POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

Imię i nazwisko studenta: Maciej Szankin
Nr albumu: 125769
Studia drugiego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Informatyka (studia w jęz. angielskim)
Specjalność/profil: Distributed applications and internet services

PRACA DYPLOMOWA MAGISTERSKA

Tytuł pracy w języku polskim: Klasyfikacja dokumentów tekstowych w środowisku równoległym

Tytuł pracy w języku angielskim: Text documents classification in parallel environment

Potwierdzenie przyjęcia pracy	
Opiekun pracy *podpis*	Kierownik Katedry/Zakładu *podpis*
dr inż. Mariusz Matuszek	

Data oddania pracy do dziekanatu:

GDAŃSK UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

Student's name and surname: Maciej Szankin
ID: 125769
Second cycle studies
Mode of study: Full-time studies
Field of study: Informatics
Specialization: distributed applications and internet services

MASTER'S THESIS

Title of thesis: Text documents classification in parallel environment

Title of thesis (in Polish): Klasyfikacja dokumentów tekstowych w środowisku równoległym

Supervisor *signature*	Head of Department *signature*
dr inż. Mariusz Matuszek	

Date of thesis submission to faculty office:

POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTRONIKI, TELEKOMUNIKACJI I INFORMATYKI

OŚWIADCZENIE

Imię i nazwisko: Maciej Szankin
Data i miejsce urodzenia: 18.08.1990, Gdynia
Nr albumu: 125769
Wydział: Wydział Elektroniki, Telekomunikacji i Informatyki
Kierunek: informatyka (studia w jęz. angielskim)
Poziom studiów: II stopnia
Forma studiów: stacjonarne

Ja, niżej podpisany(a), wyrażam zgodę/nie wyrażam zgody* na korzystanie z mojej pracy dyplomowej zatytułowanej: Klasyfikacja dokumentów tekstowych w środowisku równoległym do celów naukowych lub dydaktycznych.[1]

Gdańsk, dnia ....
podpis studenta

Świadomy(a) odpowiedzialności karnej z tytułu naruszenia przepisów ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. z 2006 r., nr 90, poz. 631) i konsekwencji dyscyplinarnych określonych w ustawie Prawo o szkolnictwie wyższym (Dz. U. z 2012 r., poz. 572 z późn. zm.),[2] a także odpowiedzialności cywilno-prawnej oświadczam, że przedkładana praca dyplomowa została opracowana przeze mnie samodzielnie.

Niniejsza(y) praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadaniem tytułu zawodowego.

Wszystkie informacje umieszczone w ww. pracy dyplomowej, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami zgodnie z art. 34 ustawy o prawie autorskim i prawach pokrewnych.

Potwierdzam zgodność niniejszej wersji pracy dyplomowej z załączoną wersją elektroniczną.

Gdańsk, dnia ....
podpis studenta

Upoważniam Politechnikę Gdańską do umieszczenia ww. pracy dyplomowej w wersji elektronicznej w otwartym, cyfrowym repozytorium instytucjonalnym Politechniki Gdańskiej oraz poddawania jej procesom weryfikacji i ochrony przed przywłaszczaniem jej autorstwa.

Gdańsk, dnia ....
podpis studenta

*) niepotrzebne skreślić

---

[1] Zarządzenie Rektora Politechniki Gdańskiej nr 34/2009 z 9 listopada 2009 r., załącznik nr 8 do instrukcji archiwalnej PG.

[2] Ustawa z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym:
Art. 214 ustęp 4. W razie podejrzenia popełnienia przez studenta czynu podlegającego na przypisaniu sobie autorstwa istotnego fragmentu lub innych elementów cudzego utworu rektor niezwłocznie poleca przeprowadzenie postępowania wyjaśniającego.
Art. 214 ustęp 6. Jeżeli w wyniku postępowania wyjaśniającego zebrany materiał potwierdza popełnienie czynu, o którym mowa w ust. 4, rektor wstrzymuje postępowanie o nadanie tytułu zawodowego do czasu wydania orzeczenia przez komisję dyscyplinarną oraz składa zawiadomienie o popełnieniu przestępstwa.

GDAŃSK UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS

STATEMENT

First name and surname: Maciej Szankin
Date and place of birth: 18.08.1990, Gdynia
ID: 125769
Faculty: Faculty of Electronics,Telecommunications and Informatics
Field of study: informatics
Cycle of studies: postgraduate studies
Mode of studies: Full-time studies

I, the undersigned, agree/do not agree* that my diploma thesis entitled: Text documents classification in parallel environment
may be used for scientific or didactic purposes.[1]

Gdańsk, ...
signature of the student

Aware of criminal liability for violations of the Act of 4th February 1994 on Copyright and Related Rights (Journal of Laws 2006, No. 90, item 631) and disciplinary actions set out in the Law on Higher Education (Journal of Laws 2012, item 572 with later amendments),[2] as well as civil liability, I declare that the submitted diploma thesis is my own work.

This diploma thesis has never before been the basis of an official procedure associated with the awarding of a professional title.

All the information contained in the above diploma thesis which is derived from written and electronic sources is documented in a list of relevant literature in accordance with art. 34 of the Copyright and Related Rights Act.

I confirm that this diploma thesis is identical to the attached electronic version.

Gdańsk, ...
signature of the student

I authorise the Gdańsk University of Technology to include an electronic version of the above diploma thesis in the open, institutional, digital repository of the Gdańsk University of Technology and for it to be submitted to the processes of verification and protection against misappropriation of authorship.

Gdańsk, ...
signature of the student

*) delete where appropriate

[1] Decree of Rector of Gdańsk University of Technology No. 34/2009 of 9th November 2009, TUG archive instruction addendum No. 8.

[2] Act of 27th July 2005, Law on Higher Education:

Art. 214, section 4. Should a student be suspected of committing an act which involves the appropriation of the authorship of a major part or other elements of another person's work, the rector shall forthwith order an enquiry.

Art. 214 section 6. If the evidence collected during an enquiry confirms that the act referred to in section 4 has been committed, the rector shall suspend the procedure for the awarding of a professional title pending a judgement of the disciplinary committee and submit formal notice of the committed offence.

Dedication

Abstract

The problem of text document categorization has became more complex than ever. With continuously increasing needs for storing huge amounts of data, need for quick access to data and automatic association also rises. Nowadays, document classification is used in wide range of application, varrying from knowledge databases, medical applications and spam filtering.

In this work algorithms of classification in parallel environment were designed, implemented and tested for processing time and recall defined as percentage of properly classified documents to all documents.

For clusterization a parallel implementation of k-means algorithm was introduced. The choice of k parameter and its influence on correctness of final results was also tested. Next, a multi-tier classification method was designed. First layer of classification consisted of kNN classifier, while second was based on SVM. Both methods were adapted to take advantage of available multi-core systems. kNN classifier was responsible for initial narrowing of possible labels for new document. Next, SVM algorithm was utilized to define final document class.

The main limitations of all methods were also highlighted in this work. They include time needed to communicate and send data between processes. The ways to overcome them are also discussed.

Initial results show that it is possible to classify a new document in satisfactory time interval of 6 seconds for a hundred of documents and scales nearly linearly on 10 core system. These values are indicators for the assumptions that this system can be a part of bigger platforms that cope with document classifications in real use cases. However, the results are only preliminary and should further tested in the future researches.

Abstrakt

Mimo wielu prowadzonych badań klasyfikacja dokumentów tekstowych pozostaje nadal skomplikowanym problemem. Wraz z wzrastającą potrzebą archiwizacji danych wzrosła również konieczność umożliwienia szybkiego i automatycznego dostępu do skategoryzowanych informacji. Obecnie klasifykacja dokumentów tekstowych jest używana w wielu profesjonalnych dziedzinach, od baz wiedzy poprzez aplikacje medyczne aż do filtrowania poczty elektronicznej w celu identyfikacji spamu.

W tej pracy algorytmy klasifykacji w środowisku rozproszonych zostały opracowane, zaimplementowane i przetestowane pod kątem czasu przetwarzania i specyficzności zdefiniowanej jako procent poprawnie sklasyfikowanych dokumentów spośród wszystkich klasyfikowanych dokumentów.

W celu klasteryzacji zastosowano wielowątkową implementację algorytmu k-means. Wybór parametru k oraz jego wpływu na poprawność rezultatów został również przetestowany. Następnie wielowarstwowa metoda klasyfikacji składająca się z dwóch algorytmów klasyfikacji kNN oraz SVM została opracowana. Oba algorytmy zostały zaprojektowane i zoptymalizowane do wykorzystywania wielordzeniowych procesorów. Klasyfikator kNN jest odpowiedzialny za zawężenie zbioru klas do których nowy dokument może zostać przypisany. Algorytm SVM, bazując na zawężonym zbiorze, podejmował precyzyjną decyzję do jakiej klasy dokument powinien należeć.

W pracy zaprezentowano także główne ograniczenia omówionych metod. Obejmują one czas potrzebny do komunikacji i przesyłania danych między procesami. Sposoby eliminacji opisanych problemów zostały również przedyskutowane.

Wstępne wyniki wskazują, iż klasyfikacja nowego dokumentu przy zbiorze 100 jest możliwa w satysfakcjonującym czasie (6 sekund). Dodatkowo, uzyskane rezultaty skalują się niemal liniowo dla wzrastającej liczby dokumentow przy systemach 10 korowych. Uzyskane wyniki pozwalają przypuszczą, iż zaprezentowane rozwiązanie może byś częścią wiekszej platformy, która klasyfikuje dokumentu w istniejących systemach. Jednakże uzyskane rezultaty są jedynie wstępne i powinny być dalej przebadane i przetestowane.

Contents

Chapter 1

Introduction

This chapter introduces the dissertation titled *"Text documents classification in parallel environment"* by presenting its context and motivation, project's objectives, hypothesis and an outline on sections 1.1, 1.2, 1.3, 1.4 respectively.

1.1. Context and motivation

The problem of text categorization is widely known in commercial world. Use cases include, for example, spam filtering, grouping articles and data organization in general. Thanks to text categorization it is possible to gain faster access to information because of usage of cataloged collections of documents [1]. Despite the fact that there have already been introduced numerous tools providing the possibility of document categorization, the problem of performance and efficiency still exists.

Taking it into consideration, the motivation for this work is to provide a framework for text document categorization in parallel environment. This innovative approach will allow to significantly reduce total time needed to parse a collection of documents and find relations between them. Using standard procedures this operation on a set of Wikipedia articles (4.9M of articles and growing [2] [2]) consumed weeks or even months. This research will try to prove that introducing parallelism will allow to shorten required time, thus creating applications more responsive to the end user.

1.2. Objectives

Project has two main objectives. First, is to design and implement multi-tier advisor for text document categorization, which will allow to test final solution application. Second, is to propose and implement framework that will make it possible to run different advisor's implementations in a parallel environment with data spanned across cluster of servers. Each of objectives is split into parts and described below.

1.2.1. Advisor

The first part of work is to prepare implementation of advisor for text document categorization. Advisor will be responsible for generating decision on class of given text document in relevance to

[2] Number of articles in English Wikipedia, https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia, access on 09.07.2015

known set of documents. It will be based on a tiered model with layers specified as follows:

1. First layer will be responsible for general decision that restricts the set of potential classes. It will be implemented using simple minimal distance kNN classifier.

2. Second layer will return more precise decision and will be implemented using more complex Support Vector Machine approach.

1.2.2. Parallel environment framework

The second part consists of designing and implementing a generic framework, which would allow users to run different classificators in parallel environment. Main goal of such system would be to provide means of speeding up total time of execution. This part includes designing multi-process system and scheduling methods for task distribution.

1.3. Hypothesis

As already mentioned, the aim of the project is to introduce framework that will allow to find relations between documents using combination of newest methods and trends of data distribution, parallel computation and machine learning algorithms. Such approach should demonstrate both efficiency and accuracy.

1.4. Outline

Document was divided into chapters and is structured as follows:

1. **Chapter 2** - Analysis of the state of knowledge in the field of: parsing text documents, methods of finding distance and relevance between them, machine learning algorithms that can be used to improve advisor's output decision, methods of distributing data across cluster and running scalable tasks on it, comparing and evaluating effectiveness of these methods.

2. **Chapter 3** - Methodology, choice of technologies, methods and algorithms for document categorization and distribution across cluster of nodes, analysis of possibilities of their realization in context of the project.

3. **Chapter 4** - Design of the system, aim of the project, potential users, system requirements.

4. **Chapter 5** - Description and implementation of the system.

5. **Chapter 6** - Development of the test scenarios, performing the tests and conducting experiments. Description of the achieved results.

6. **Chapter7** - Drawing conclusions from the results of work and possibilities of the future work.

Chapter 2

State of the art

2.1. Introduction

The aim of this chapter is to present the main goal that document classification should achieve. Each of required steps with a variety of available methods is shown and briefly described followed by real use cases. The chapter starts with introducing the concept of data mining in large data sets. In next section of this chapter methods of extraction best features that represent sample and allow to distinguish documents with each other will be discussed. In next two sections of this chapter concepts of data clusterization and classification will be introduced along with known algorithms. Fourth section deals with problems associated with documents classification performance, describing methods for distributing data across cluster of nodes and performing scalable tasks. Finally, taking into consideration all aspects that were raised in previous sections, the general concept of text document classification with associated problems will be explained.

2.2. Background

The manual pattern identification has been known since early 1700s, when Bayes' theorem was discovered [3]. However the increasing power of computer technology has lead to fast increment of data collection. Over the past decades, text documents became more and more accessible, because of digitalization of information, introducing strong need for efficient storage and fast information lookup [4]. Both of those needs have caused two questions to grow in importance - how to classify given document with certain confidence of accuracy in relevance to the set of other documents, and how to perform such task in acceptable time frame.

As digital document archives has grown in number and complexity, manual data analysis has been augmented with automatic algorithms of data processing aided by such discoveries as neural networks or decision trees. Application of these methods motivated by uncovering hidden patterns in dataset is known as data mining, an interdisciplinary subfield of computer science [5]. This process is an analysis step of the Knowledge Discovery in Databases, which aims at extracting patterns from dataset and transform them into structures that could be further used [6].

The process of Knowledge Discovery in Databases can be divided into following steps:

1. **Pre-processing** - it is necessary to assemble the target dataset before applying data mining algorithms. Preparing the target dataset is not an easy task because it must be large enough to contain patterns while remaining concise enough to be processed in acceptable time period.

In this step the target set is often cleaned to remove noise, missing data or data that does not provide valuable information.

2. **Data mining** - the actual process of pattern extraction, which involves:

   - **anomaly detection** - finding unusual data samples that require further investigation
   - **dependency modeling** - association rule learning to find relations between variables, for example *Apriori algorithm, decision tree*
   - **clustering** - known as *unsupervised learning* - finding groups of data without any supervision [7], , for example *K-means algorithm, K-nearest neighbour algorithm, Hierarchical algorithm*
   - **classification** - known as *supervised training* - training classifier with the labeled examples to create a set of rules or model that can search for feachures captured from a new sample and label it [7], , for example *Bayesian classifier, Naive Bayes*
   - **regression** - finding a function which can model the dataset with the least error
   - **summarization** - representation of the data

3. **Results validation** - the final step of Knowledge Discovery process is to verify whether discovered patterns can be applied to a wider dataset. It may turned out to be impossible to use some of the patters and reproduce results on a new sample. The process of using the model that is excessively complex and includes too many parameters in statistics and machine learning is known as over-fitting. A model that has been overfit may have poor predictive performance, as it exaggerates even small fluctuations of information. The accuracy of the model can then be measured from how many of new samples are correctly classify based on patterns learned from the training set. The binary classificator can also be evaluated by some statistical methods, for example the receiver operating characteristic, ROC.

2.3. Knowledge Discovery in Databases

2.3.1. Pre-processing

When designing a classifier for text documents, the feature extraction step is one of the most crucial to the final outcome. It should be build regarding the context that it is going to be used in, or it might render final solution to be inaccurate. One must remember that the choice of right feature can dramatically affect the achieved results and outcome. Feature extraction is most commonly used in signal and image processing [7]. In this case the features are purely mathematical concepts, like wavelet coefficients or geometric measures, which include perimeter, compactness, major and minor axes, eccentricity or Fourier Descriptors. In the presented case the problem is not so trivial. Many features might be unidentified or unnamed. However, it is very important to extract them, as they might have potential of improving classification or clustering significantly. In this step of Knowledge Discovery in Databases process, the target set is prepared and then, data mining algorithms are applied to it in order to find characteristic patterns. Some ways of cleaning and generalizing existing information are described below.

Bag of words

The process of lemmatization and stemming are the normalization techniques which allows to create a connection between base word and word modifications. This normalization is essential in various natural language processing (NLP) systems, for example text classification and information extraction, as it brings out actual grammatical or semantic meanings which are not accessible by the software [8].

1. **Stemming**

   Stemming is a process of getting word stem from its inflected and derived forms. The main algorithm criteria is the fact that the stem has to be identical to the morphological root. There are several types of stemming algorithms. Each of them differs in respect to accuracy and achieved performance.

   (a) **Lookup algorithms** - In this approach each of the inflected form together with related stem is collected in the structured form [9], a lookup table (def. 1). If a proper table is prepared, the algorithm is very efficient and fast. However the process of creating the table may be quite long and the table itself can be very large. Moreover for every new word it has to be updated.

   Definition 1. *Lookup table.*
   Lookup table - an array that replaces runtime computation with an array indexing operation

   (b) **Suffix-stripping algorithms** - The main goal of suffix stripping is to reduce the size and complexity of the data. As mentioned in [10] generally a document is represented by a vector of words. Suffix-stripping algorithm takes advantage of the fact that usually words with the same stem will have a similar meaning, for example: *connect, connected, connection*. Removal of these suffixes may lead to improving performance of document classification and clustering because of conflating groups of terms into a single term.

   There are several strategies and approaches to suffix stripping, for example using stem dictionary or suffix list. However defining a list of suffixes with various rules is correspondingly difficult. In some cases a combination of letters may really be a suffix, while in others it may be in fact a part of the stem and its removal is unhelpful, because the meaning of the word will be altered. One example that illustrates this problem is ER ending. Conflating words SAND and SANDER is correct, but for WAND and WANDER it is an error. That it way this problem is not so trivial and system of this kind can easily become very complex because of a need to specify many additional rules [10].

2. **Lemmatisation**

   The lemmatisation algorithm reduces inflectional forms of a word to a base/root form (or dictionary form known as *lemma*) by using vocabulary and morphological analysis of words. This process is similar, but not identical to stemming [8]. At first the part of speech of a word is determined, and then different normalization rules for each part of speech are applied. The algorithm is limited by the possibility to obtain the correct lexical category.

3. **Stochastic algorithms**

In this method, a probabilistic model is developed and used to identify the base form of a word. This model usually consist of a wide range of complex linguistic rules. Some lemmatisation algorithms may also be stochastic, because a word may belong to more than one parts of speech. In this case a probability is assigned to each possible part.

4. **Term Frequency, (Inverse) Document Frequency**

   This is a numerical statistic which specifies how important a word is to a document in a text corpus (def. 2). The tf-idf ratio increases proportionally to the number of occurrences of a word in a document and as a result may be used as a weighting factor [11].

   Definition 2. *Text corpus.*
   Text corpus - in linguistics a large set of documents

2.3.2. Data mining - Dependency modeling

Dependency modeling focuses on finding association rules using various measures of interestingness [12]. Association rule learning is a popular method for discovering interesting relations between variables. Generally the association rule is a rule that determines a data set that could be found in the database if the other data set exists in this database. In order to select interesting rules, some constraints have to be defined:

Definition 3. *Support.*
Support value of X is the proportion of transactions that contain a certain data set **X**

Definition 4. *Confidence.*
Confidence value of an association rule

$$\{X\} \Rightarrow \{Y\}$$

is the proportion of transactions that contain a certain data set **X** and also the other data set **Y**

Apriori algorithm

The Apriori algorithm helps to find which variables can depend on others. For example, the rule can look as follows

$$\{bread, cheese\} \Rightarrow \{butter\}$$

It means that if someone buys bread and cheese, he buys also butter. Such information can be used as the basis for decisions about marketing activities. The steps of the Apriori algorithm will be described using following example:

1. Consider following transactions gathered in a database presented in the table 2.5

Table 2.1. Apriori algorithm - example of transactions

Transaction	Items
1	{1,2,3,4}
2	{1,2}
3	{2,3,4}
4	{1,2,3}
5	{2,3}

2. The Apriori algorithm will be used to determine the frequent item sets of this database. We will assume that an item set is frequent if it appears in at least N transactions. This value (N) is the support threshold.

3. At first we will count the support of each item separately.

Table 2.2. Apriori algorithm - support of each item

Item	Support
{1}	3
{2}	5
{3}	4
{4}	2

4. Let the support threshold equals 3 (N=3). As a result all of the items, except item {4} are frequent.

5. In the next step we generate a list of pairs of frequent items.

Table 2.3. Apriori algorithm - support of each pair

Item	Support
{1,2}	3
{1,3}	5
{2,3}	4

6. All of the pairs meet or exceed the minimum support, so they are frequent. Now we can count support of triples.

Table 2.4. Apriori algorithm - support of each triple

Item	Support
{1,2,3}	2

7. Taking into consideration a result collected in the table 2.4, there are no frequent triples in the considered example.

FP-growth algorithm

The algorithm for finding frequent patterns is quite simple and includes two passes over the dataset [13].

- Pass 1

    1. support of each item is first counted and stored to 'header table'
    2. infrequent items are discarded

3. the FP-tree structure is created by inserting instances into it, items in each instance are sorted by descending order of their frequency

- Pass 2

1. read transactions and map them to a path in the tree

The construction of the FP-tree will be described using following example. Consider the same database reffp-alg as presented in the Apriori algorithm example.

Table 2.5. FP-growth algorithm - example of transactions

Transaction	Items
1	{1,2,3,4}
2	{1,2}
3	{2,3,4}
4	{1,2,3}
5	{2,3}

1. After reading first transaction

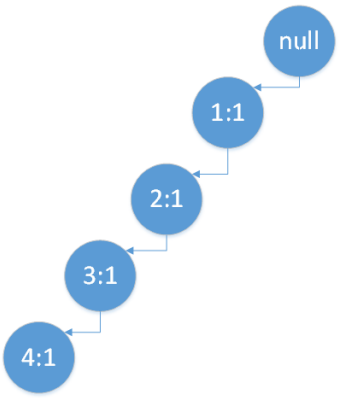


Figure 2.1. FP-growth example, after reading first transaction

2. After reading second transaction

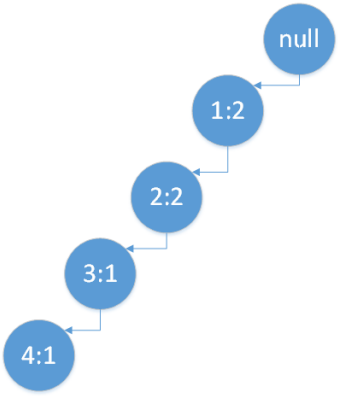


Figure 2.2. FP-growth example, after reading second transaction

3. After reading third transaction

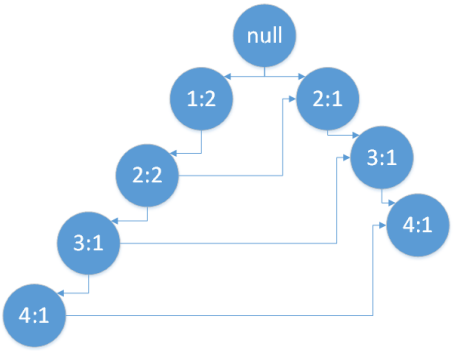


Figure 2.3. FP-growth example, after reading third transaction

4. After reading fourth transaction

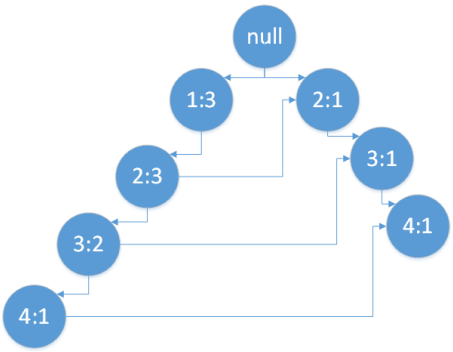


Figure 2.4. FP-growth example, after reading fourth transaction

5. After reading fifth transaction

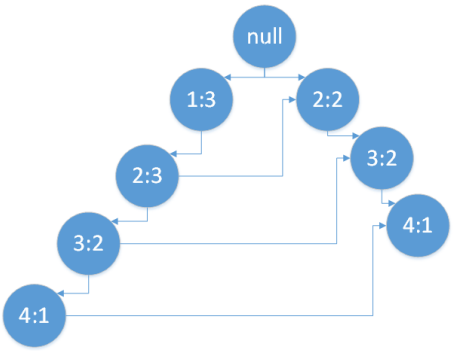


Figure 2.5. FP-growth example, after reading fifth transaction

2.3.3. Data mining - Documents clusterization

K-means algorithm

K-means is one of the most frequently used methods in data clustering. K-means and its derivatives have linear time complexity in relevance to the number of documents [14] This algorithm assumes that samples are divided into K groups. Its goal is to find the most optimal clusters based on this assumption. Each of clusters has its center and each pattern will be matched to cluster with center closest to matching the pattern. In algorithm, best centers are found in an iterative process.

K-means can be described using following steps and visual example:

1. Initialize k centers randomly, as presented in Fig. 2.6

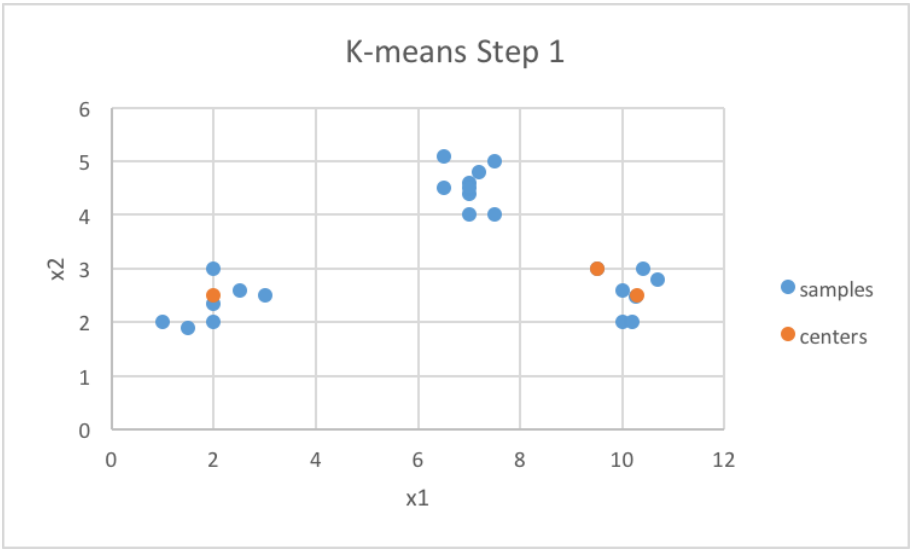


Figure 2.6. K-means algorithm, step 1

2. Find distances between each of centers and each of samples, as presented in Fig. 2.7

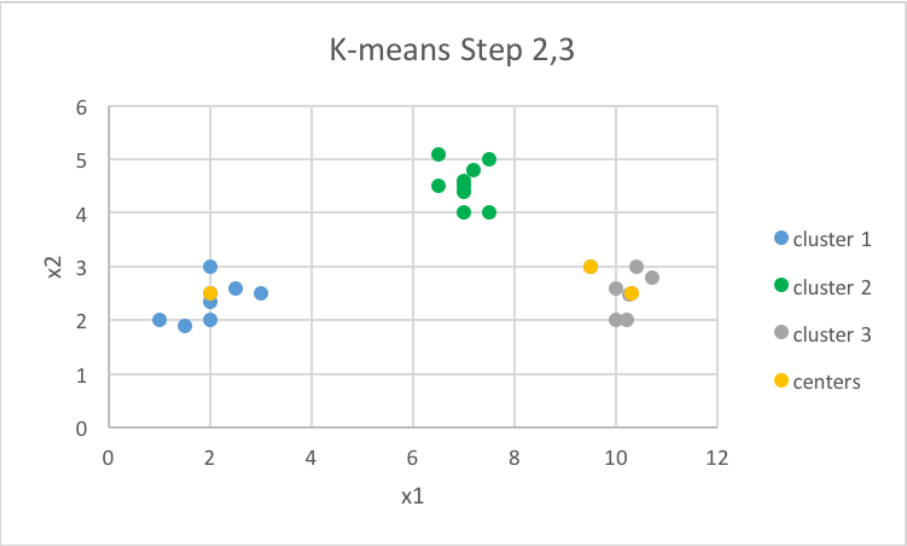


Figure 2.7. K-means algorithm, step 2

3. Assign samples to clusters, for whose calculated distance between sample and cluster's center is minimal, as presented in Fig. 2.8

4. Find new cluster centers. New cluster centers are the samples that are positioned closest to the average of all samples in its cluster, , as presented in Fig. 2.8

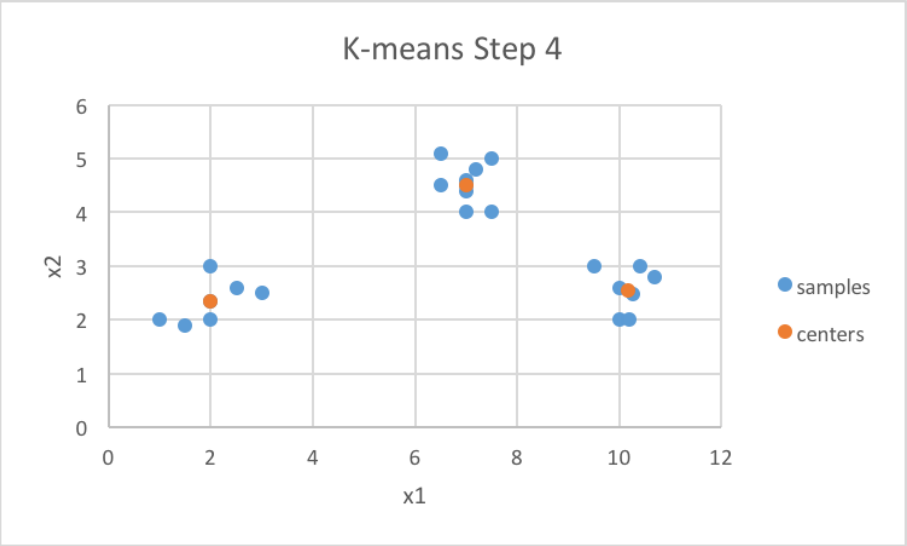


Figure 2.8. K-means algorithm, step 3 & 4

5. If during previous iteration none of the samples did not change it's cluster, go to next step. Otherwise go back to step 3.

6. Finish. Calculated clusters are the outputs of the algorithm, as presented in Fig. 2.9

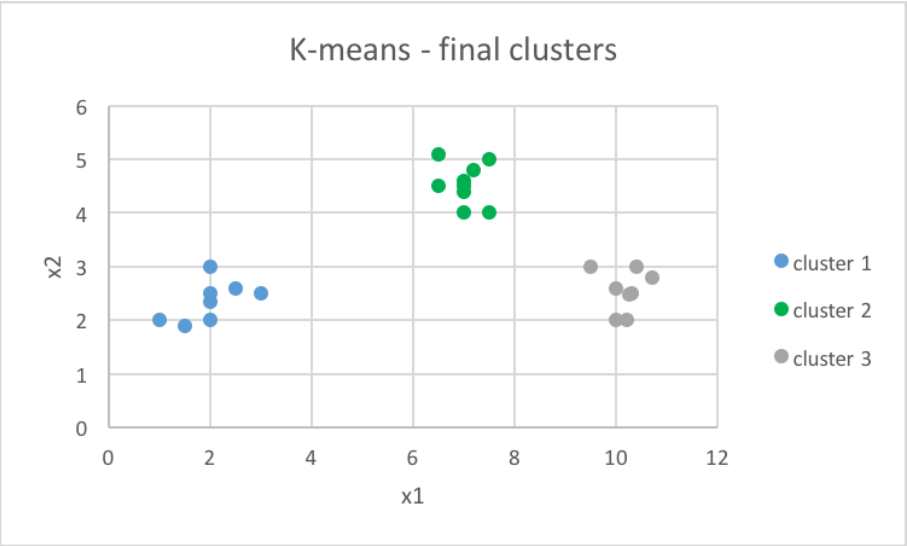


Figure 2.9. K-means algorithm, step 5

Hierarchical algorithm

Hierarchical clustering is a method which aims to build a hierarchy of clusters. It is often presented as the one that offers better quality, but its use is limited due to quadratic time complexity [14]. Hierarchical clustering goal is to produce a tree with single all-inclusive cluster as the root, that than divides into smaller, nested partitions with a single clusters at the bottom. Each of mid-points can be described as a combination of two clusters from lower level or as a part that was produced from splitting higher level. The output of such method can be visualized as a tree that is called dendogram (see Fig. 2.10). This clustering offers two approaches, that share mentioned principles:

- **Divisive** (top down) - start with single all-inclusive cluster. At each step divide it into two smaller clusters. With each step it is required to make a decision on which cluster to split and how to perform it.

- **Agglomerative** (bottom up) - start from single clusters, at each step finding the most similar or closest pair, merging it together to form upper intermediate level. Requires definition of similarity or distances of clusters.

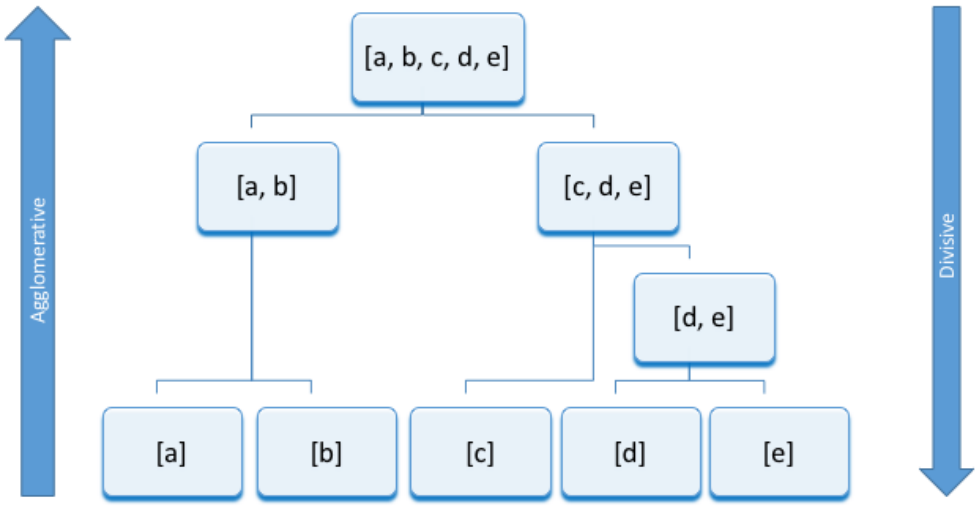


Figure 2.10. Hierarchical algorithm, dendogram example for divisive and agglomerative approach

2.3.4. Data mining - Documents classification

K-Nearest Neighbors algorithm

This section focuses on important classification algorithm called K-Nearest Neighbors (kNN). It is simple and efficient to use, as it does not require any labeling for data processing [15]. KNN is an non parametric lazy learning algorithm, because it doesn't make any assumptions on distribution of information and it doesn't contain any training phase or this phase is very minimal [16].

In KNN algorithm some assumptions are made:

1. Each training dataset consists of vectors labeled by a class differentiator.
2. Data is in feature/metric space, so it is a scalar or multidimensional vector. This assumption is necessary to be able to use a notion of distance [16].
3. It is necessary to specify a k number which determines number of neighbors that have influence on classification.

The algorithm of labeling a sample using kNN algorithm involves following steps:

- **Case 1, k=1**
  1. Let **A** be a point, which we want to label.
  2. Find a point **B**, that is closest to **A** taking into account the distance between **A** and **B**.
  3. Assign label of **B** to **A**
- **Case 2, k=k**
  1. We want to find k Nearest Neighbors and perform the majority voting.
  2. Let assume that we have k nearest neighbors of the point **A**, such as $k=2n+1$ and n instances belong to class 1 - **C1** and $n+1$ instances belong to class 2 - **C2**.

3. We want to label **A** sample.

4. The sample **A** will be labeled as **C2**, as instances from this group form majority.

All of the described steps are illustrated using figures 2.11, 2.12, 2.13. We have two labeled groups **C1** and **C2**.

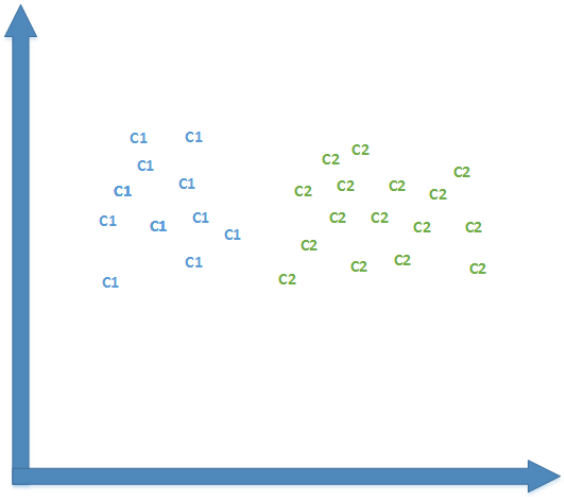


Figure 2.11. K-nearest neighbor algorithm, two labeled groups

We want to label a new sample **A**, so we find k nearest neighbors (let k equals 3).

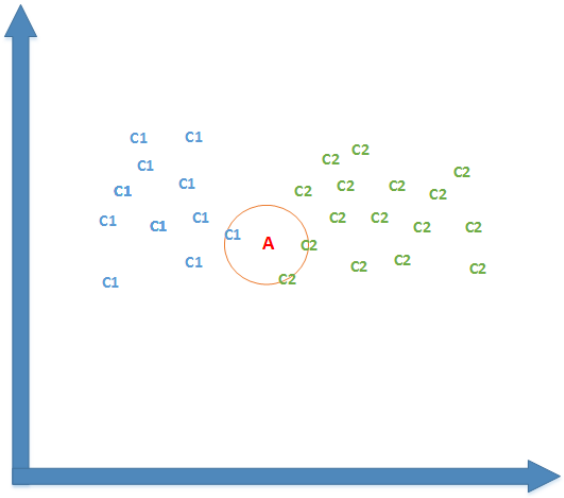


Figure 2.12. K-nearest neighbor algorithm, finding nearest neighbors

As can be seen instances from group **C2** forms majority of nearest neighbors, so the sample **A** is labeled as **C2**.

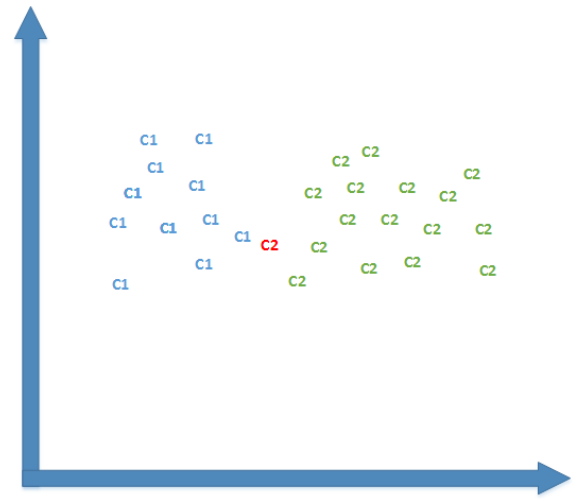


Figure 2.13. K-nearest neighbor algorithm, labelling sample

- **Case 3, weightened kNN** - in this case each point has a weight which is calculated using the distance to the sample, instead of giving all points weight equals 1.

Obviously this algorithm may result an error. However this statement holds only when a dataset is not big and the error turns out to be quite small, less than twice the Bayes error rate as described in [17]. One of the authors statements is to obtain a tight error bound to the Nearest Neighbor Rule:

$$P* \leqslant P \leqslant P * (2 - \frac{c}{c-1} P*)$$

where:

- $P*$ - Bayes error rate
- c - number of classes
- P - Nearest Neighbor error rate

Support Vector Machine model

The Support Vector Machine algorithm is a discriminative classifier which uses hyperplane to separate new examples. The main goal of the algorithm is to find the optimal hyperplane, yet the question is which parameters define it. After analysing a plot illustrated in the figure 2.14, it becomes obvious that a line that passes too close to points may make classification be noise sensitive. Taking it into consideration we can intuitively define a criterion of the best hyperplane: it should pass as far as possible from all points and then, samples that are closest to the hyperplane are called support vectors.

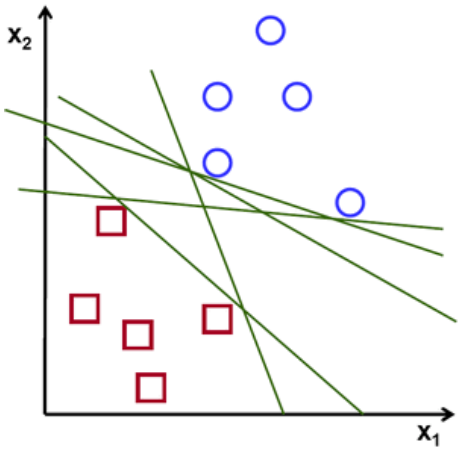


Figure 2.14. SVM algorithm - multiple lines which separate two classes - Cartesian plane instead of hyperplanes to simplificate the problem, source [18]

In SVM theory the distance between the optimal hyperplane and nearest points of both classes is known as margin. Therefore the optimal hyperplane leads to maximization of this margin (fig. 2.18).

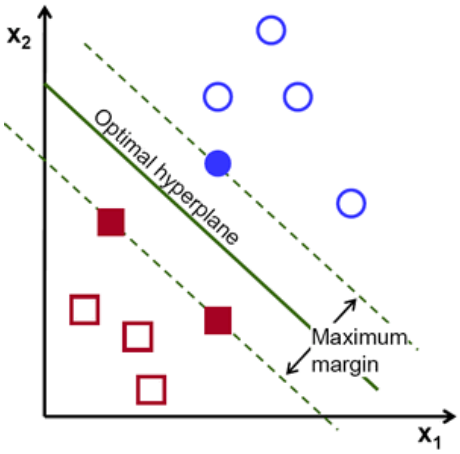


Figure 2.15. SVM algorithm - maximum margin, source [18]

Formally, a hyperplane may be defined as:

$$f(x) = \beta_0 + \beta^T x$$

where:

$$\beta - \text{weight vector}$$

$$\beta_0 - \text{bias}$$

Therefore the optimal plane may be formulated as:

$$\left|\beta_0 + \beta^T x\right| = 1$$

Multi-class SVM algorithm (MCSVM) can be implemented by combining binary SVM processes [19]. We can distinguish two strategies for MCSVM algorithm [20]:

1. One Against One - approach which focuses on creating one SVM for each pair of classes and train it to distinguish samples of one class from the others. Usually, classification of a new pattern is performed based on maximum voting, where each SVM votes for one class [20].

2. One Against All - approach which focuses on creating one SVM for each class and train it to distinguish samples of one class from the others. Usually, classification of a new pattern is performed based on the maximum output of all SVM. [20]

Taking into account some assumptions made by autors of [20] the "one against all" strategy seems to be more accurate than the "one against one". On the other hand although the training time of an SVM increases rapidly with the number of training samples, for a small number of training samples, it is easier to train SVMs in OAO algorithm.

Bayesian Classifier

This method is one of the most popular algorithms used for classification [7]. Bayesian theory evaluates cost of decision that were made to extract some rules from dataset. In this method one important assumption has to be taken into account: we assume that we know the probability distribution of groups and we choose the class whose *a priori probability* is larger. *A priori probability* is a main concept of Bayesian theory. Before classification we denote *a priori probability* of each class. Generally, such decisions are quite random and unpredictive. Moreover they ignore observations made from a system [7]. A Bayes classification process can be better explained using a real case example. Let assume that we have a picture with two areas that need to be distinguished. We can define *class conditional probability* of two classes:

$$P(x|\omega_i)$$

where X is a color vector. Defined probability measure guarantees that a given pixel which belongs to class i, will be in range of X. However normally we want to determine the class of selected pixel which has color X, so we want to count the *aposteriori probability* that a given pixel belongs to class i:

$$P(\omega_i|X)P(X) = P(X|\omega_i)P(\omega_i)$$

therefore:

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

For any pixel we have to calculate *aposterior probability* and choose a class which has the biggest value of this measure. Since value of *P(X)* is the same for all classes we can simplify the problem to analysing following condition:

$$P(X|\omega_1)P(\omega_1) > P(X|\omega_2)P(\omega_2)$$

Naive Bayes

Naive Bayes is a fast statistical classifier which is frequently used in spam detection systems [21]. The algorithm can be easily presented using message classification process. Let assume that a message is represented by a vector of values

$$x_1, ..., x_n$$

of atributes

$$X_1, ..., X_n$$

If a characteristic is present in the message we set its value *1* to it, otherwise value *0* is assigned. Then, to classify each candidate we compute *mutual information* - *MI* of each candidate attribute with category-denoting variable [**?**]:

$$MI(X, C) = \sum_{x \in 0,1; c \in C_1, C_2} P(X = x, C = c) log \frac{P(X = x, C = c)}{P(X = x)P(C = c)}$$

Then, we select attribute with the highest value of *mutual information*. Probabilities are estimated from the training dataset.

Neural Network

Neural networks are computional methods which allow to solve nonlinear classifications, because of their nonlinear nature. Being inspired by biological neural structures, they are composed of neurons connected through weights. First step focuses on training the network with given samples [7] in order to specify best weights. Once they are defined, the network may be used for classification.

2.3.5. Results validation

Evaluation of binary classifiers

The accuracy of the model can be evaluated by some statistical methods. The receiver operating characteristic is a graphical plot which presents the true positive rate *TPR* (def. 5) against the false positive rate *FPR* (def. 6).

Definition 5. *TPR.*
TPR - True positive rate, known as sensitivity in signal detection and biomedical engineering and recall in machine learning. It is a rate that measures the proportion of properly classified positive samples:

$$TPR = \frac{TP}{TP + FN}$$

where:

1. TP - True Positive, correctly identified
2. FN - False Negative, incorrectly rejected

Definition 6. *FPR.*
FPR - False positive rate, known as fall-out (1-specificity). Specificity is a rate that measures the proportion of properly classified negative samples:

$$FPR = 1 - \frac{TN}{TN + FP}$$

where:

1. TN - True Negative, correctly rejected
2. FP - False Positive, incorrectly identified

Some statistical measures of the performance of a binary classification test were collected in the table 2.6.

Table 2.6. Confusion matrix - statistical measures of the performance of a binary classification test

		Condition			
		Positive	**Negative**	**Prevalence**	
Test outcome	**Positive**	True Positive with hit	False Positive type I error, false alarm	Positive Predictive Value, Precision	False Discovery Rate
	Negative	False Negative false II error, with miss	True Negative correctly rejected	False Omission Rate	Negative Predictive Value
	Accuracy	True Positive Rate Sensitivity, Recall	False Positive Rate Fall-out	**Positive Likelihood Ratio**	**Diagnostic Odd Ratio**
		False Negative Rate Miss Rate	True Negative Rate Specificity	**Negative Likelihood Ratio**	

Where:

1. **Accuracy, ACC**
$$ACC = \frac{TP + TN}{TP + TN + FP + FN}$$
2. **True Positive Rate, Recall, TPR**
$$TPR = \frac{TP}{TP + FN}$$
3. **False Negative Rate, Miss Rate, FNR**
$$FNR = 1 - TPR = \frac{FN}{TP + FN}$$
4. **False Positive Rate, Fall-out, FPR**
$$FPR = 1 - TNR = \frac{FP}{TN + FP}$$
5. **True Negative Rate, Specificity, TNR**
$$TNR = \frac{TN}{TN + FP}$$

6. **Positive Predictive Value, Precision, PPV**

$$PPV = \frac{TP}{TP + FP}$$

7. **False Omission Rate, FOR**

$$FOR = 1 - NPV = \frac{FN}{TN + FN}$$

8. **False Discovery Ratio, FDR**

$$FDR = 1 - PPV = \frac{FP}{TP + FP}$$

9. **Nageative Predictive Value, NPV**

$$NPV = \frac{TN}{TN + FN}$$

10. **Positive Likelihood Ratio**

$$LR^{+} = \frac{TPR}{FPR}$$

11. **Negative Likelihood Ratio**

$$LR^{-} = \frac{FNR}{TNR}$$

12. **Diagnostic Odd Ratio**

$$DOR = \frac{LR^{+}}{LR^{-}}$$

Then, the trade-off between the measures can be presented using TPR and FPR by plotting them on receiver operating characteristic curve (example of this curve is presented in the figure 2.16).

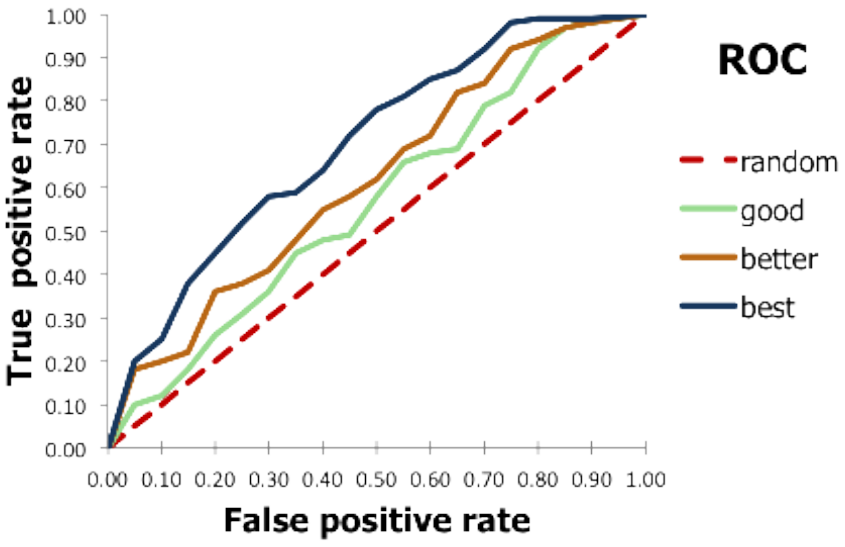


Figure 2.16. Example of ROC curve, source [22]

Evaluation of multiclass classificators

In order to evaluate multiclass classificators, the ROC curve may also be used. As in several multi-class problems, the idea is to perform pairwise comparison (one against one, one agaist all) [23].

2.4. Parallel processing and scalability

Although classification is well-studied data mining problem, there are still some issues that limits its usability There are two main approaches to scaling:

1. **Horizontal scaling** Horizontal scaling, also referred to as scale out, is the ability to increase computation performance by adding more machines into pool of resources. In database context horizontal scaling refers to partitioning data across nodes, so each node contains part of it.

Figure 2.17. Scale out method, source [24]

Horizontal scaling is the easiest way to achieve performance improvements in large scale. It often allows to add more machines to the cluster without affecting existing nodes, resulting in minimal downtime.

With horizontal scaling comes also important feature from cloud computing - high availability. Cluster configured to work in high availability mode multiplies services and data so they meet replication level requirement. With this approach, if one node fails, responsibility for serving lost data and failed services is handed over to another node, allowing to keep downtime even more minimal.

Finally, with scaling out there is possibility to scatter clusters across different data centers or geographical locations. It allows to handle unforeseen natural catastrophes without compromising your application.

2. **Vertical scaling** Vertical scaling is the ability to increase computation performance by adding more CPUs/cores or memory to existing machine.

Figure 2.18. Scale up method, source [24]

Vertical scaling is the easiest way to achieve significant performance improvements in small scale, scaling usually involves downtime and comes with an upper limit.

2.5. Existing solutions

With development of new technologies different means of using artificial intelligence to deal with real world problems became more and more common, creating a niche for providing such services. Despite the fact that those solution are often not free to use, they might sometimes turn out to be beneficial than creating own solution from scratch. Designing own classification system and developing it might be very time consuming, rendering product planning decisions more crucial.

2.5.1. Machine learning

Amazon AWS Machine Learning

Amazon is said to be the biggest cloud services provider on the market. It has overwhelming market share leadership, with over 10 times more infrastructure available on demand than the aggregate total of the other 14 providers [25]. Recently tools for applying machine learning were added to their portfolio, allowing end-users to take advantage of scalable environment which in combination with machine learning design system with high-level of abstraction allows to efficiently design and deploy application.

2.6. Use cases

Development was always driven by arising needs, and so classification development has few different fields which are the forces that push it ahead. This section illustrates the most noteworthy areas and briefly discusses source of those needs and most common use cases.

2.6.1. Data mining

1. **Business** Business has always been main motivating force behind nearly all modern developments. Current world, being based on information, requires from everyone to process knowledge in increasing speed and the business is no different. Modern companies usually

own their data warehouses, that are required to store more and more data in exploding rate. It has become impossible to analyze trends with standard means [26].

Data mining applied in this category helps analyze historical business activities with a goal to reveal hidden pattern and trends. It allows to go through large amounts of data and discover previously unknown strategic business informations. Examples of scenarios in which data mining can be used include performing market analysis to identify new product markets, to prevent customer attrition and targeting customers with more accuracy.

2. **Science** Data mining has become increasingly popular in fields of science and engineering. It allows to analyze huge amounts of data and retrieve key informations, that may occur crucial.

   One of most important examples is the study of human genetics. Sequence mining helps to understand the relation between single DNA sequences with probability of developing common diseases like cancer. It helps to diagnose such cases, allowing to start treatment as soon as possible, and sometimes even to prevent it completely [27].

   Another worth mentioning example is the area of power engineering, where data mining methods have been widely adapted for condition monitoring of high voltage electrical equipment. Condition monitoring itself supplies information, while data clustering techniques, such as self-organizing map, allow to detect abnormal conditions and denominate possible nature of the abnormalities [28].

Other use cases cover such aspects as filtering unsolicited bulk e-mail [21].

2.7. Conclusion

This chapter reviewed a number of issues and concerns that might have risen through out the discussion about performing scalable text document classification. Different methods of classification and clusterization were discussed, including kNN and SVM. The state of the art presented in this section provides a complete source of knowledge required to implement application of this kind. Moreover, restrictions mentioned previously should be taken into account. The state of the art presented in this chapter was prepared in reference to the hypothesis of the project.

Chapter 3

Methodology

3.1. Introduction

In the first section of this chapter the significance of a project is briefly explained. Then, according to it, the work plan is presented in the second section. In next sections designed algorithms are described taking into consideration the possibility to provide a framework for text document categorization in parallel environment.

3.2. Significance

Data mining algorithms play an essential role not only in business and science applications but also in medicine as second-opinion diagnostic tools. On the other hand, nowadays text documents become more and more accessible, because of information digitalization. Taking it into consideration there is a strong need to find a tool which allows to speed up information lookup by classifying given document with certain confidence of accuracy in relevance to the set of other documents. This solution based on using data mining algorithms in parallel environment can doubtless have a wide range of applications. The motivation for this work is to provide algorithms for documents features extraction, documents clasterization and classification. Next step is to scale this solution into application based on parallel processing. This combination may lead to previously undiscovered system that allows to accelerate document classification significantly. Many new aspects analyzed and defined in this paper may turned out to be important in the future researches and significant for development in many areas, for example:

1. spam filtering
2. email routing
3. language identification

3.3. Work plan

According to results that should be achieved, the plan of the project is divided into following parts:

- **Task 1** Implementation and validation of algorithms for feature extraction

- **Task 2** Development and validation of algorithms for documents clusterization and specyfing documents groups based on testing data set
- **Task 3** Development and validation of algorithms for new document classification
- **Task 4** Analyzing collected data and validating accuracy of used methods
- **Task 5** Development and validation of algorithms for parallel processing
- **Task 6** Analyzing collected data and validating accuracy of parallel processing algorithms

3.4. Methodology

3.4.1. Feature extraction

For human beings, documents are an essential mean of both preserving and exchanging knowledge. Those documents store different kinds of information using written form of language. Document consists of sentences, words, notes linking one article with another, marking relation between them. While it gives us, humans, the most common way of storing information, it also rises numerous problems for machines to process and understand informations stored in such way. To overcome this issue, a number of solutions were introduced.

This section summarizes methods that has been used or considered to be used for the best feature extraction that could possibly lead to more accurate clusterization and classification.

Tokenization

First step in feature extraction process is to tokenize document. Tokenization means that ordered sentences are turned into a collection of unordered words. During this step it is possible to reduce the overall weight of corpus by deleting stop words (def. 7), all special signs and converting all characters to lower case - they do not carry any information that would be considered important during clusterization and classification.

Definition 7. *Stop words.*
In documents analysis, stop words are words which are filtered out before or after processing of natural language text. They usually refer to the most common words in a specific language, so creating universal list of stop words is quite challenging.

During tokenization there is possibility of losing too much information, though. Some set of words are not supposed to be divided into seperate, unordered pieces. Consider following sentance as an example:

"*A popular tourist destination, San Francisco is known for its cool summers.*"

Tokenizing above sentence results in:

['a', 'popular', 'tourist', 'destination,', 'san', 'francisco', 'is', 'known', 'for', 'its', 'cool', 'summers'].

As can be seen, *'San Francisco'* has beed divided into two parts - *'san'* and *'francisco'*. In this particular case tokenization caused additional loss of information, as 'san' and 'francisco', when presented seperatly, might have another meaing. To bypass this issue, n-grams can be used.

n-grams

n-grams posses some advatnages over tokens. Tokens hold information about single word, while n-grams are a sequence of N >=1 consecutive words that appear in text. An n-gram of size 1 is referred to as a unigram, size 2 is a bigram (or a digram), size 3 is a trigram. Example presented above turned into n-grams with N=3 will result in:

[['A', 'popular', 'tourist'], ['popular', 'tourist', 'destination,'], ['tourist', 'destination,', 'San'], ['destination,', 'San', 'Francisco'], ['San', 'Francisco', 'is'], ['Francisco', 'is', 'known'], ['is', 'known', 'for'], ['known', 'for', 'its'], ['for', 'its', 'cool'], ['its', 'cool', 'summers']]

As can be seen, resulted trigram is a contiguous sequence of 3 items. An n-gram model is a type of probabilistic language model for predicting the next item in text sequence. These models have recently gained a lot of popularity for using them in probability, communication theory, computational linguistics and data compression.

Stemming and lemmatization

Important step in feature extraction process is word stemming and lemmatization. They both allow for derived words that might be in inflectional forms (def. 8) and sometimes derivationally related forms of a word to be brought to a common root (def. 9).

Definition 8. *Inflectional form.*
In grammar, inflection or inflexion is the modification of a word using a prefix, suffix or infix, or another internal modification such as a vowel change. The aim is to express various grammatical categories for example tense, case, voice, aspect, person, number, gender, and mood.

Definition 9. *Root word.*
A root word is a word that does not have a prefix (before the word) or a suffix (after a word). The root word is the primary lexical unit of a word. It defines a word family. A root is often called base word, which carries the most significant aspects of semantic content and cannot be reduced into smaller constituents. Some common root words, their meanings and words that are formed from this blocks are presented below:

- Act - move or do - action, activity, transaction
- Ambul - move or walk - amble, ambulant
- Auto - self or same - automate, automatic
- Cardio - heart - cardiology
- Cede - go - exceed, accessible
- Counter - against or opposite - counteract, counterpoint

- Demo - people - democracy, demographic
- Derma - skin - dermatology, epidermis
- Equi - equal - equity
- Semi - half - semicircle

Stemming is an approach based on different algorithms for calculating root of a given word. It consist of methods like lookup tables (def. 10), suffix (def. 11) and postfix (analogically) stripping algorithms. While this method is efficient, its accuracy suffers. Root might not always be a lexically correct form, but it still enables both counting occurrences of all forms of given word across the document and helps to reduce the overall weight of corpus.

Definition 10. *Lookup table.*
In computer science, a lookup table is an array that allows to reduce runtime computation by using a simpler array indexing operation. Shortening of processing time can be significant, as retrieving a value from memory is often faster than undergoing an computation or input/output operation. The table can be precalculated and stored in static program storage, calculated during a program's initialization phase (memoization), or even stored in hardware. The advantages of the lookup table is that it is simple, fast, and can handle exceptions. The disadvantages include large size of a table and possibility of loosing some words. All inflected forms must be explicitly listed in the table and sometimes new or unfamiliar words are not handled, even if they are perfectly regular.

Definition 11. *Suffix stripping.*
Suffix stripping algorithms rely on a smaller list of "rules" than a lookup table (a lookup table consists of inflected forms and root form relations). This shorter list used in suffix stripping provides a path for the algorithm to find its root form from an input form. Some examples of the rules are presented below:

- if the word ends in 'ed', remove the 'ed'
- if the word ends in 'ing', remove the 'ing'
- if the word ends in 'ly', remove the 'ly'

On the other hand, stripping algorithms have been shown to have a poor performance when dealing with exceptional relations (for example 'ran' and 'run'). The results given by suffix stripping algorithms are limited to those which have well defined suffixes. This is a problem, as not all words have a well formulated set of rules. Lemmatisation (def. 12) attempts to improve upon this challenge. It is based on different dictionaries to find a root of a word. The difference between lemmatization and stemming is that a stemmer operates on a single word without any context and without knowledge about part of a speech. On the other hand, stemmers are easier to implement and this algorithm is usually faster.

For example:

- The word worse has lemma bad. This relation is not found by stemming and it requires a look-up table.
- The word walk is the root form of walking, and it can be matched in both stemming and lemmatisation.
- Depending on a context, the word meeting can be either the root of a noun or a form of a verb. For example 'in our last meeting' and 'they are meeting tomorrow'. In this case lemmatisation can select the appropriate lemma depending on the context, while stemming can't.

The most common dictionary used in lemmatization process is WordNet.

Definition 12. *Lemmatization.*
In linguistics, a lemmatization is the process of grouping together the various inflected word forms so they can be analysed as a single item. In other words, it is a process of determining the lemma (def. 13) for a given word.

Definition 13. *Lemma.*
A lemma is the canonical form, dictionary form, or citation form of a word. For example the verb 'to walk' may be presented as walk, walked, walks, walking. The base form walk, which might be looked up in a dictionary, is called the lemma. The combination of the base form with the part of speech is called the lexeme. Another example include run, runs, ran and running are forms of the same lexeme, with run as the lemma.

Term Frequency and Inverse Document Index

In order to define document's topic it is essential to find relevant keywords. To achieve this, TF-IDF term weighting algorithm can be utilized. It is a numerical statistic method which reflects how important is a given word to a document in corpus. Value calculated by this method increases proportionally to the number of times a given word appears in a document, but is reduced by the frequency of the word in corpus. This approach allows words that are important for one document and appear frequently in it but rarely in corpus to score higher than ones that appear frequently in whole corpus.

Consider following sentences and matrix of tokens as an examples for further presentation of TD-IDF (tokens were lemmatized and stopwords were removed, for the sake of simplicity of visualization n-grams were not used):

Table 3.1. TF-IDF - Example documents

	Content	Terms
Document 1.	The sky is blue.	sky blue
Document 2.	The sun is bright.	sun bright
Document 3.	The sun in the sky is bright.	sun sky bright
Document 4.	We can see the shining sun, the bright sun.	sun see bright shining

- **TF - Term Frequency** - is a measure of frequency with which given word occurs in document. Because every document is different in length and thus longer documents will contain words with higher occurrence count, value returned by TF is usually normalized. Words that occur more often in a document have higher value than those that do not.

  As can be observed in 3.2, matrix created in a process tends to contain many empty cells (word does not occur in a document). It might be worth to consider using sparse matrix implementation in order to preserve memory.

$$TF_{\text{word}} = \frac{word\ occurence\ count}{total\ words\ in\ document}$$

Table 3.2. TF - Words frequencies

	blue	sun	sky	see	bright	shining
Document 1.	1		1			
Document 2.		1			1	
Document 3.		1	1		1	
Document 4.		2		1	1	1

Table 3.3. TF - Normalized words frequencies

	blue	sun	sky	see	bright	shining
Document 1.	0.5		0.5			
Document 2.		0.5			0.5	
Document 3.		0.333	0.333		0.333	
Document 4.		0.4		0.2	0.2	0.2

- **IDF - Inverse Document Frequency** - it can be observed that certain words that do not carry much information appear very frequently across all documents written in English. Example of these words are: 'a', 'an', 'the' and many others. But because not only articles and stop words can turn out to be redundant, IDF was introduced to counter the emphasis that these words carry with them. IDF diminishes the weight of words that occur frequently and increases the weight of words that occur rarely across all documents in corpus.

Table 3.4. Terms with IDF values assigned

Term	IDF
blue	2.386
sun	1.288
sky	1.693
see	2.386
bright	1.288
shinning	2.386

$$IDF_{\text{word}} = 1 + log_e \frac{total\ number\ of\ documents\ in\ corpus}{number\ of\ documents\ with\ word\ in\ it}$$

- **TF-IDF - Term Frequency-Inverse Document Frequency** Final result for TF-IDF is achieved by multiplying TF and IDF values.

$$TFIDF_{\text{word}} = TF_{\text{word}} * IDF_{\text{word}}$$

For the documents presented above, TF-IDF values will be:

Table 3.5. Final result of TF-IDF

	blue	sun	sky	see	bright	shining
Document 1.	1.193		0.847			
Document 2.		0.644			0.644	
Document 3.		0.429	0.429		0.564	
Document 4.		0.515		0.477	0.258	0.477

Cosine similarity

The documents clustering can result in faster data analysis, such as information retrieval and information extraction, by grouping similar kind of information. While performing clusterization, a similarity between a pair of objects has to be defined. Accurate clustering requires a definition of it in terms of either similarity or distance. Different similarity and distance measures have already been described and used, for example Euclidean distance or cosine similarity. Cosine similarity is one of the most popular similarity measure and it is often applied for text documents clusterization. The main advantage of the cosine similarity is its independence of document length. For two documents d_i and d_j, the cosine similarity between them can be defined as

$$cos(d_i, d_j) = \frac{d_i \cdot d_j}{||d_i||||d_j||}$$

where $d_i \cdot d_j$ is a *dot product* (def. 14) and $||d_i||||d_j||$ is an *Euclidean norm* (def. 15) of the vector (the intuitive notion of length of the vector).

As can be expected, documents are identical if value of cosine similarity is 1, and if the document vectors are orthogonal to each other this value is 0 and it means that documents are totally different.

In proposed solution this metric could be used to define a level of similarity between a pair of objects in order to group together nearest documents.

Table 3.6. Cosine similarity calculated between documents in example set

	Document 1	Document 2	Document 3	Document 4
Document 1	1.0	0.0	0.394	0.0
Document 2	0.0	1.0	0.732	0.615
Document 3	0.394	0.732	1.0	0.45
Document 4	0.0	0.615	0.45	1.0

Definition 14. *Dot product.*
Dot product indicates the length of projection of one vector onto another in n-dimensional vector space. Returned value is a scalar number. Dot product of two vectors A and B can be defined as following:

$$A \cdot B = \sum_{n=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

Definition 15. *Euclidean norm.*
Norm is a function that measures length of given vector in n-dimensional vector space.

$$||X|| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

3.4.2. Clusterization

Once documents are in a form of vectors it is possible to prepare data clusters. For this, one of the mostd frequenlty used unsupervised methods is the K-means algorithm. It has linear time complexity in relevance to the number of documents and it is possible to run it concurrently on multiple threads.

First step in K-means is to select K documents that will function as centers of groups. Most often these centers are selected randomly, which makes each run of K-means a bit different. Knowing the distances between each pair of documents, we divide them into K groups. Each group is a set of documents with the distances closest to center of each group. Once groups are created, new centers are selected by looking for a document that is closest to the calculated average of document distances in its cluster. We repeat these steps until centers are still - during iteration new centers are the same as previous ones. More detailed implementation of a single-process K-means was described in 2.3.3.

There are many variants of K-means parallel implementation, mostly depending on architecture of the solution. In our implementation, the following steps are proposed:

1. **Initialization**
   Each of processes receives own copy of all documents represented by a single matrix containing

calculated vectors. If processes are running on seperate nodes and document set is large, it should be considered to compress data to save time while sending it over network.

2. **Select K centers**
   Master process selects K centers and distributes them to processes. With an assumption that each process has an equal processing power, each process receives $\frac{K}{N}$ centers.

3. **Calculating distances between documents and centers**
   Each process calculates distances between each pair of documents and centers that were received in a previous steps. This partial result is sent back to master process.

4. **Assigning documents to groups**
   Master process using the distances received from other processes, can assign each document to the closest group created around one of K documents selected previously.

5. **Calculating new centers for the groups**
   Once the gorups are formed, next step is to find new centers. For each group, master process calculates mean value of distances between vectors that form it. It is possible that calculated mean value does not indicate any document directly, so the one with closest distance to it is selected.

6. **Sending updated centers**
   Master process sends updated group centers to other processes. Repeat step 4.

7. **Ending condition**
   If groups centers did not change in past iteration - clusterization has finished.

3.4.3. Classification

Classification is process of assigning a label to new document. It is basing on knowledge gathered in previous step and assumes that documents are grouped and labeled. In this project a two-step classification is proposed. In first step a kNN algorith is used for narrowing the potential class memberships. In second step a SVM algorith used for a precise decision to which class new documet should be assigned to. As both algorithms were described in details in previous chapter ([2.3.4]), this section focuses on an aspects of parallelism.

k-Nearest-Neighbors

k-Nearest-Neighbors is a simple distance-based classification algorithm which gives information about class membership of the most similar (in nearest neighborhood) documents. Knowing that the majority of neighbors belong to certain class, it assumes that this document must be this class as well. Because of its simplicity its parallel implementation is not complex and looks very similar to the single-threaded one.

1. **Extracting features from new document**
   First step is a repetition of document parsing that was made for all the documents that are already in the system. Because in this process only one document is taken into account, this part is not parallelised because all operations are atomic and run on a small set of words. In this step words in new document are stemmed and their statistic are counted. This allows to

present new document in a form of features that is comparable with other documents that are already in the system.

2. **Calculating distances between documents**
   Once the document is presented as a feature matrix, the distances between new and all existing documents can be calculated. This step is run in parallel with an assumption that every process has an equal procesing power. Each of N processes is given a $\frac{1}{N}$ of known documents and does the distance calculations.

3. **Aggregating results and finding nearest neighbors**
   All results are aggregated by a single process. Once done, k documents with highest distance value (distance is a function of similarity which value ranges from 0 to 1, the higher value the more similar documents are) are selected.

4. **Calculating class membership for new document**
   Optional. Because this implementation uses kNN to narrow data on which SVM will work this step is not required.

   For final *kNN* result documents selected in previous step are checked for their class membership. Class that contains the most documents is chosen as a label for new document.

Support Vector Machine

In this implementation Support Vector Machine is a second step in classification. To enhance its performance first dataset is narrowed by finding the most relevant documents using kNN algorithm. SVMs are inherently two-class classifiers, so there are to approaches to be considered when dealing with multi-class documents. First approach is called "one-versus-one" - it creates $\frac{C(C-1)}{2}$ classifiers to do binary classification in pairs. Second one is "one-versus-all", in which C classifiers do binary classification in this-class versus all-others manner. The first approach was chosen for this implementation due to possibility to distribute workloads between processes.

1. **Extracting features from new document**
   Optional. Executed only if document was not parsed before, for instance if kNN was not run.

   First step is a repetition of document parsing that was made for all the documents that are already in the system. Because in this process only one document is taken into account, this part is not parallelised because all operations are atomic and run on a small set of words. In this step words in new document are stemmed and their statistic are counted. This allows to present new document in a form of features that is comparable with other documents that are already in the system.

2. **Create $\frac{C(C-1)}{2}$ classifiers**
   Create $\frac{C(C-1)}{2}$ classifiers, where C is the number of classes in corpus indicated by kNN. This results in creating a single classifier for each possible pair of classes. Add classifiers to processing queue.

3. **Classification**
   Each of worker processes receives pair of classes from processing queue and runs classification. The result is sent back to master process which aggregates received results.

4. **Assigning label to new document**
   Master process counts aggregated results. New document is labeled with class that occurred most frequently.

3.5. Conclusion

This chapter includes a complete description of the methodology used to prepare analysis of the requirements for created system. Parallelization of documents classification is the main scientific part of proposed solution. Providing algorithms for documents features extraction, clasterization and classification in the application based on parallel processing may lead to previously undiscovered system that allows to accelerate document classification significantly.

Chapter 4

System design

4.1. Introduction

This chapter defines aim of the project and describes potential users of the application. Secondly, it divides requirements that were specified for this project into separate sections - functional, non-functional, hardware, software and quality - where they are described in detail. Next, the project of the system is presented by UML and classes diagrams. Finally, the most important libraries used for document classification framework are defined.

4.2. Aim of the project

Main aim of this project is to develop a platform for automated text document classification. Proposed solution will be capable of classificating text documents with a focus on parallel execution of algorithms. Classificator will implement a dual-layer model. In first layer a k-Nearest-Neighbors [2.3.4] algorithm will be used for a general decision narrowing the set of possible membership classes. More detailed and precise classification decision will be made by second layer with Support Vector Machine [2.3.4] algorithm.

The application will be created with the intent of providing a complete set of tools for text document classification and documenting usability of various machine learning algorithms. This includes:

- **feature extraction** - creating feature vectors for text documents, methods for reducing vectors dimensionality,
- **clusterization** - finding groups of similar documents that would define clusters,
- **classification** - defining position of new documents in cluster space and finding its best matching group membership,
- **algorithms performance tests and results** - all algorithms will be tested with various parameters and the results will be described.

The final solution should also indicate how complex the computation part is and how crucial it is to design a system that supports parallel tasks execution, distributes workloads in efficient manner between available worker processes and makes a proper use of all available resources.

The project may become a useful tool not only as a base framework for data scientists, but also for writers whom it should aid assigning a proper label to the document.

4.3. Potential users

The application will be dedicated mainly to data scientists, for whom it could act as a base natural language processing and classification framework. Application design will make it possible to add new modules and test the flow in high performance parallel environment.

On the other hand, users like article writers and administrators of knowledge bases will also be able to use this tool. For writers it might serve as an advisor which will help to mark document with proper labels, removing the burden of having to know the context of other documents already in the database. For knowledge base administrators this framework might be convenient way keeping the database and documents grouped in consistency.

4.4. Requirements

In this section the requirements for the project are defined. It starts with placing application in today's business context, where examples of possible adaptations are described. Moreover, the prospect of future work, which outlines the possible ways of evolution is also presented. Next, the non-functional and functional requirements are described along with the example of use cases. Then, it describes software and hardware requirements, outlining minimal configuration that must be met and recommending hardware parameters that would help increase overall applications performance. Finally, architecture and design are described in detail and illustrated with class, sequence and flow diagrams.

4.4.1. General requirements

1. **Business context**

   Application will be implemented with portability in mind. Everone will be able to use it by downloading it and running localy, either on commodity hardware or server-grade workstations with multiple processors. This project will make management of knowledge databases more attractive in the context of:

   (a) keeping database clean and well organized by sorting documents into seperate categories

   (b) providing means of assigning label to new document without forcing user to have knowledge of all documents context

   Furthermore, application will be available online, what provides faster access to it for a wide range of users. Moreover, because of releasing it under GPLv2 license, one will be able to adapt framework to ever changing goals by being able to change existing or add new modules. This release approach will make the application more available to all potential users, thus helping to increase its user-base.

2. **Prospectiveness**

   Application will be designed with object oriented principles, enabling it to be a part of a larger system by exposing different interfaces. In the future, application could be adapted

to users' needs and be a part of a bigger system. As the implementation itself is modular, it is possible to add, remove or rearrenge logic blocks inside the application. This indicates the prospectiveness of proposed system, its agile approach and possibility of including it as a part of another platform.

4.4.2. System requirements

Non-functional requirements

Another goal for this project was to create a platform that allows to run methods mentioned above (4.4.2) in efficent way by providing means for utilizing modern multi-core or multi-processor systems. There are multiple factors that must be taken into account. First of all, multi-core or multi-processor systems - platforms with such hardware are capable of executing multiple tasks in parallel. In order to provide processors with tasks, a task scheduling algorithms will be introduced for effective usage of available resources. Second aspect is memory management. While using multiple threads and processes, extra computing boost is given, but the overall performance may be greatly affected by data passing methods. For every algorithm an appropriate approach will be chosen.

Restrictive requirements

Most of the application steps will be automated, but still may require additional configuration or attention from the user. Application will be written without graphical user interface and, as such, will require users to poses basic skills of using command line interface.

Functional requirements

One of goals for this project was to create a platform that allows to run clusterization and classification algorithms in efficent way.

Diagrams presented in the figures 4.1 and 4.2 show general functional requirements available in the application.

These diagrams were used to present possible interactions between users and functions offered by the system. Each oval shape on the diagram is a single use case within which different flows called scenarios may occur. Each use case shows system functionalities together with initial requirements and steps that lead to the result observable by the actor. There are also two dependencies **include** and **extend**. A dependency **extend** describes additional feature, which in some cases (when the specific extension point is reached and the extension requirement is filled) may appear in the base sequence. An **include** dependency describes the inclusion of another use case into base sequence. This use case is responsible for different behavior.

To present more detailed interactions, use cases from following categories were extended and shown on separate diagrams.

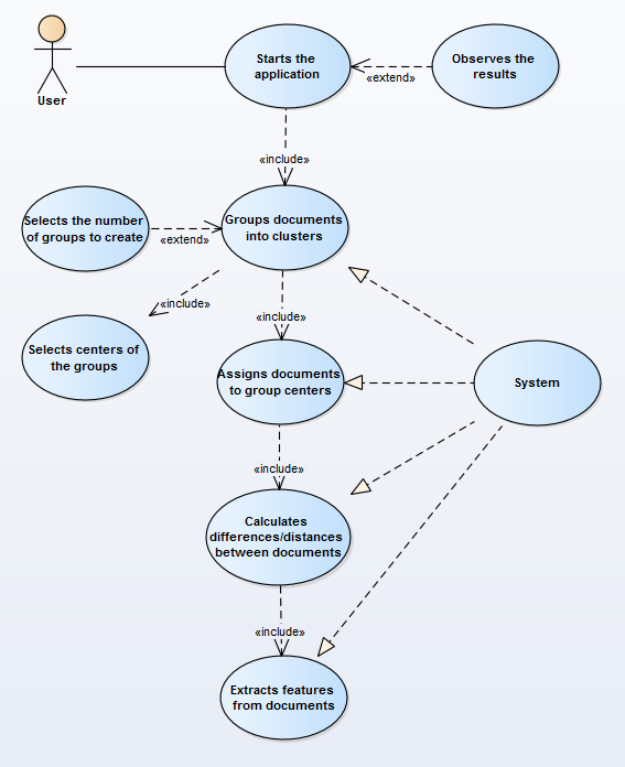


Figure 4.1. Use case 1. Clusterizing documents

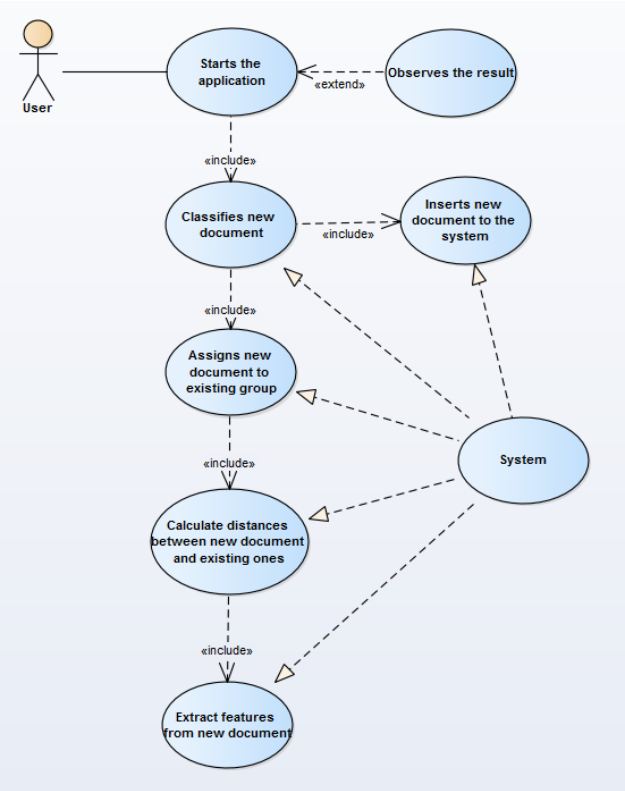


Figure 4.2. Use case 2. Classifying new documents

Descriptions of all of the use cases including their initial/alternative conditions and results were collected in the table 4.1.

Use case	**Initial conditions**	**Alternative conditions**	**Results**	**Description**
starts the application	code of the application was downloaded/cloned	any	application is running	a user starts application via CLI - command line interpreter/interfce
groups documents into clusters	the code of the application is accessible to a user, the application was started, set of documents that will form clusters is available	classifies new document	documents from a given set are grouped into created groups based on their similarity	a user starts the application and use option that gropus documents into specific number of clusters
selects the number of groups to create	the application was started, the user wants to group documents	any	parameter that specifies number of groups is passed to the application by setting it with the application option	the user starts the application and use option that gropus documents followed by option that specifies number of clusters

selects centers of the groups	the user chose option **group documents into clusters**, set of documents that will form clusters is available	any	centers of the groups are defined	the user starts the application, chooses an option to cluster documents, centers of groups are at first randomly selected and then in each iteration calulated as the average document in each cluster, this step is terminated when any center is changed
assigns documents to group centers	the user chose option **group documents into clusters**, set of documents that will form clusters is available, centers of the groups were chosen	any	each document is assigned to one cluster	the user starts the application, chooses an option to cluster documents, centers of clusters are chosen and each document is assigned to the cluster to which its calculated distance is shortest

calculates differences/distances between documents	the user chose option **group documents into clusters**, centers of the groups were chosen	any	distance between each document and each center is calculated	the user starts the application, chooses an option to cluster documents, centers of clusters are chosen and distance between each center and each document is calculated as cosine similarity based on extracted features
extracts features from documents	the user chose option **group documents into clusters**, set of documents that will form clusters is available	any	features for each document are calculated	the user starts the application, chooses an option to cluster documents, specific features for each document in each iteration are extracted
classifies new document	the user chose option **classify new document**, a new document is available, clusters of other documents are available	groups documents into clusters	a new document is assigned to one of a given clusters	the user starts the application, chooses an option to classify a new document, source for available clusters and path to a new document via application options

insert a new document to the system	the user chose option **classify new document**, a new document is available	any	a new document is added to the system	the user starts the application, chooses an option to classify a new document and selects the source of new document. New document is loaded to the system. New document is parsed basing on statistics already calculated for whole corpus. As a result document's representation, the feature vector, is created.
assigns a new document to existing group	he user chose option **classify new document**, a new document is available, clusters of other documents are available	any	a new document is classified	the user starts the application and chooses an option to classify a new document. User can select number of documents that will be voting in kNN algorithm and can choose wether to run SVM method for detailed decision.

Table 4.1. Use cases description

4.4.3. Hardware and software requirements

In order to use the application the following software and hardware requirements will have to be filled.

Hardware requirements

This application will be tested on various hardware configuration to prove that it can work reliable on even older machines. Although the following configuration is not a server-grade hardware, it may allow to execute flow of the application without running out of available resources in most of the presented test cases. It will be considered as a recommended minimal configuration.

- Processor - at least a quad-core processor is recommended. Usage of technologies like Intel's HyperThreading is not advised and the number of spawned child processes should not exceed the number of physical cores. Application will rely heavily on reads and writes to shared memory, thus limiting use of virtual cores ([29]). Processor used during tests will be Intel i7 with 4 physical cores and 8 logical processors.

- System memory - probably will have the greatest impact on performance of the application. The more memory, the less caching mechanisms will be used. Caching uses hard drives as a backend for storing processed documents, which greatly affects overall performance and execution time. Machine with 16 GB DDR3 memory will be used during tests.

- Hard drives - speed of reads and writes to hard drives is a crucial factor during startup of application and in case of using cache. SSD drive should be considered in place of standard hard drive due to higher speeds of reads and writes and O(1) latency of access [30]. A SSD drive connected to PCI-e slot will be used during tests.

Software requirements

There are no specified requirements about the operating system installed on the device. Application should run on every modern operating system, either Linux, OS X or Windows. Tests will be run on both Linux and OS X machines, however the Windows environment will be never fully tested and thus is not recommended for running the application.

Taking software requirements into consideration, a Python interpreter will have to be installed on a machine along with a set of libraries. Python is an interpreted language that provides good performance, scalability features and possibility of adapting software on-the-fly without a need to recompile the code. All required libraries are listed in both section 4.5.5 and in source code in *source/requirements3.txt* file.

4.5. Project of a system

This section presents the structure of the implemented system. It describes the architecture and flow of information as presented on included UML diagrams.

4.5.1. Activity diagram of the performing test flow

The activity diagram shows the logical flow of application in steps described in high-level abstraction. Performed sequence of activities are presented as a rectangle with rounded corners and decisions are depicted as rhombuses. The diagram in the figure **??** shows the sequence of actions that are required to complete whole workflow of implemented application. Flow initializes by starting the application, which first reads parameters stored in a configuration file. Next application attempts to read data source and checks if there are ungrouped documents. In positive case application

starts to process them and runs clusterization algorithm, otherwise it goes to next decision. If new document was indicated, application performs the classification. Flow ends if all preceding decisions were Ńoór selected modules finilized their execution.

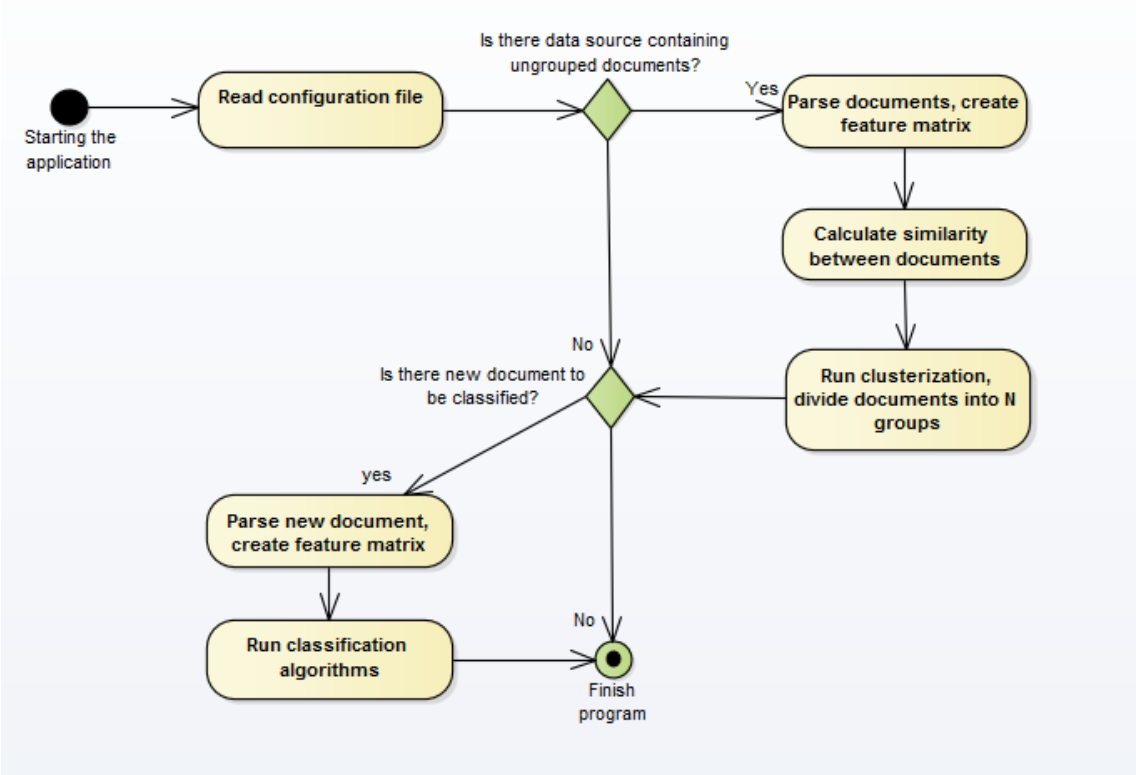


Figure 4.3. Activity diagram of the performing test flow

4.5.2. Sequence diagram

A sequence diagram (figures 4.4, 4.5, 4.6) was used to present steps needed to perform a complete flow of application, including both clusterization and classification.

For the enlarged diagram, please refer to Appendix B

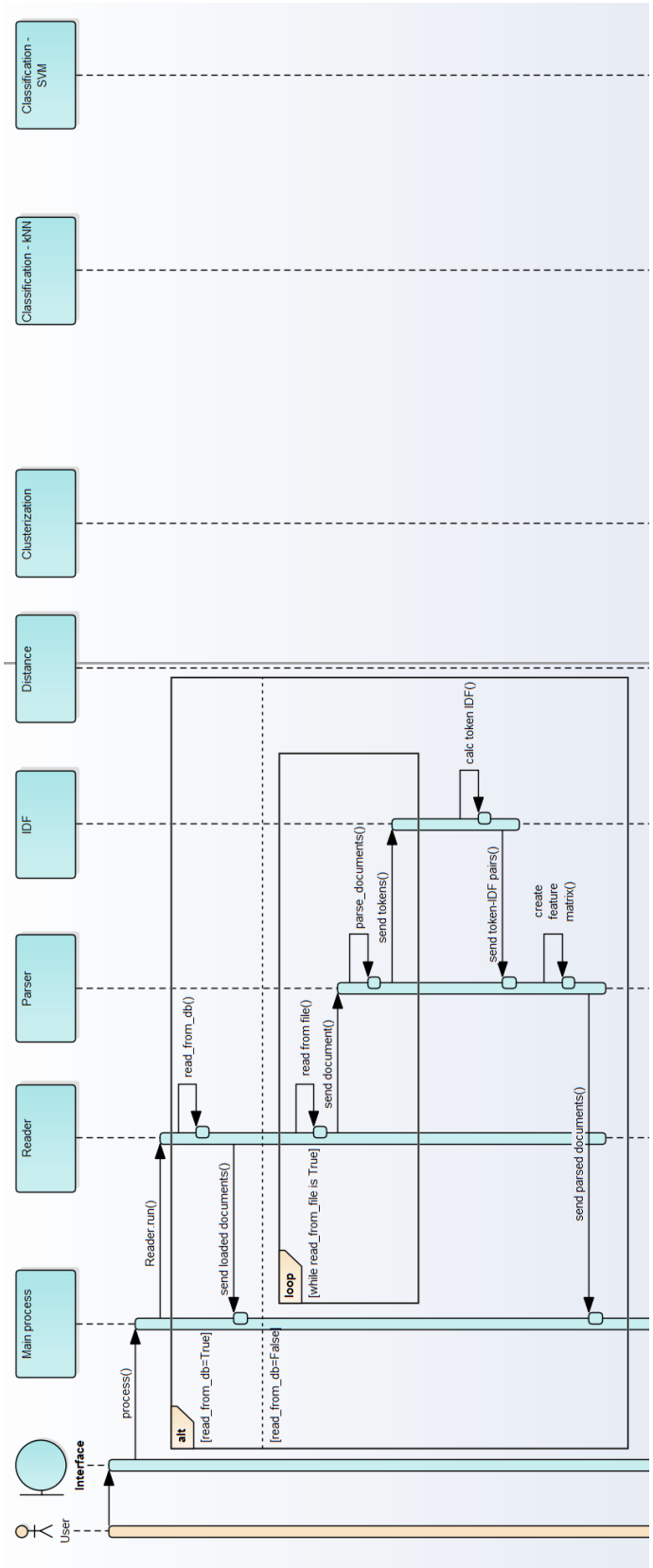


Figure 4.4. Sequence diagram, part 1

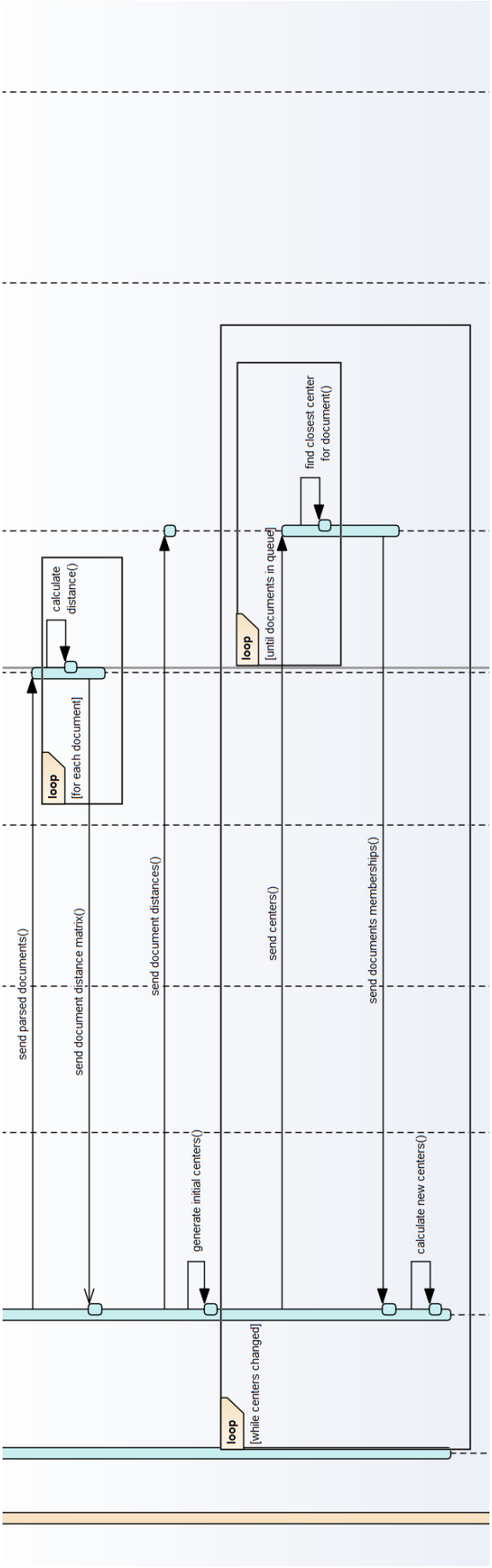


Figure 4.5. Sequence diagram, part 2

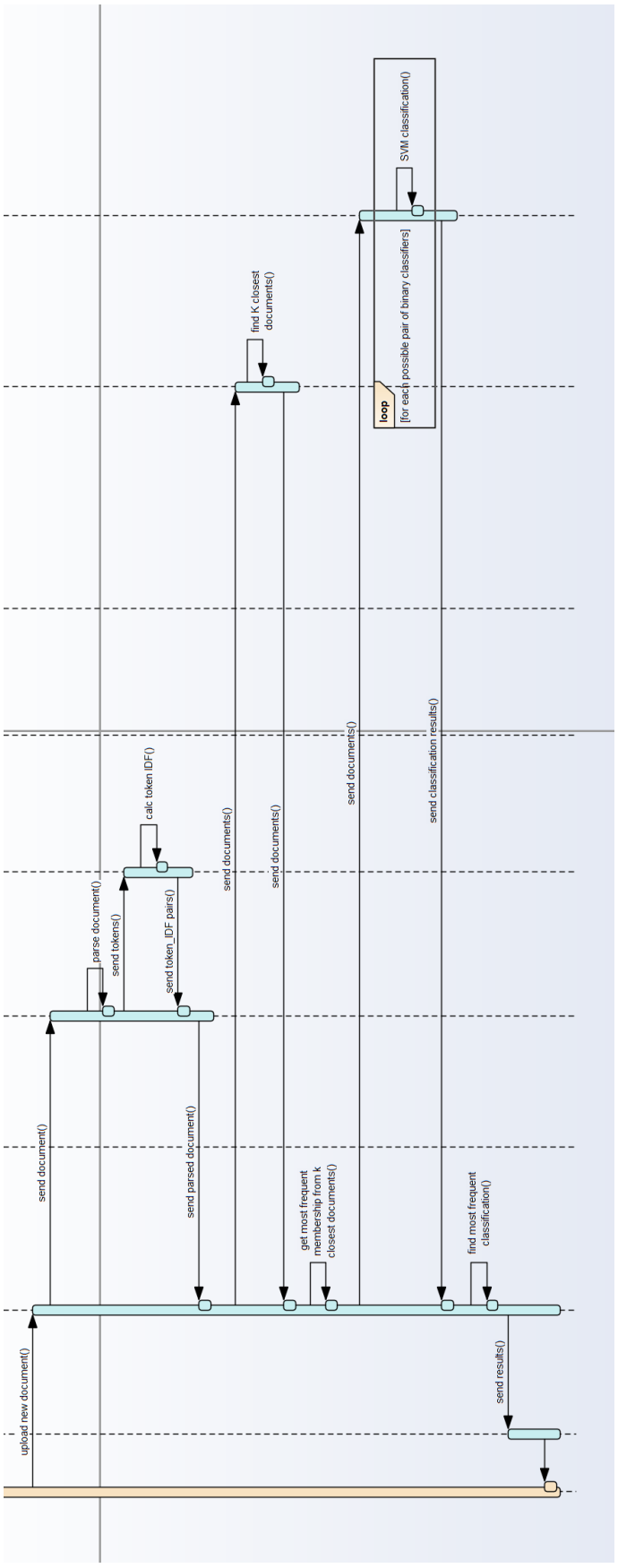


Figure 4.6. Sequence diagram, part 3

4.5.3. Class diagram

Class diagrams 4.7 and 4.8 expose implementation structure. Blocks were used to present classes and their internal methods and fields. Classes connected with association line communicate with each other either by passing values as parameters in function calls or by sending message objects using process-safe pipe/queue implementation.

For the enlarged diagram, please refer to Appendix A

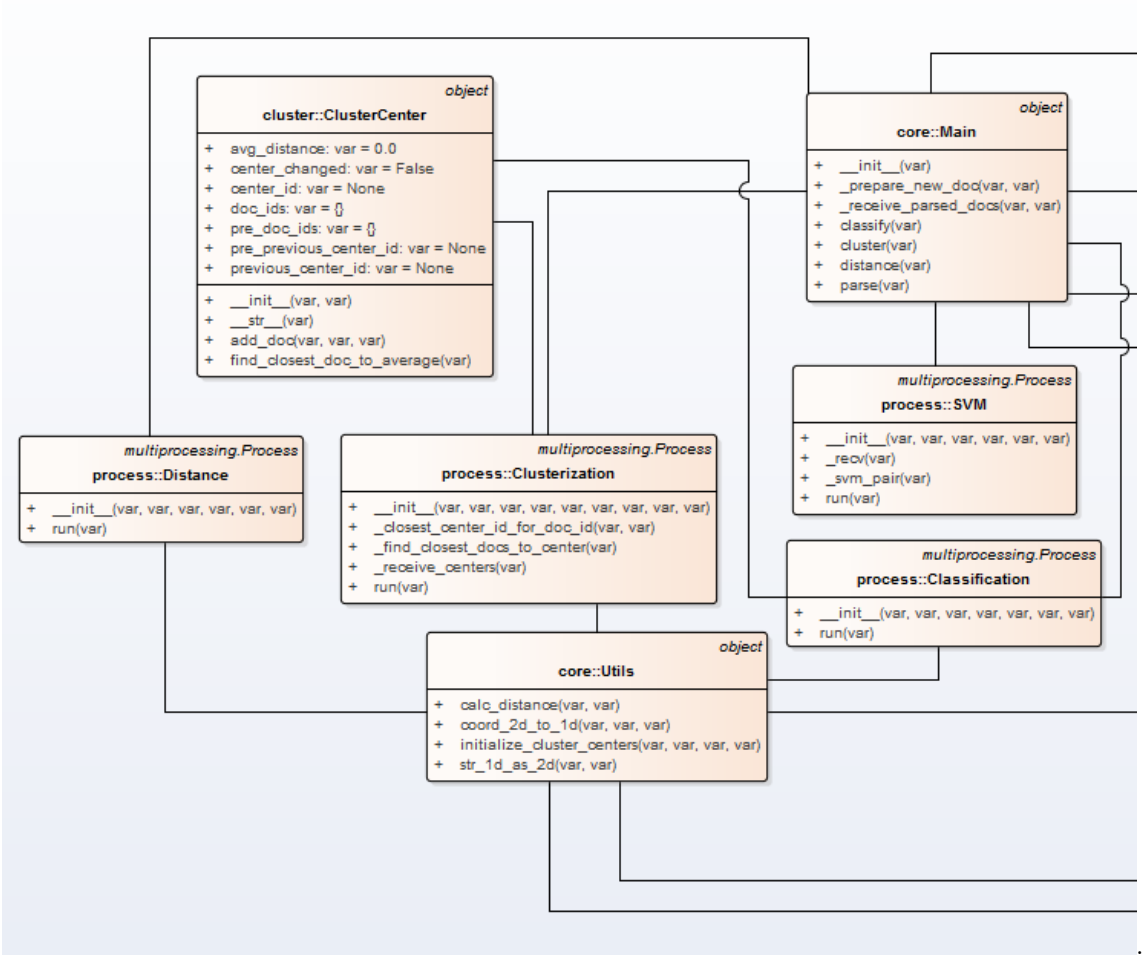


Figure 4.7. Class diagram

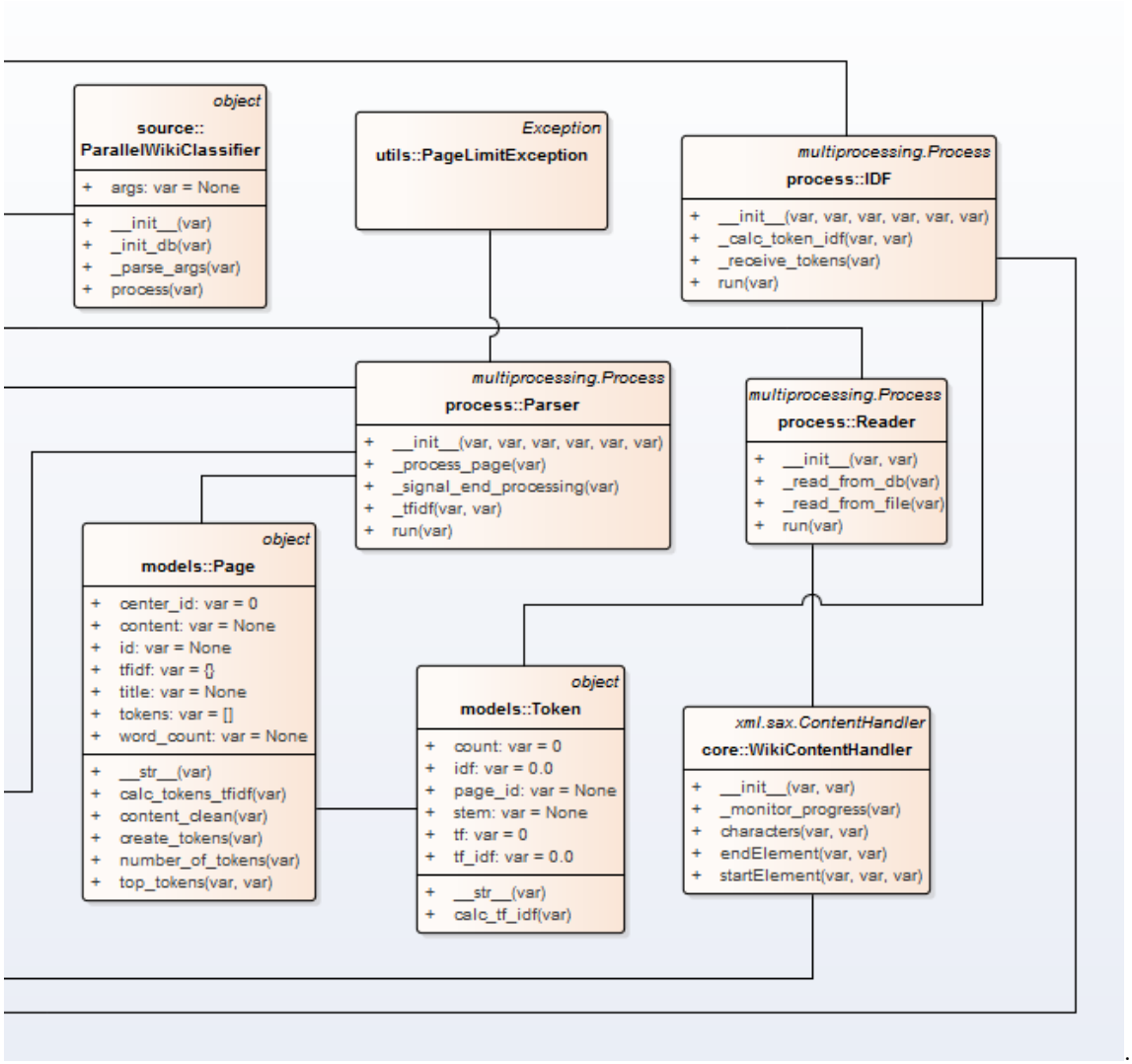


Figure 4.8. Class diagram

A short description of responsibilities of each class presented in diagrams 4.7 and 4.8 is shown below:

- **ParallelWikiClassifier**
  Object of this class is an entry point of the application. It is responsible for parsing command line arguments passed to the application and starting main process.

  There is only one instance of this class running in single process.

- **core.Main**
  Class of the main process of the application. Instance of this object holds the state of the application, reads the configuration file, partitionates and passes data, manages worker processes and delegates tasks.

  There is only one instance of this class running in single process.

- **core.Process.Reader**
  Process of this class responsible for handling data source related operations. Depending on configuration it either calls Parser instance for parsing XML file or connects to the database,

from which it reads parsed documents. Cooperates with Parser and IDF processes, with whom it exchanges data using queues and pipes to deliver parsed documents in a form of feature matrix.

There is only one instance of this class running in single process.

- **core.Process.Parser**
  Process of this class responsible for parsing documents. It tokenize documents and sends trimmed data to IDF process from which it awaits response with Inverse Document Frequencies of tokens to calculate TF-IDF. If all TF-IDF values for given document were received, Parser sends its feature matrix back to Parser process to aggregate.

  There are multiple instances of this class which run in parallel as seperate processes.

- **core.Process.IDF**
  IDF values of tokens are calculated in process of this class. It works in two phases, between which transition is dependent on the state of Parser process. In first phase it aggregates all tokens received from Parser processes. Once Parser processes monit that all documents were prepared, IDF process calulates IDF values for all aggregated tokens sending each token back to Parser process after competition.

  There is only one instance of this class running in single process.

- **core.Process.Distance**
  Instance of this class is responsible to calculating a distance between parsed documents. Processes of this class work on shared memory object that holds matrix of documents distances. Every process works on a single row of matrix, which represents distances between one document to every other. Upon completition it moves to next row that is not locked by other working processes.

  There are multiple instances of this class which run in parallel as seperate processes.

- **core.Process.Clusterization**
  Class of clusterization processes, which are responsible for finding documents that are closest to given center of group. Each process is given a subset of all documents, through which it iterates assigning documents to closest center. Clusterization processes communicate with master process which is responsible for aggregating results and sending new centers back to processes, if required.

  There are multiple instances of this class which run in parallel as seperate processes.

- **core.Process.Classification**
  Objects of Classification class are conducting both kNN and SVM classification algorithms.

  There are multiple instances of this class which run in parallel as seperate processes.

- **core.WikiContentHandler**
  This class enables reading dumps of Wikipedia database. Database dump is saved in a form of XML file which contains all articles in structured form. This class implements stream reader to deal with the size of XML files, which often span across multiple gigabytes of data. Object of this class is created if XML file was set as a source of documents in configuration file.

- **models.Page**
  For each document in the system there is a coresponding Page class object. It holds compressed information about documnt in a form of feature matrix.

- **models.Token**
  For each token in the system there is a coresponding Token class object. Each Token object holds statistics for a single stemmed word in the system.

- **models.cluster.ClusterCenter**
  Class that holds information about document which was chosen as cluster center in classification process. It holds a list of documents that were assigned to this group.

- **utils.PageLimitException**
  Special exception which is used if document limit was set in configuration file.

4.5.4. Architecture

The application will be based on master/slave model. It is a distributed application structure that allows for better resource management by partitioning tasks and data. In this approach, master process is responsible for partitioning data and scheduling tasks between worker processes called slaves. The main idea of this model is presented in the figure 4.9

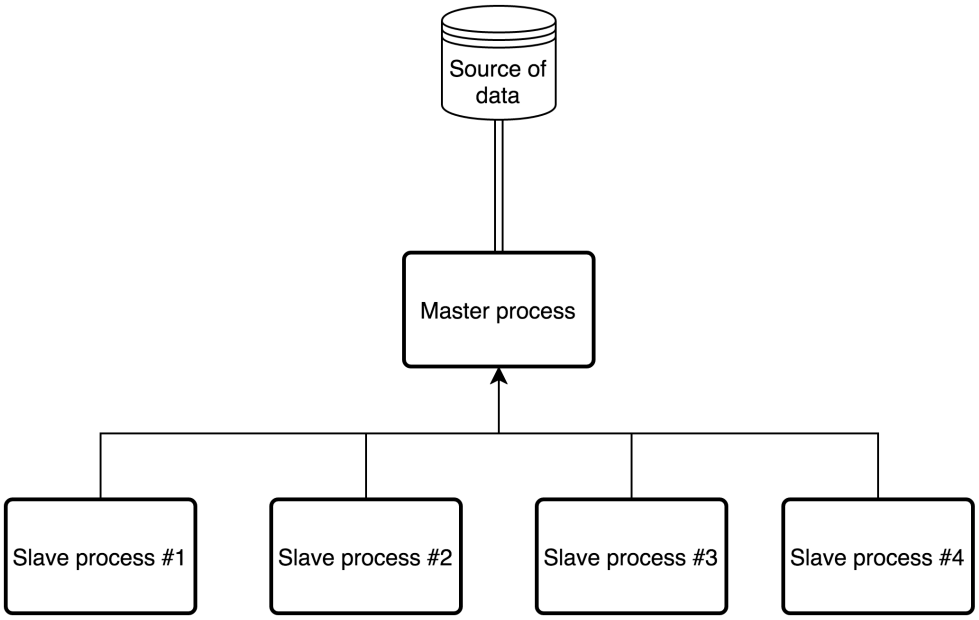


Figure 4.9. Main idea of the master-slave model

Master process is responsible for conducting whole workflow of application. It tracks progress of application execution holding its state, decides when specific methods should be executed, which data will be provided for a given work and which of slave processes will conduct the job. Master process holds state of the application and uses multiprocessing interfaces to communicate with workers.

Slaves do not hold the state of application and their life span is limited to the time of execution of received method. After computing its part and sending results back to master, slave process is killed to free up resources it used.

In case of this implementation it's the best to assume that master and slave processes are running on the same physical machine. Though it is possible to configure solution to work over the network, expect delays in communication to cause significant performance loss.

4.5.5. Libraries

This project is designed to be portable across different platforms and to have minimal dependencies. It will use Python platform and libraries that are freely available via pip, the Python package installer.

Platform dependencies that will have to be installed by user:

- **Python 3.5.1** - the Python language interpreter.
- **pip 8.1.2** - Python package installer.

Libraries that are available in default Python 3.5 installation and thus will not require additional steps to install:

- **multiprocessing** - will be used for parallelisation mechanisms. Contains basic shared memory data structures and allows for usage of processes. Processes will be chosen over threads because they are not blocked by Python's Global Interpreter Lock (GIL) and allow to execute tasks in parallel not only locally, but also over network.
- **xml.sax** - will be used to access XML files using streams. Allows to open files that are too big to fit in memory.

External Python libraries that can be installed using pip:

- **nltk 3.2.1** - Natural Language ToolKit - is used for word stemming thanks to build-in WordNet
- **SQLAlchemy 1.0.14** - library that will be used for database queries. It is optional, as this project uses database as a cache only when requested by user. By default a sqlite database will be used as a backend because it does not have any additional requirements and is very portable. While its performance may suffer as it does not offer any database management system, it is not the problem in project's implementation.

4.5.6. Application release

Presented solution will be developed as an open-source application under GPLv2 license and will be freely available to everyone for using, testing or modifying purposes. Its both source code and documentation will be available online at git repository[1] hosted on GitHub.

[1] https://github.com/macsz/parallel-wiki-classifier

4.6. Conclusion

This chapter presents the architecture used to implement algorithms described in the previous chapter. It contains description and analysis of requirements for proposed solution. The designed architecture takes advantage of multi-process approach to provide efficient platform for documents clusterization and classification.

Chapter 5

Implementation

5.1. Introduction

The presented solution uses a master-slave architecture in order to provide a scalable platform with means of parallel task execution. This chapter describes a code implementation for all of the algorithms and methods used in text document clusterization and classification process.

5.2. Implementation

5.2.1. Preparing documents

The first major step of the designed application is to prepare documents for further processing. This task includes loading documents from the source, converting them into feature matrix, reducing dimmensionality and finally measuring similarity between two documents.

Reading and parsing documents

Whole task is orchestreted by the *parse()* method of the main process. It creates all processes involved in reading and parsing documents:

- a Reader process,
- multiple Parser processes,
- a single IDF process.

It also initializes communication channels for child processes that are used to exchange information - queue, pipe and event objects are utilized.

Listing 5.1. Main.parse() - Main process method for conducting document parsing

```
1  def parse(self):
2      global parsed_docs
3      global largest_id
4      global tokens_idf
5
6      # initialize communication
7
8      queue_unparsed_docs = multiprocessing.Queue()
9      queue_parsed_docs = multiprocessing.Queue()
10     pipe_tokens_to_idf_parent, pipe_tokens_to_idf_child = multiprocessing.Pipe()
```

```
    pipes_tokens_to_processes_parent = []
    pipes_tokens_to_processes_child = []
    for i in range(PROCESSES):
        pipe_tokens_to_processes_parent, pipe_tokens_to_processes_child = \
            multiprocessing.Pipe()
        pipes_tokens_to_processes_parent.append(pipe_tokens_to_processes_parent)
        pipes_tokens_to_processes_child.append(pipe_tokens_to_processes_child)

    # additional pipe to transfer IDF values from IDF process to master
    pipe_idf_master_parent, pipe_idf_master_child = multiprocessing.Pipe()
    pipes_tokens_to_processes_parent.append(pipe_idf_master_parent)

    event = multiprocessing.Event()
    event.clear()

    # set up processes

    ps_reader = Reader(q_unparsed_docs=queue_unparsed_docs)
    ps_parsers = create_parsers(
        queue_unparsed_documents=queue_unparsed_docs,
        pipe_tokens_to_idf_child=pipe_tokens_to_idf_child,
        event=event,
        pipes_tokens_to_processes_child=pipes_tokens_to_processes_child,
        queue_parsed_docs=queue_parsed_docs,
        process_num=PROCESSES
    )
    ps_idf = IDF(
        pipe_tokens_to_idf_parent=pipe_tokens_to_idf_parent,
        docs_num=int(CONF['general']['item_limit']),
        event=event,
        pipes_tokens_to_processes_parent=pipes_tokens_to_processes_parent,
        process_num=PROCESSES
    )

    # read all the articles from XML and do TF-IDF
    ps_reader.start()

    LOG.info("Started␣processing␣documents␣using␣{0}␣processes".format(
        PROCESSES))
    for ps_parser in ps_parsers:
        ps_parser.start()
    ps_idf.start()

    # receive tokens IDF values from IDF process
    tokens_idf = pipe_idf_master_child.recv()

    ps_reader.join()
    ps_idf.join()

    # processes will not end until all the data is not received
    parsed_docs = self._receive_parsed_docs(queue_parsed_docs)

    for ps_parser in ps_parsers:
        ps_parser.join()
```

The first step of the presented workflow is connected with loading documents into a system. This implementation allows for it in two alternative ways:

- parse database dump represented in a form of a large XML file
- load them from the prepared database.

The number of solutions for loading documents can be easily expanded as their functions are self-contained and the only output are the documents. As long as any algorithm provides documents in a proper form as its output, it can be used in this step. To prevent workflow from blocking, this

task is conducted by the Reader class in a single, seperate process. Main method of the Reader class is presented on listing 5.2.

Listing 5.2. Reader.run() - Reader class process main method

```
def run(self):
    if str2bool(CONF['general']['load_from_db']):
        LOG.info('Loading from DB')
        self._read_from_db()
    else:
        self._read_from_file()
```

Loading documents from a database (listing 5.3) is the most basic and straight forward solution. This method consist of making a connection to a database and reading data found in **Doc** table into the Page class instance, that would serve as a container for a single document throughout the whole application lifecycle. The created Page objects are put into queue for further processing.

Listing 5.3. Reader._read_from_db() - reading docs from database

```
def _read_from_db(self):
        Db.init()
        session = Db.create_session()
        pages = session.query(Models.Doc).all()

        for page in pages:
                p = Page()
                p.id = page.id
                p.title = page.title
                p.content = page.text
        self._q_unparsed_docs.put(p)
        # A pill for other threads
        self._q_unparsed_docs.put(None)
```

The second approach (listing 5.4) includes loading documents from a database dump in a form of an XML file. This approach is more complicated, as it includes reading data from a file that most often can not be fit into computers memory - size of XML containing articles from the Wikipedia often span across multiple gigabytes of data. Knowing that this file can not be read in a standard way, another approach was proposed. Python's XML.SAX module allows to treat an XML file as a stream of data and parse it using own handlers. For this purpose the WikiContentHandler class was designed and implemented. After reading each article it converts them into the Page class objects and puts them into the queue for further processing.

Listing 5.4. Reader._read_from_file() - reading documents from XML file

```
def _read_from_file(self):
    wiki_handler = WikiContentHandler(self._q_unparsed_docs)
    sax_parser = xml.sax.make_parser()
    sax_parser.setContentHandler(wiki_handler)

    try:
        data_source = open('../data/wiki_dump.xml')
        sax_parser.parse(data_source)
        LOG.info('Parsed {0} items'.format(wiki_handler.items_saved))
    except PageLimitException as page_limit_exception:
        LOG.info(page_limit_exception)
    except KeyboardInterrupt:
        exit()
    finally:
        # A pill for other threads
        self._q_unparsed_docs.put(None)
```

Both cases are finished with putting None value to the queue. It is a signal for other processes listening on the queue that all documents were read from source and put to the queue.

In the next step the loaded documents are parsed. Since the Reader process is sending documents for processing one by one from the very begining of reading data source, a self-contained instances of the Parser class are started in seperate processes along with it. For better efficency multiple processes of the Parser class are created.

Parser process logic is executed inside *run()* method presented in listing 5.5. It runs a loop until *_process_page()* method (listing 5.6) returns None value, which is send by the Reader class object once all documents were read. If parsed document is received, it will be added to the list of documents parsed by this process.

In other case the *_signal_end_processing()* method is called to notify all other processes that all documents in this process were parsed. Once list contains all documents that were assigned to this process, a _tfidf() (listing 5.7) method is called.

Listing 5.5. Parser.run() - Parser class process main method

```
1  def run(self):
2      self.parsed_pages_num = 0
3      parsed_pages = []
4      while True:
5          page = self._process_page()
6          if page:
7              parsed_pages.append(page)
8          else:
9              self._signal_end_processing()
10             break
11
12     self._tfidf(parsed_pages)
```

The Parser._process_page() presented in listing 5.6 is called for every single object send from the Reader process using queue. It is responsible for receiving and processing received documents. If document is received, its text is tokenized and lemmatized, and for each token a TF value is calculated in *Page.create_tokens()* method. All created tokens are send to the IDF process (IDF process is responsible for counting IDF value for each token, which then will be used to calculate TF-IDF values), where they are aggregated and counted. The Reader process sends None object after all documents were loaded from source, in which case the None value is returned to notify calling method that processing work was finished.

Listing 5.6. Parser._process_page() - Processing of unparsed document

```
1  def _process_page(self):
2      page = self._queue_unparsed_docs.get()
3      if page is None:
4          return None
5      page.create_tokens()
6      for token in page.tokens:
7          self._pipe_tokens_to_idf_child.send(token.stem)
8      page.content_clean()
9      self.parsed_pages_num += 1
10     return page
```

Once list in *Parser.run()* (listing 5.5) contains all documents that were assigned to this process, a _tfidf() (listing 5.7) method is called. First, a lock in a form of multiprocessing event is checked. If event is set to low state its *wait()* method causes calling process to stop and wait until event is set to high state. Once lock is raised, process starts receiving pairs of token and IDF values, which

allows to calculate a TF-IDF value for every token in the document and create a feature matrix. Parsed document is sent to the main process as soon as all values are calculated for it.

Listing 5.7. Parser._tfidf() - calculating TD-IDF values for tokens in documents

```
def _tfidf(self, parsed_pages):
    print('Process␣{0}␣waiting␣on␣IDF␣to␣finish...'.format(self.pid))
    self._event.wait()
    recv_tokens = self._pipe_tokens_to_processes_child.recv()
    print('Process␣{0}␣received␣{1}␣tokens␣from␣IDF'.format(
        self.pid, len(recv_tokens)))
    for page in parsed_pages:
        for token in page.tokens:
            try:
                token.idf = recv_tokens[token.stem]
                page.tfidf[token.stem] = token.calc_tf_idf()
            except KeyError as ke:
                print('error', token)
        self._queue_parsed_docs.put(page)
    # sending process-end pill
    self._queue_parsed_docs.put(None)
```

Calculating distances

Calculating distances, or the similarity between documents, is a crucial pre-requirement for clusterization and classification. It is conducted after all documents have been parsed and the feature matrixes are available. As a result, the matrix of distances is created, where row and column numbers corresponds to document IDs. For better efficiency multiple processes can be assigned to this task.

Task orchestration is conducted in *Main.distance()* method presented in listing 5.8. It spawns desired number of Distance class processes that will calculate distances for assigned portion of work. All processes have access to shared memory object of multiprocessing.Array class. It is an array implementation that allows to share data with read and write permissions between processes. As long as every process performs write operations on seperate cells of array, there is no performance degredation.

Listing 5.8. Main.distance() - Main process method for conducting distance calculations

```
def distance(self):
    global distances
    distances = multiprocessing.Array('d', (largest_id+1)*(largest_id+1))

    dist_ps = []
    for i in range(PROCESSES):
        dist_p = Distance(
            iteration_offset=i,
            iteration_size=PROCESSES,
            distances=distances,
            largest_id=largest_id,
            parsed_docs=parsed_docs
        )
        dist_p.start()
        dist_ps.append(dist_p)

    for dist_p in dist_ps:
        dist_p.join()
```

Internal operations of single process during document similarity calculations are presented in listing 5.9. Every process starts with a row which ID is equal to process ID. Upon completition of a single row, counter is increased by the number of processes working on this task. Because

distance is a commutative property it is calculated only once for a pair of documents X and Y and is inserted at once under X:Y and Y:X location in the matrix.

Listing 5.9. Distance.run() - Distance class process main method

```
def run(self):
    row = self.iteration_offset
    while row < (self.largest_id + 1):
        try:
            doc1 = self.parsed_docs[row]
            self.distances[Utils.coord_2d_to_1d(row, row, (self.largest_id + 1))] = 1.0
            for col in range(row):
                distance = 0.0
                try:
                    doc2 = self.parsed_docs[col]
                    distance = Utils.calc_distance(doc1, doc2)
                except:
                    distance = -2
                self.distances[Utils.coord_2d_to_1d(col, row, (self.largest_id + 1))] =
                    distance
                self.distances[Utils.coord_2d_to_1d(row, col, (self.largest_id + 1))] =
                    distance
        except:
            # there is no document with such ID, fill it with -1
            # distances
            for col in range(row+1):
                self.distances[Utils.coord_2d_to_1d(col, row, (self.largest_id + 1))] = -1
                self.distances[Utils.coord_2d_to_1d(row, col, (self.largest_id + 1))] = -1
        row += self.iteration_size
```

5.2.2. Clusterization

The implementation of k-means clustering algorithm presented in listing 5.10 is a continuous cooperation between master and slave processes. Master process is responsible for initializing communication channels, for scheduling workloads to slave processes and also for aggregating work done by slaves, assigning documents to groups, rescheduling all work and sending new group centers to slave processes.

At first, initial centers must be generated. It is done in *Utils.initialize_cluster_centers()* method, which returns a set of randomly selected documents that will act as initial cluster centers. The number of centers used during clusterization process can be predefined by changing *clusterization/centers* value in the configuration file. Centers are instances of *ClusterCenter* class, that holds different pieces of information like groups central document ID and the list of all documents that were assigned to this center.

All centers are sent to slave processes in order to find the closest center for given documents. Once slave processes send back results, master process aggregates all results and attempts to define more appropriate center in each cluster by looking for a document with distance value closest to the average of distances in whole cluster.

Whole process is repeated until one of conditions is met: either there was no change at the end of iteration or the iterations number is over the limit. Default value of iterations limit is *100*, but it can be changed in the configuration file.

Listing 5.10. Main.cluster() - Main process method for conducting clustering

```
def cluster(self):
    global distances
    global parsed_docs

```

```
5       center_num = int(CONF['clusterization']['centers'])
6       centers = Utils.initialize_cluster_centers(
7           center_num=center_num,
8           start=0,
9           end=largest_id,
10          parsed_docs=parsed_docs
11      )
12      new_centers = {}
13
14      cluster_ps = []
15      pipe_receive_results, pipe_send_results = multiprocessing.Pipe()
16
17      for pid in range(PROCESSES):
18          pipe_send_centers, pipe_receive_centers = multiprocessing.Pipe()
19          cluster_p = Clusterization(
20              offset=pid,
21              shift=PROCESSES,
22              pipe_send_centers=pipe_send_centers,
23              pipe_receive_centers=pipe_receive_centers,
24              parsed_docs=parsed_docs,
25              distances=distances,
26              largest_id=largest_id,
27              pipe_send_results=pipe_send_results,
28          )
29          cluster_p.start()
30          cluster_ps.append(cluster_p)
31
32      iteration = 0
33      iteration_limit = int(CONF['clusterization']['iterations_limit'])
34      changed = False
35      docs_num = 0
36      while iteration < iteration_limit:
37          docs_num = 0
38          for cluster_p in cluster_ps:
39              cluster_p.pipe_send_centers.send(list(centers.keys()))
40          new_centers = {}
41          not_finished = PROCESSES
42          while not_finished:
43              recv = pipe_receive_results.recv()
44              if not recv:
45                  not_finished -= 1
46              else:
47                  cid = recv['cid']
48                  did = recv['did']
49                  dist = recv['dist']
50                  centers[cid].add_doc(doc_id=did, distance=dist)
51                  parsed_docs[did].center_id = cid
52          for cid in centers:
53              docs_num += len(centers[cid].doc_ids)
54          for cid in centers:
55              new_cid = centers[cid].find_closest_doc_to_average()
56              if not centers[cid].center_changed:
57                  new_cid = cid
58              new_center = ClusterCenter()
59              new_center.doc_ids = {}
60              new_center.pre_doc_ids = {}
61              new_center.center_id = new_cid
62              new_centers[new_cid] = new_center
63              if cid != new_cid:
64                  changed = True
65
66          if not changed:
67              break
68          centers = new_centers
69          iteration += 1
70
71      for cluster_p in cluster_ps:
```

```
72          cluster_p.pipe_send_centers.send(None)
73          cluster_p.join()
74      print('Docs sum: ', docs_num)
75      print('parsed docs: ', len(parsed_docs))
76      print('centers:', len(centers))
```

Clusterization class process conducts search for closest documents. It awaits for master process to send a set of selected cluster centers. Each k-means iteration is calculated within a single iteration in each process in *_find_closest_docs_to_center()* method presented in listing 5.12. Each process awaits for master process to send new centers (*_receive_centers()* method) and then it starts computation presented in 5.12.

Listing 5.11. Clusterization.run() - Clusterization class process main method

```
1  def run(self):
2      while True:
3          self._receive_centers()
4          if not self.centers:
5              break
6          self._find_closest_docs_to_center()
```

_find_closest_docs_to_center() presented in listing **??** iterates through all documents that were sent to process and responds with a dictionary that contains information about document ID, its closest center and the distance between these two. Closest center and distance is calculated in *_closest_center_id_for_doc_id()* method in listing 5.13.

Listing 5.12. Clusterization._find_closest_docs_to_center() - Finding closest center for a document during clusterization, part 1

```
1  def _find_closest_docs_to_center(self):
2      did = self.offset
3      while did < self.largest_id:
4          try:
5              ret = self._closest_center_id_for_doc_id(did)
6              if ret:
7                  closest_cid, distance = ret
8                  self.pipe_send_results.send({
9                      'cid': closest_cid,
10                     'did': did,
11                     'dist': distance,
12                 })
13             did += self.shift
14         except Exception as ex:
15             print(ex)
16     self.pipe_send_results.send(None)
```

Last method used in document clusterization is *_closest_center_id_for_doc_id()* method presented in listing 5.13. It takes a single document and compares distances between it and all cluster centers. Because distance is a function of similarity in this implementation, center with highest value is selected.

Listing 5.13. Clusterization._closest_center_id_for_doc_id() - Finding closest center for a document during clusterization, part 2

```
1  def _closest_center_id_for_doc_id(self, did):
2      try:
3          test = self.parsed_docs[did]
4      except:
5          return None
6      closest_cid = None
7      closest_cid_distance = -100
8      for cid in self.centers:
```

```
9           cid_distance = self.distances[Utils.coord_2d_to_1d(cid, did,
10                                                             self.largest_id)]
11          if closest_cid_distance < cid_distance:
12              closest_cid = cid
13              closest_cid_distance = cid_distance
14      if closest_cid is None:
15          raise Exception('Error in finding closest '
16                          'distance doc_id:{0}'.format(did))
17      return closest_cid, closest_cid_distance
```

5.2.3. Classification

Classification is run in the final phase of application. It consist of two parts - kNN and SVM. First one is used for general decision that narrows the possible class memberships on which the second one will work on, giving more precise decision.

kNN

kNN's classification is run in parallel in seperate processes. Main process is responsible for initializing communication, launching processes and dividing work between them. It bases on informations provided by clusterization phase described in detail in section 5.2.2.

kNN goal is to find k nearest number for selected doument. k number is customizable and can be changed in configuration file under *classification/k* key. Document can be pointed out in configuration file as well by assigning it's ID to *classification/new_doc_start_id* key. In current implementation documents for classification can be only read from database.

New document, before being classified, must be prepared in the same way as documents before distance calculations were parsed. It is done in *_prepare_new_doc()* method presented in listing 5.15.

Next, each of worker processes (listing 5.16) receives a sub set of existing documents to calculate distances between them and the new document. Resulst are sent back to the main process, were thay are aggregated and the k closest ones are selected. Label for new document is created basing on the most frequent membership in documents in the selected group.

Listing 5.14. Main.classify() - Main process method for conducting kNN classification

```
1   def classify(self):
2       Db.init()
3       session = Db.create_session()
4       docs = session.query(Models.Doc).filter(
5           Models.Doc.id == int(CONF['classification']['new_doc_start_id'])
6       )
7       if docs.count():
8           for doc in docs:
9               new_doc = self._prepare_new_doc(doc)
10              class_distances = multiprocessing.Array('d', (largest_id + 1))
11              class_ps = []
12              for i in range(PROCESSES):
13                  class_p = Classification(
14                      iteration_offset=i,
15                      iteration_size=PROCESSES,
16                      class_distances=class_distances,
17                      largest_id=largest_id,
18                      parsed_docs=parsed_docs,
19                      new_doc=new_doc,
20                  )
21                  class_p.start()
22                  class_ps.append(class_p)
23
24              for class_p in class_ps:
```

```
                class_p.join()

            id_dist = []
            for i in range(largest_id + 1):
                try:
                    item = {
                        'id': i,
                        'distance': class_distances[i],
                        'class': parsed_docs[i].center_id
                    }
                    id_dist.append(item)
                except KeyError:
                    pass

            # finding most frequent center in close neighborhood
            id_dist.sort(key=lambda x: x['distance'], reverse=True)
            k_id_dist = id_dist[:int(CONF['classification']['k'])]
            classes = [c['class'] for c in k_id_dist]
            counted_classes = Counter(classes)
            new_doc.center_id, _ = counted_classes.most_common(1)[0]
            LOG.info('New doc ({0}) classified as belonging to {1} : {2}'.
                     format(new_doc.title, new_doc.center_id,
                     parsed_docs[new_doc.center_id].title))
            print([parsed_docs[doc].title for doc in parsed_docs if
                   parsed_docs[doc].center_id ==
                   new_doc.center_id])

    else:
        LOG.info('No documents to classify')
```

To be able to find closest neighbors to new document, it must be prepared. It is done in *_prepare_new_doc()* presented in listing 5.15. Document is transformed into Page class object and its feature matrix is created. For each token a TF-IDF must be calculated. TF value is calculated for a single document, so no further action was required. In the case of IDF it was necessary to look up token's IDF value in existing structures, as by rule this value is releated to word frequency in whole corpus.

Listing 5.15. prepare_new_doc() - Method used in classification for parsing new document

```
def _prepare_new_doc(self, doc):
    page = Page()
    page.title = doc.title
    page.content = doc.text
    page.create_tokens()
    # import tokens IDF values from already classified documents
    for page_token in page.tokens:
        try:
            page_token.idf = tokens_idf[page_token.stem]
        except KeyError:
            # token did not appear in previous documents
            page_token.idf = 1 + math.log((len(parsed_docs) + 1) / 1.0,
                                          math.e)
        finally:
            page.calc_tokens_tfidf()
    return page
```

Listing 5.16. Classification.run() - Classification class process main method

```
def run(self):
    doc_id = self.iteration_offset
    while doc_id < (self.largest_id + 1):
        try:
            existing_doc = self.parsed_docs[doc_id]
            distance = Utils.calc_distance(self.new_doc, existing_doc)
            self.class_distances[doc_id] = distance
```

```
8           except:
9               # there is no document with such ID, distance is -1
10              self.class_distances[doc_id] = -1
11
12          doc_id += self.iteration_size
```

SVM

In this implementation the SVM algorithm works on a narrowed corpus that was selected during the kNN classification phase. According to presented methodology (3.4.3) the *one-versus-one* approach was chosen to be incorporated in this system. The function (5.17) is run in the main process of application and is responsible for data partitioning and scheduling tasks between worker processes. It generates all possible combinations of pairs of classes and assigns documents to chosen classes creating a task that is then put to the queue. Once worker process has finished working on a task, it sends back results to master process to aggregate. After receiving all results, main process choses the most frequent result as a label for the new document.

Listing 5.17. Main.classify_svm() - Main process method for conducting SVM classification

```
1   def classify_svm(self):
2       if len(self.k_classes) < 2:
3           LOG.info('There␣is␣only␣one␣possible␣class,␣not␣need␣to␣run␣SVM')
4           return
5
6       LOG.debug('Document␣classes␣for␣SVM:␣{0}'.format(self.k_classes))
7
8       ### Gather all documents, grouped into classes
9       classes_doc = {}
10      for class_id in self.k_classes:
11          # select *ALL* documents that belong to classes indicated by kNN
12          docs_id_in_class = [parsed_docs[doc_id].id for doc_id in
13                              parsed_docs if
14                              parsed_docs[doc_id].center_id == class_id]
15          classes_doc[class_id] = docs_id_in_class
16
17      pair_queue = multiprocessing.Queue()
18      result_queue = multiprocessing.Queue()
19      results = {}
20      svm_ps = []
21      for pid in range(PROCESSES):
22          svm_p = SVM(
23              pair_queue=pair_queue,
24              result_queue=result_queue,
25              classes_doc=classes_doc,
26              parsed_docs=parsed_docs,
27              new_doc=self.new_doc
28          )
29          svm_p.start()
30          svm_ps.append(svm_p)
31
32      # generate n(n-1)/2 class pairs
33      combinations = itertools.combinations(self.k_classes, 2)
34
35      for pair in combinations:
36          LOG.debug('Sending␣class␣pair:␣{0}'.format(pair))
37          pair_queue.put(pair)
38
39      for i in range(PROCESSES):
40          pair_queue.put(None)
41
42      not_finished = PROCESSES
43      while not_finished:
```

```
44          res = result_queue.get()
45          if not res:
46              not_finished -= 1
47              continue
48          LOG.debug('Received␣SVM␣pair:␣{0}␣{1}␣with␣result␣{2}'.format(
49              res['class1'], res['class2'], res['result']))
50          try:
51              results[res['result']] += 1
52          except:
53              results[res['result']] = 1
54
55      class_id, _ = sorted(results.items(), key=operator.itemgetter(1))[-1]
56      LOG.info('SVM␣group␣id:␣{0}␣({1})'.format(
57          class_id,
58          parsed_docs[class_id].title),
59      )
60
61      for svm_p in svm_ps:
62          svm_p.join()
63
64      LOG.info('Finished␣ciassification')
```

Method presented on listing 5.18 is the main entry point for every worker process. It invokes *_svm_pair()* (5.19) function as long as *_recv()* (5.20) returns value different than None.

Listing 5.18. SVM.run() - SVM class process main method

```
1  def run(self):
2      while self._recv():
3          self._svm_pair()
```

_recv() method (5.19) conducts receiving tasks for given worker thread as long as there is any object in the task queue. In this case, the None value is returned.

Listing 5.19. SVM._recv() - SVM class process main method

```
1  def _recv(self):
2      val = self.pair_queue.get()
3      if not val:
4          self.result_queue.put(None)
5          return None
6      self.current_pair = val
7      return val
```

Method presented in the listing 5.20 represents the main logic of parallelized SVM algorithm execution. Worker processes are given a pair of classes. Basing on linear kernel, SVM will classify the new document to one of them. Classification begins with creating a feature matrix that includes features gathered from existing documents that belong to one of two classes and the new document. Once matrix is prepared, SVM classifier is trained on old documents, which eventually allows new document to be classified. Most matching class is selected and the decision is sent back to master process for further evaluation.

Listing 5.20. SVM._svm_pair() - SVM class process main method

```
1  def _svm_pair(self):
2      class1, class2 = self.current_pair
3
4      class1_docs = self.classes_doc[class1]
5      class2_docs = self.classes_doc[class2]
6      all_docs = []
7      all_docs.extend(class1_docs)
8      all_docs.extend(class2_docs)
9
10     tokens_template = {}
```

```
    class_distribution = []
    # create template for existing documents
    for doc_id in all_docs:
        doc = self.parsed_docs[doc_id]
        class_distribution.append(doc.center_id)
        for token in doc.tokens:
            tokens_template[token.stem] = 1

    for token in self.new_doc.tokens:
        tokens_template[token.stem] = 1
    tokens_template = list(tokens_template.keys())

    feature_matrix = []
    for doc_id in all_docs:
        feature_vector = []
        doc = self.parsed_docs[doc_id]
        for token in tokens_template:
            try:
                feature_value = doc.tfidf[token]
            except:
                feature_value = 0.0
            feature_vector.append(feature_value)
        feature_matrix.append(feature_vector)

    new_feature_vector = []
    for token in tokens_template:
        try:
            feature_value = self.new_doc.tfidf[token]
        except:
            feature_value = 0.0
        new_feature_vector.append(feature_value)

    class_distribution = [class1]*len(class1_docs)+[class2]*len(class2_docs)
    clf = svm.SVC(kernel='linear', C=1.0)
    clf.fit(feature_matrix, class_distribution)
    result = clf.predict(new_feature_vector)

    # send results
    self.result_queue.put({
        'class1': class1,
        'class2': class2,
        'result': result[0]
    })
```

5.3. Conclusion

This chapter presented detailed implementation of the system. It describes flow of the application in sequence diagrams and presents possible use cases. Attached class diagram reflects relation between logical blocks of code used in the system. Source code for all algorithms was included and described.

Chapter 6

Tests and results

6.1. Introduction

This chapter focuses on tests that were run to provide feedback on performance of implemented system and the accuracy of used algorithms with different parameters.

6.2. Performance of algorithms in parallel processing

First set of tests includes performance measurements on 32-core processor. Each part of the implementation was run in multiple configurations to observe effectiveness of parallelization.

Tests were using corpuses that contained 10, 100, 1000 and 10000 documents and were run on range of processor cores from 1 to 20.

Each test case was run 10 times and the average of measured times was used for final results. In order to ensure consistency between tests, assumptions were made:

- k=10 for kNN
- k=10 for k-means

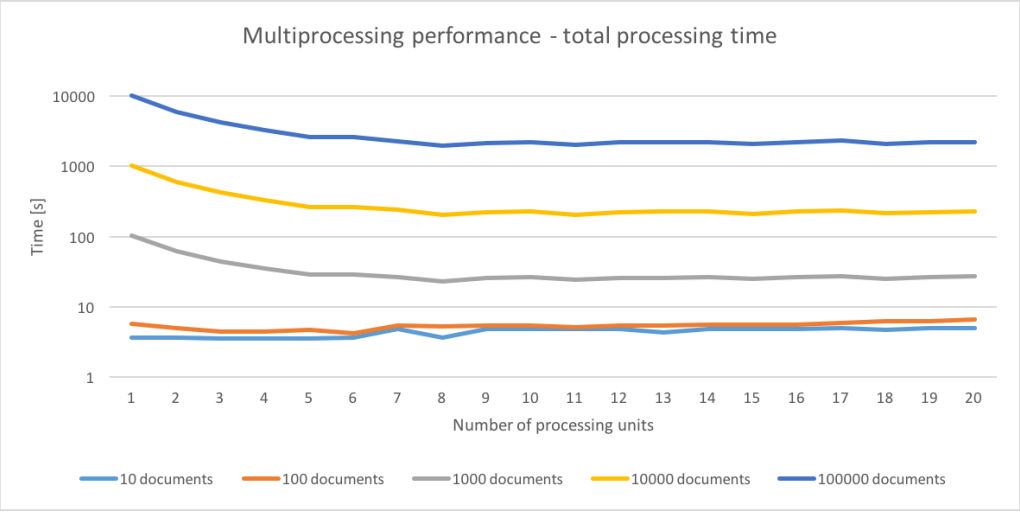


Figure 6.1. Overall system multiprocess performance

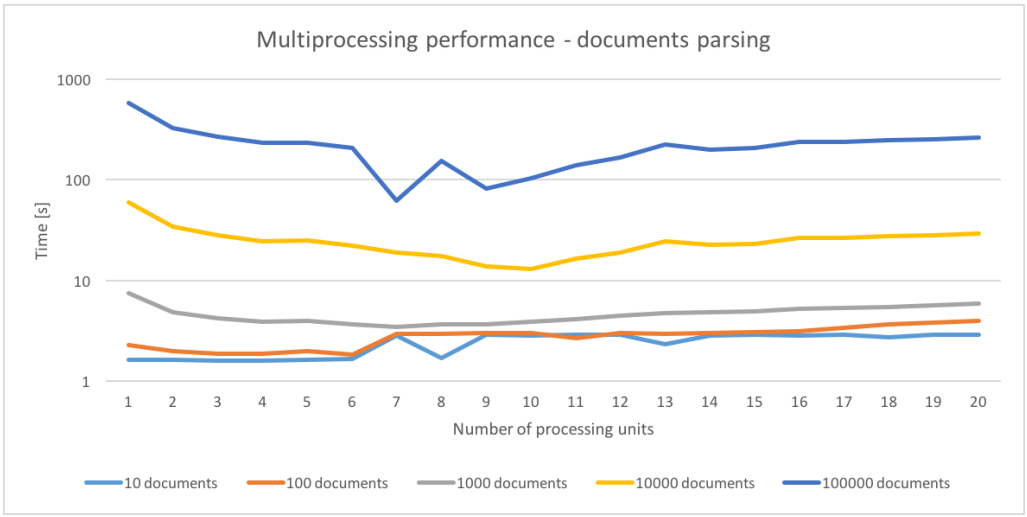


Figure 6.2. Document parser module multiprocess performance

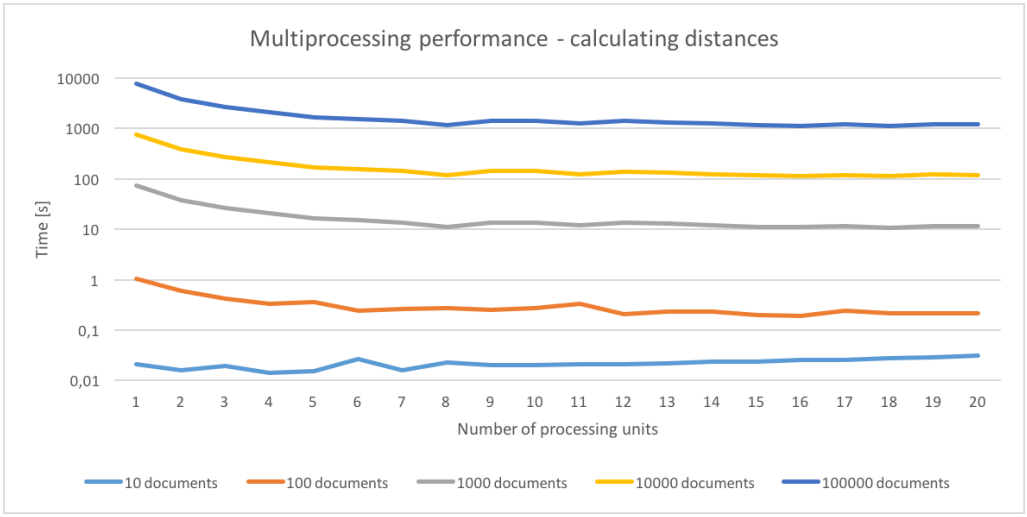


Figure 6.3. Distance calculation module multiprocess performance

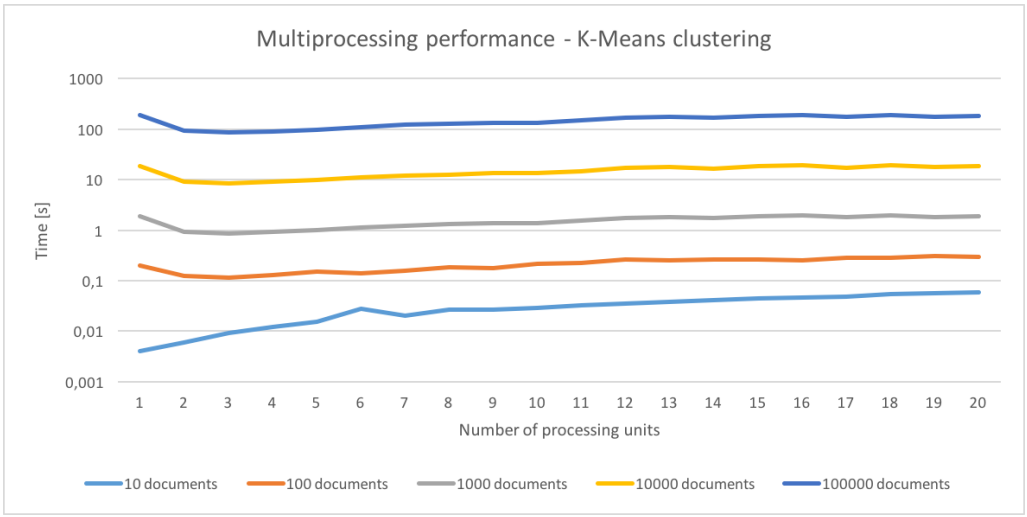


Figure 6.4. K-means clustering module multiprocess performance

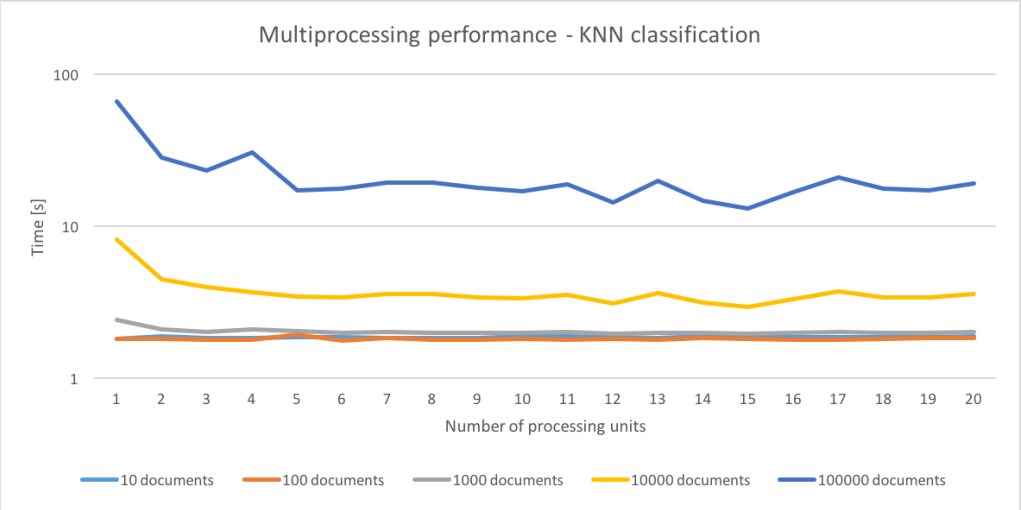


Figure 6.5. kNN classification module multiprocess performance

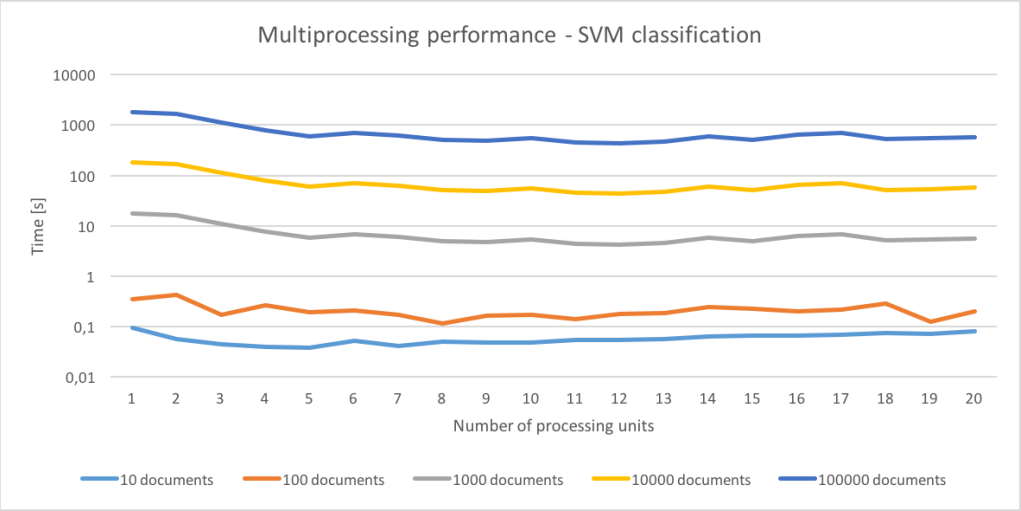


Figure 6.6. SVM classification module multiprocess performance

Across all performmance tests presented on charts above (6.1, 6.2, 6.3 6.4, 6.5, 6.6, 6.7, 6.8 and 6.9) it can be noticed that the best performance gain is when around 6 worker processes are working upon completing tasks. Using more workers than that results in no further gains and constant processing time, or even makes processing longer. Main limiting factor that has to be taken into account is communication overhead - a lot of data has to be transfered between processes and at some point master process is not efficent enough to schedule tasks and receive results from workers.

6.3. Classification accuracy

For testing classification accuracy true-positive ratio (TP, 5) and false-negative ratio (FN, 3) values were calculated for each of presented classification algorithms. Defining whether document has been classified properly by kNN and SVM algorithms was conducted by external expert with arbitrary knowledge.

In order to ensure consistency between tests, some assumptions were made:

- k=20 for kNN
- k=300 for k-means
- 1000 documents in corpus

Table 6.1. kNN and SVM effectiveness with different K value in k-means

K value in k-means		75	150	300
kNN	TP	0,79	0,81	0,81
	FN	0,21	0,19	0,19
SVM	TP	0,8	0,78	0,79
	FN	0,2	0,22	0,21

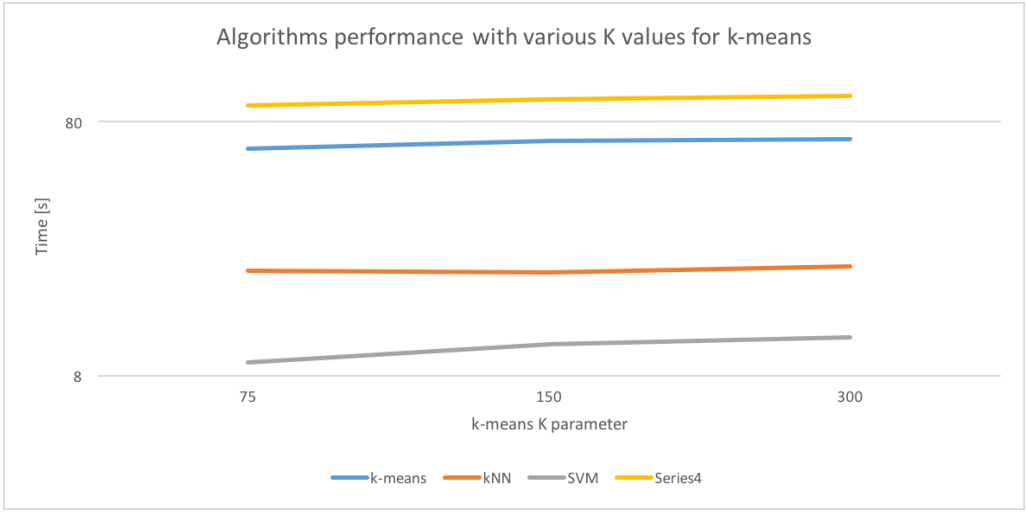


Figure 6.7. Algorithms performance with various K values for k-means algorithm

Test results indicate that increasing k parameter in k-means brings small recall improvement for both kNN and SVM classificators, however the change is marginal (6.1). Also, increasing k does not affect processing time.

Table 6.2. kNN and SVM effectiveness with different K value in kNN

K value in kNN		10	20	40
kNN	TP	0,74	0,79	0,82
	FN	0,26	0,21	0,19
SVM	TP	0,67	0,71	0,79
	FN	0,33	0,29	0,21

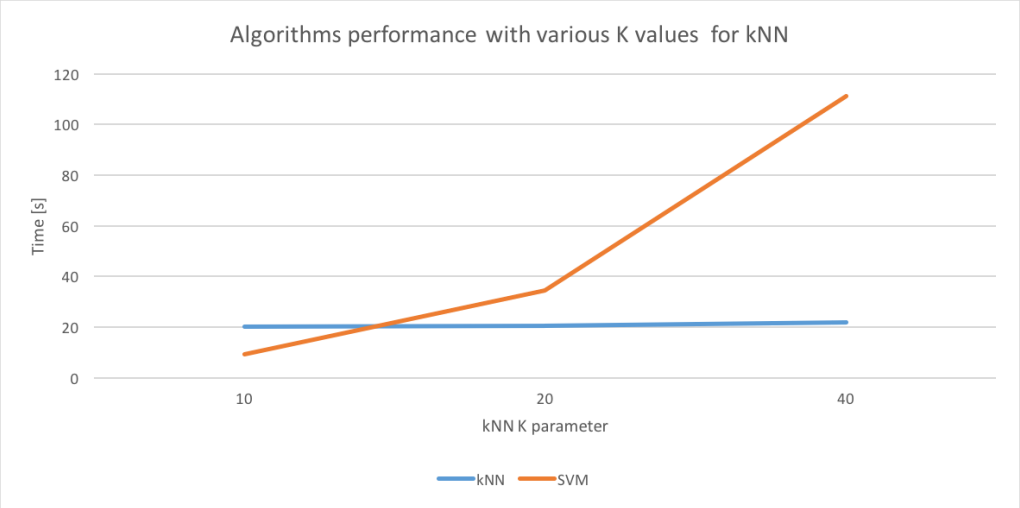


Figure 6.8. Algorithms performance with various K values for kNN

Changing k parameter in kNN has more influence on results. Increasing it from 10 to 40 allows to achieve better recall for kNN by 8% and for SVM by 12%. Also, for kNN it does not affect total processing time, but in case of SVM it extends it significantly (21 times in presented test case).

Table 6.3. SVM effectiveness with different kernels

SVM kernel		linear	rbf	poly
SVM	TP	0,7	0,71	0,71
	FN	0,3	0,29	0,29

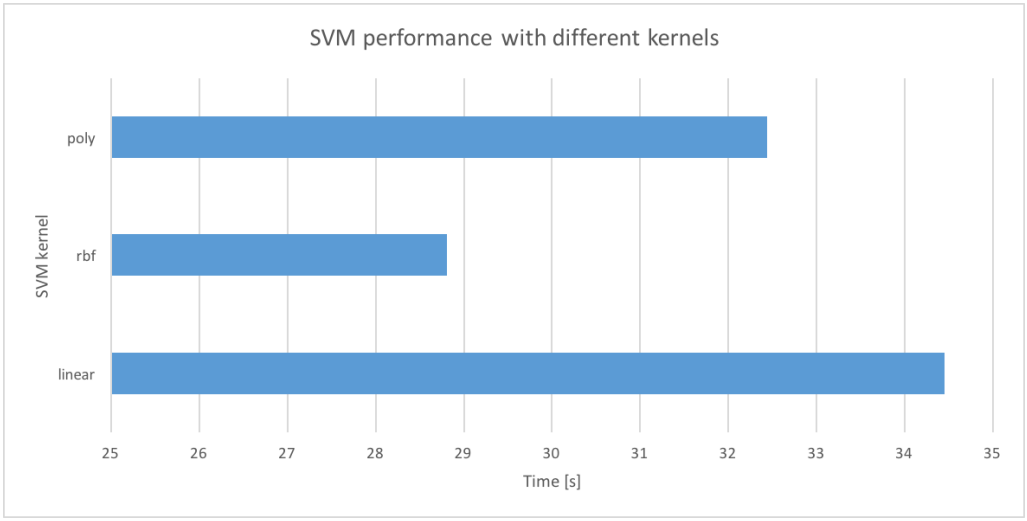


Figure 6.9. Algorithms performance with various SVM kernels

Last case tested whether there can be gained recall when changing kernels with which SVM works. Tests indicates, that final results were not affected by these changes, but there were significant processing time fluctuations.

6.4. Conclusion

Thic chapter outlines test cases used to verify system performance and classification algorithms accuracy with different configurations. Tests were conducted in a function of number of clusters and number of nearest neighbours and SVM kernels.

Chapter 7

Conclusions

7.1. Project overview

The aim of this project was to create a platform that would allow for text document classification in parallel envoronment. Such platform was to classify documents using multi-tier approach, with first layer giving general decision basing on which the second layer would give the precise answer. In order to deliver this goal, many other features which are described in this paper had to be introduced to the system. Conducted research shows that proposed implementation scores acceptaple accuracy when classifying documents and that it can be scaled using multiple processes.

7.2. Future work

Although the results are promising, they are only preliminary and should be furtherly confirmed.

Test results indicated performance bottleneck that could be removed in future release. Main performance issue is having a single master process. While worker processes were quite efficent in executing their tasks, master process was burdened with scheduling tasks and aggregating received results. Many memory operations and need to communicate data between processes created overhead that could not be easily shadowed by sheer computing power of multiple worker processes.

Fixing this issue would require changing architecture from master-slave into a processing tree or map-reduce. Introducing multiple master processes instead of one would offload work and distribute it not only in width but also in depth. Master processes would create branches and would take care of preliminary data aggregation and send back to main master process only important results that would be crucial for whole application.

7.3. Conclusion

All tasks set for this project were completed. Created system allows for text document classification with multi-tier classifiers and the whole flow is executed in parallel using multiprocessing techniques. Based on results of performed tests the conclusions were formed which may help in the future extensions of this application and works.

List of Figures

List of Tables

Listings

Bibliography

[1] Julian Szymański, Marcin Roman, Grzegorz Borczuch, Radosław Szulgo, "AUTOMATYCZNA KATEGORYZACJA ARTYKUŁÓW WIKIPEDII", Politechnika Gdańska, Wydział Elektroniki Telekomunikacji i Informatyki, Katedra, Architektury Systemów Komputerowych, ZESZYTY NAUKOWE WYDZIAŁU ETI POLITECHNIKI GDAŃSKIEJ, nr 8/2010

[2] Number of articles in English Wikipedia, https://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia, accessed on 09.07.2015

[3] Roger C. Barr, Loren W. Nolte, and Andrew E. Pollard, "Bayesian Quantitative Electrophysiology and Its Multiple Applications in Bioengineering". IEEE Rev Biomed Eng. 2010; 3: 155–168. doi: 10.1109/RBME.2010.2089375

[4] Lovrek, Ignac, Howlett, RobertJ, Jain, LakhmiC, "Comparing Document Classification Schemes Using K-Means Clustering"

[5] Hastie, Trevor; Tibshirani, Robert; Friedman, Jerome (2009). "The Elements of Statistical Learning: Data Mining, Inference, and Prediction". http://statweb.stanford.edu/ tibs/ElemStatLearn/ access 23.08.2015

[6] Usama Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, "From Data Mining to Knowledge Discovery in Databases", http://www.kdnuggets.com/gpspubs/aimag-kdd-overview-1996-Fayyad.pdf access 19.08.2015

[7] Kayvan Najarian, Robert Splinter, "Biomedical Signal and Image Processing", Taylor & Francis, CRC Press, 2006, FL 33487-2742

[8] Karl Ingason, Sigrún Helgadóttir, Hrafn Loftsson, Eiríkur Rögnvaldsson, "A Mixed Method Lemmatization Algorithm Using a Hierarchy of Linguistic Identities (HOLI)", A. Ranta, B. Nordström (Eds.): GoTAL 2008, LNAI 5221, pp. 205–216, 2008. c Springer-Verlag Berlin Heidelberg 2008

[9] Sharifloo, A. A. and Shamsfard, M. 2008. "A Bottom up Approach to Persian Stemming", in the Proceedings of the Third International Joint Conference on Natural Language Processing. Hyderabad, India.

[10] M.F.Porter, "An algorithm for suffix stripping", Program 14, no. 3, pp 130-137, July 1980.

[11] Rajaraman, A.; Ullman, J. D. (2011). "Data Mining". Mining of Massive Datasets (PDF). pp. 1–17. doi:10.1017/CBO9781139058452.002. ISBN 9781139058452.

[12] Piatetsky-Shapiro, Gregory (1991), Frawley, William J, "Discovery, analysis and presentation of strong rules", Knowledge Discovery in Databases, AAAI/MIT Press, Cambridge, MA.

[13] Florian Verhein, "Frequent Pattern Growth (FP-growth) algorithm", School of Information technologies, The University of Sydney, Australia, June 2008

[14] Michael Steinbach, George Karypis, Vipin Kumar, "A Comparison of Document Clustering Techniques", Department of Computer Science and Egineering, University of Minnesota, Technical Report #00-034

[15] Bhavani Thuraisingham,Latifur Khan,Mamoun Awad,Lei Wang, "Design and Implementation of Data Mining Tools", CRC Press, 18.06.2009

[16] Saravanan Thirumuruganathan, "A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm", May 17, 2010, https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/, access: 15.08.2015

[17] Richard O. Duda, Peter E. Hart, David G. Stork, "Pattern Classification, 2nd Edition", ISBN: 978-0-471-05669-0, November 2000

[18] Introduction to Support Vector Machines
http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html, access: 12.09.2015

[19] Chamasemani, F.F.; Singh, Y.P., "Multi-class Support Vector Machine (SVM) Classifiers – An Application in Hypothyroid Detection and Classification," in Bio-Inspired Computing: Theories and Applications (BIC-TA), 2011 Sixth International Conference on , vol., no., pp.351-356, 27-29 Sept. 2011

[20] Jonathan Milgram, Mohamed Cheriet, Robert Sabourin. "One Against One" or "One Against All": Which One is Better for Handwriting Recognition with SVMs?. Guy Lorette. Tenth International Workshop on Frontiers in Handwriting Recognition, Oct 2006, La Baule (France), Suvisoft.

[21] Deshpande, V.P.; Erbacher, R.F.; Harris, C., "An Evaluation of Naive Bayesian Anti-Spam Filtering Techniques," in Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC , vol., no., pp.333-340, 20-22 June 2007

[22] Roc curve example https://docs.eyesopen.com/toolkits/cookbook/python/plotting/roc.html, access: 01.09.2015

[23] Thomas C.W. Landgrebe, Robert P.W. Duin, "Approximating the multiclass ROC by pairwise analysis", Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, P.O. Box 5031 2600 GA Delft, Mekelweg 4 2628CD Delft, The Netherlands Received 1 September 2006; received in revised form 18 March 2007 Available online 16 May 2007

[24] Scale out and scale down approach architecture diagram, http://www.cisco.com/c/dam/en/us/solutions/collateral/borderless-networks/advanced-services/white_paper_c11-553711.doc/_jcr_content/renditions/white_paper_c11-553711-01.jpg, access:11.09.2015

[25] http://www.cisco.com/c/dam/en/us/solutions/collateral/borderless-networks/advanced-services/white_paper_c11-553711.doc/_jcr_content/renditions/white_paper_c11-553711-01.jpg, access: 08.09.2015

[26] Doug Alexander, Data Mining, http://www.laits.utexas.edu/ anorman/BUS.FOR/course.mat/Alex/, access:02.09.2015

[27] Palmisano WA1, Divine KK, Saccomanno G, Gilliland FD, Baylin SB, Herman JG, Belinsky SA., "Predicting lung cancer by detecting aberrant promoter methylation in sputum.", Cancer Res. 2000 Nov 1;60(21):5954-8.

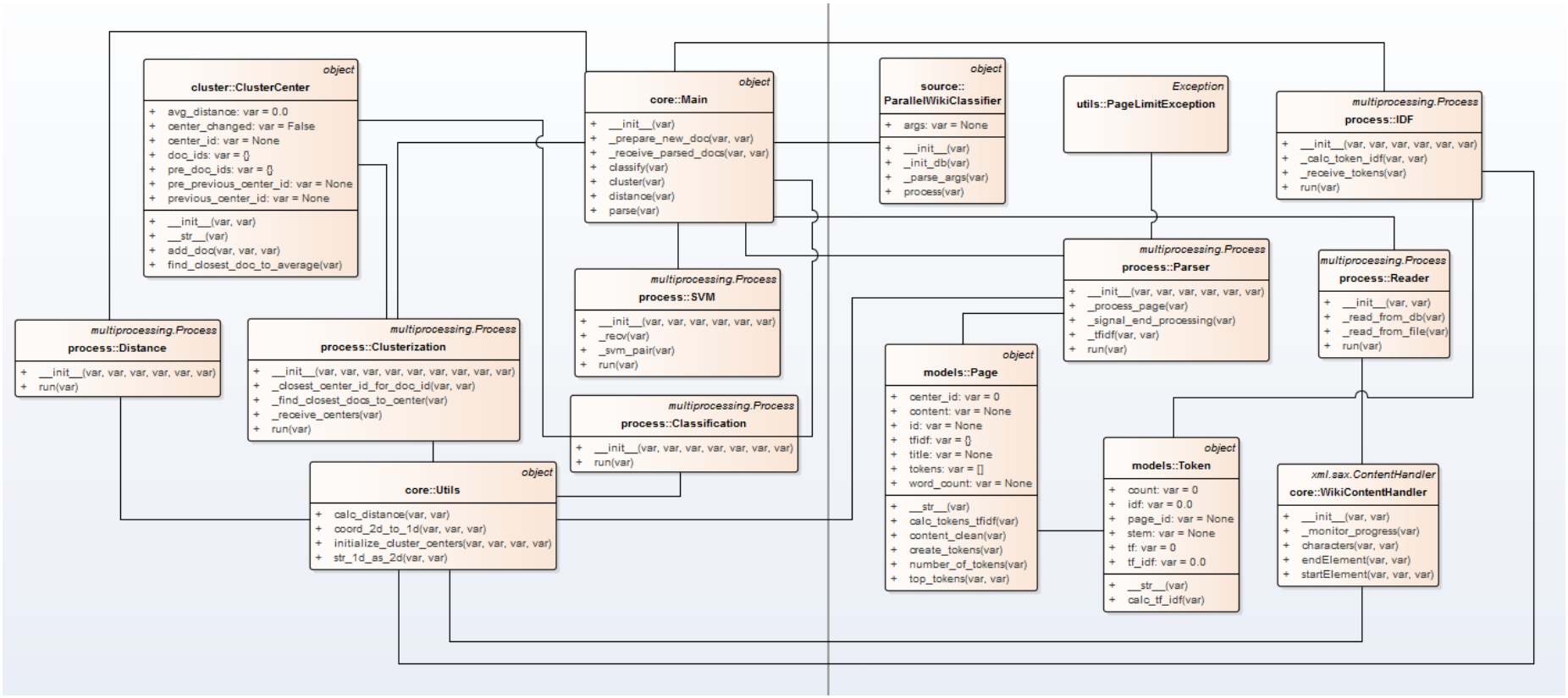
[28] McGrail, Anthony J.; Gulski, Edward; Allan, David; Birtwhistle, David; Blackburn, Trevor R.; Groot, Edwin R. S., "Data Mining Techniques to Assess the Condition of High Voltage Electrical Plant". CIGRÉ WG 15.11 of Study Committee 15.

[29] Victor Alessandrini, "Shared memory application programming. Concepts and strategies in multicore application programming", ISBN: 978-0-12-803761-4, p. 392

[30] Lin, H.-X., Alexander, M., Forsell, M., Knüpfer, A., Prodan, R., Sousa, L., Streit, A., "Euro-Par 2009, Parallel Processing - Workshops", ISBN: 978-3-642-14121-8, p. 381

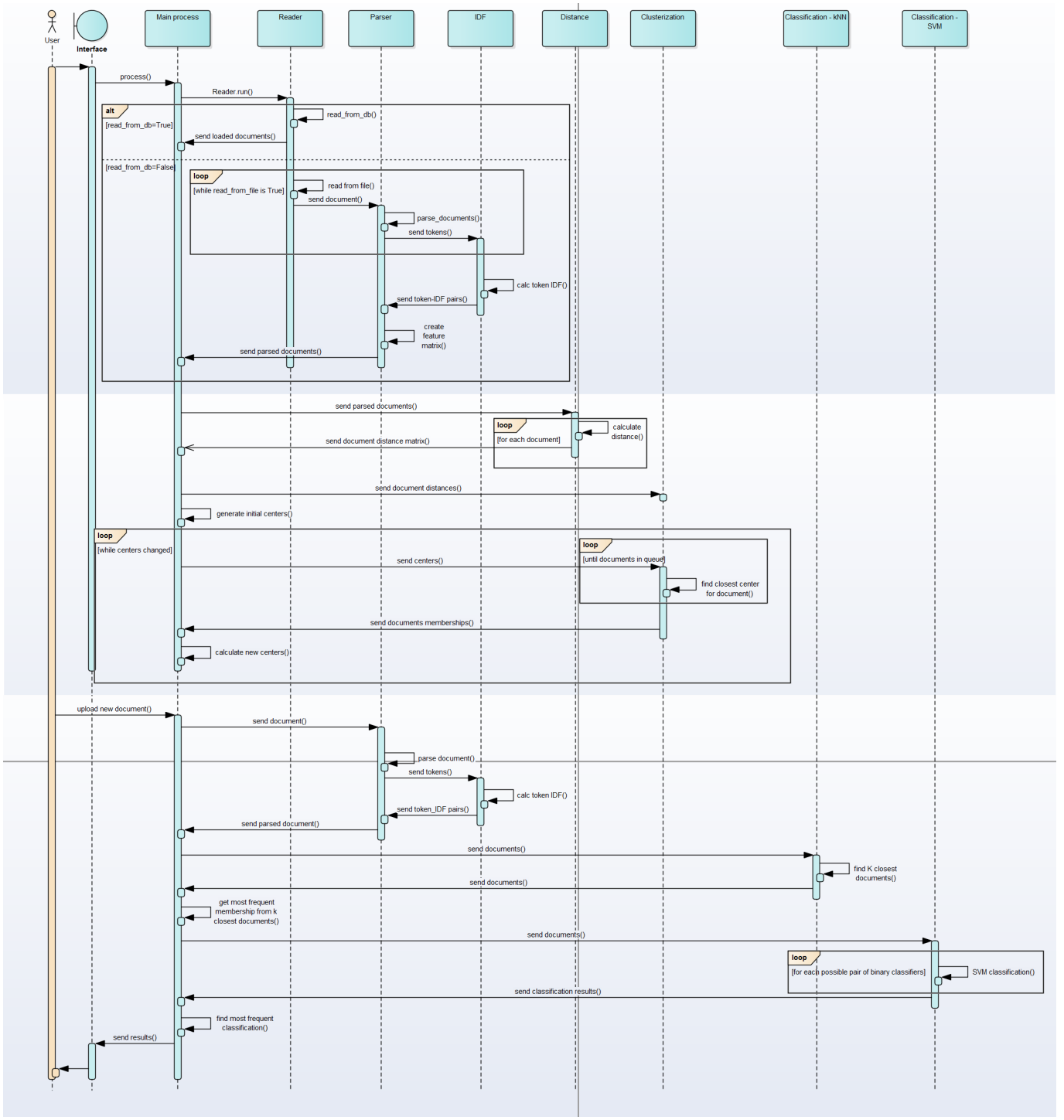
Appendix A

Class diagram


object
cluster::ClusterCenter
+ avg_distance: var = 0.0
+ center_changed: var = False
+ center_id: var = None
+ doc_ids: var = {}
+ pre_doc_ids: var = {}
+ pre_previous_center_id: var = None
+ previous_center_id: var = None
+ __init__(var, var)
+ __str__(var)
+ add_doc(var, var, var)
+ find_closest_doc_to_average(var)
object
core::Main
+ __init__(var)
+ _prepare_new_doc(var, var)
+ _receive_parsed_docs(var, var)
+ classify(var)
+ cluster(var)
+ distance(var)
+ parse(var)
object
source::
ParallelWikiClassifier
+ args: var = None
+ __init__(var)
+ _init_db(var)
+ _parse_args(var)
+ process(var)
Exception
utils::PageLimitException
multiprocessing.Process
process::IDF
+ __init__(var, var, var, var, var, var)
+ _calc_token_idf(var, var)
+ _receive_tokens(var)
+ run(var)
multiprocessing.Process
process::Distance
+ __init__(var, var, var, var, var, var)
+ run(var)
multiprocessing.Process
process::Clusterization
+ __init__(var, var, var, var, var, var, var, var)
+ _closest_center_id_for_doc_id(var, var)
+ _find_closest_docs_to_center(var)
+ _receive_centers(var)
+ run(var)
multiprocessing.Process
process::SVM
+ __init__(var, var, var, var, var, var)
+ _recv(var)
+ _svm_pair(var)
+ run(var)
multiprocessing.Process
process::Classification
+ __init__(var, var, var, var, var, var)
+ run(var)
multiprocessing.Process
process::Parser
+ __init__(var, var, var, var, var, var)
+ _process_page(var)
+ _signal_end_processing(var)
+ _tfidf(var, var)
+ run(var)
multiprocessing.Process
process::Reader
+ __init__(var, var)
+ _read_from_db(var)
+ _read_from_file(var)
+ run(var)
object
core::Utils
+ calc_distance(var, var)
+ coord_2d_to_1d(var, var, var)
+ initialize_cluster_centers(var, var, var, var)
+ str_1d_as_2d(var, var)
object
models::Page
+ center_id: var = 0
+ content: var = None
+ id: var = None
+ tfidf: var = {}
+ title: var = None
+ tokens: var = []
+ word_count: var = None
+ __str__(var)
+ calc_tokens_tfidf(var)
+ content_clean(var)
+ create_tokens(var)
+ number_of_tokens(var)
+ top_tokens(var, var)
object
models::Token
+ count: var = 0
+ idf: var = 0.0
+ page_id: var = None
+ stem: var = None
+ tf: var = 0
+ tf_idf: var = 0.0
+ __str__(var)
+ calc_tf_idf(var)
xml.sax.ContentHandler
core::WikiContentHandler
+ __init__(var, var)
+ _monitor_progress(var)
+ characters(var, var)
+ endElement(var, var)
+ startElement(var, var, var)

Appendix B

Sequence diagram


User
Interface
Main process
Reader
Parser
IDF
Distance
Clusterization
Classification - kNN
Classification - SVM
process()
Reader.run()
alt
[read_from_db=True]
read_from_db()
send loaded documents()
[read_from_db=False]
loop
[while read_from_file is True]
read from file()
send document()
parse_documents()
send tokens()
calc token IDF()
send token-IDF pairs()
create feature matrix()
send parsed documents()
send parsed documents()
loop
[for each document]
calculate distance()
send document distance matrix()
send document distances()
generate initial centers()
loop
[while centers changed]
loop
[until documents in queue]
send centers()
find closest center for document()
send documents memberships()
calculate new centers()
upload new document()
send document()
parse document()
send tokens()
calc token IDF()
send token_IDF pairs()
send parsed document()
send documents()
find K closest documents()
send documents()
get most frequent membership from k closest documents()
send documents()
loop
[for each possible pair of binary classifiers]
SVM classification()
send classification results()
find most frequent classification()
send results()

Appendix C

Instructions

C.1. Installation

1. Install prerequisites: Python 3.5 and git
2. Clone code repository using command

   ```
   git clone git@github.com:macsz/parallel-wiki-classifier.git
   ```

3. To install all required libraries, execute command:

   ```
   pip install -r source/requirements.txt
   ```

4. Install dictionaries for lemmatization and stop words removal:

   ```
   python -c "import␣nltk;␣nltk.download('stopwords');␣nltk.download('wordnet')"
   ```

C.2. Usage