steffen Remus and Chris Biemann. Now Focus! N-Gram Language Models as Simple Means for Focused Web Crawling. Language Technology Group, TU Darmstadt. Unpublished.

[51] Jay Sampat, Anmol Jain, and Dharmeshkumar Mistry. Focused Web Crawler and its Approaches. 2014. Mumbai, India. `http://inpressco.com/wp-content/uploads/2014/09/Paper53121-31241.pdf` (checked 08/03/2015).

[52] Dominik Sobania. An Evolutionary Approach for Automatically Assigning Research Methods to Publications in the Domains of Social Sciences and Educational Research. 2013. UKP Lab, TU Darmstadt. Unpublished.

[53] Hunter Stern. Disposition Chain Processors. 2009. `https://webarchive.jira.com/wiki/display/Heritrix/Disposition+Chain+Processors` (checked 08/05/2015).

[54] Hunter Stern. Processing Chains. 2010. `https://webarchive.jira.com/wiki/display/Heritrix/Processing+Chains` (checked 08/05/2015).

[55] Hunter Stern. Fetch Chain Processors. 2011. `https://webarchive.jira.com/wiki/display/Heritrix/Fetch+Chain+Processors` (checked 08/05/2015).

[56] Mike Thelwall and David Stuart. Web Crawling Ethics Revisited: Cost, Privacy, and Denial of Service. *J. Am. Soc. Inf. Sci. Technol.*, 57(13):1771–1779, November 2006. ISSN 1532-2882. doi: 10.1002/asi.v57:13. New York, NY, USA.

[57] Alan. M. Turing. Computing Machinery and Intelligence. *Mind*, 59(236):433–460, 1950. ISSN 00264423. Oxford University Press on behalf of the Mind Association.

[58] UserAgentString.Com. List of All Crawlers. `http://www.useragentstring.com/pages/Crawlerlist/` (checked 07/28/2015).

[59] Jigang Wang, Predrag Neskovic, and Leon N. Cooper. Improving Nearest Neighbor Rule with a Simple Adaptive Distance Measure. *Pattern Recognition Letters*, 28(2):207–213, 2007. New York, NY, USA.

[60] Joseph Weizenbaum. ELIZA – A Computer Program for the Study of Natural Language Communication Between Man and Machine. *Commun. ACM*, 9(1):36–45, January 1966. ISSN 0001-0782. doi: 10.1145/365153.365168. New York, NY, USA. `http://web.stanford.edu/class/linguist238/p36-weizenabaum.pdf` (checked 08/19/2015).

[61] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-552-5.