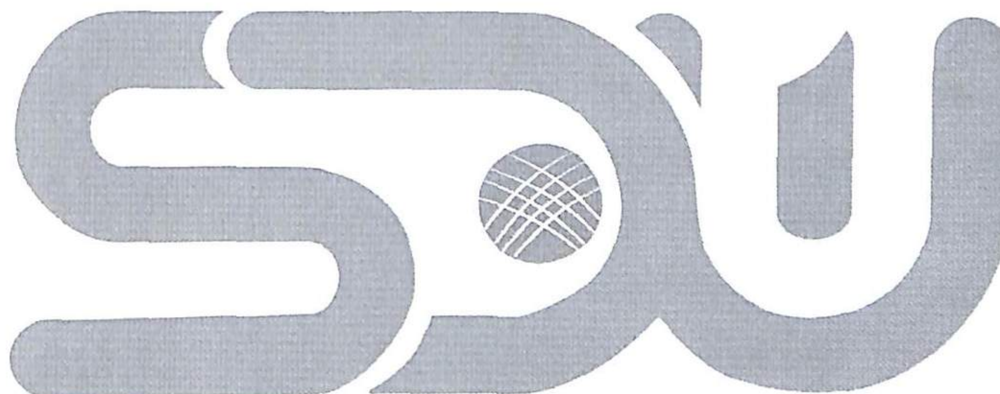
MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN
SULEYMAN DEMIREL UNIVERSITY

UDK 004.4



BEGADIL SERIK

Development of software applications in the Smart-Campus system

Specialty: "6M070400 – Computing systems and software"

Academic Degree: Master of Computer Sciences

"Admitted to defense":
Dean of Faculty
Assist. Prof., PhD Zhaparov M.
«9» 06 2018г.

Head of Department — Assist. Prof.,PhD. Alimanova M.U.

Scientific advisor — Dr. tech.sc.professor Amirgaliyev Y.N.

Kaskelen, 2018

ABSTRACT

One of the urgent tasks of the modern information society is to find documents that are partially and completely similar to each other. The ability to determine the similarity between documents can improve quality of clustering documents by content, search engines by removing unnecessary information, identifying copyright infringements, filtering search and mail spam. Huge amount of data makes its direct solution (by pairwise comparing the texts of documents) practically impossible at reasonable time. Therefore, faster document comparison algorithms are required. At the same time, algorithms that successfully work for a specific statement of the problem are inapplicable or give poor results for other problems.

This master's thesis explains the work of an application that searchs similarities between documents. Documents can be in pdf, doc, docx, jpeg, png or jpg format and should be in Russian, English or Kazakh. If the document format is jpeg, png or jpg, then the application converts text information from image to text using OpenCV library, then performs a search from the database. Using this application, users can add documents to the application database and check documents for duplicates. For example, teachers can add student courseworks, theses, scientific articles to the database, and then check the work of other students for duplicates.

АННОТАЦИЯ

Одной из актуальных задач, возникающих в современном информационном обществе, является нахождение документов, полностью и частично похожих друг на друга. Возможность определения сходства текстовых документов позволяет улучшать качество работы поисковых систем за счет удаления избыточной информации, фильтровать поисковый и почтовый спам, кластеризовать тексты по содержанию, выявлять нарушения авторских прав. В связи с большим объемом текстов прямой поиск по словам - очень трудоемкая задача. Поэтому требуется использование более быстрых алгоритмов сравнения документов. При этом алгоритмы, успешно работающие для конкретной постановки задачи, бывают неприменимы или дают плохие результаты для других задач.

Эта магистерская диссертация поясняет работу приложение, которое ищет сходство между документами. Документы могут быть в формате pdf, doc, docx, jpeg, png или jpg и должны быть на русском, английском или на казахском языке. Если формат документа jpeg, png или jpg, то приложение преобразует текстовую информацию из изображения в текст с помощью библиотеки OpenCV, затем выполняет поиск из базы данных. Используя это приложение, пользователи могут добавлять документы в базу данных приложений и проверять документы на дубликаты. Например, преподаватели могут добавлять курсовые работы студентов, тезисы, научные статьи в базу данных, а затем проверять работу других студентов на дубликаты.

АҢДАТПА

Қазіргі заманауи ақпараттық қоғамда туатын өзекті тапсырмалардың бірі толықтай және жартылай ұқсас құжаттарды анықтау болып табылады. Бір-біріне ұқсас мәтіндік құжаттарды анықтау білу бізге іздестіру жүйесінің сапасын артық ақпараттарды алып тастау арқылы жақсарту, іздестіру жүйесіндегі және пошталық спамдарды сұрыптау, мәтіндерді мазмұны бойынша кластерлеу, авторлық құқықтардың бұзылуын әшкерелеу сияқты мүмкіншіліктерін береді. Үлкен мәтіндердің көлемі салдарынан сөздерді кезек-кезегімен салыстыра отырып тексеру - компьютердің ресурстарын өте көп қажетсінетін жұмыс. Құжаттарды салыстыру үшін компьютердің ресурстарын аз қолдана отырып, тез жұмыс атқаратын алгоритмдерді қолдануымыз қажет.

Бұл магистрлік диссертация ұқсас құжаттарды анықтау бағдарламасының жұмысын түсіндіреді. Бағдарламадағы құжаттар pdf, doc, docx, jpeg, png немесе jpg пішімінде және орыс тілінде, ағылшын немесе қазақ тілінде болуы тиісті. Егер құжат пішімі jpeg, png немесе jpg болса, яғни сурет түріндегі құжат болса, онда бағдарлама суреттегі мәтіндік ақпараттарды анықтап, компьютер түсінетін мәтінге айналдырады. Содан соң деректер қорынан ұқсас құжаттарды іздейді. Осы бағдарламаны қолдана отырып, қолданушылар бағдарламаның деректер қорына жаңа құжаттарды қоса алады және де сол бағдарламаның деректер қорынан құжаттардың телнұсқасын іздей алады. Мысалы, оқытушылар осы бағдарлама арқылы студенттердің курстық жұмыстарын, диссертацияларын, ғылыми мақалаларын деректер қорына қоса алады және де екінші бір студенттің жұмысын телнұсқаға тексере алады.

CONTENTS

INTRODUCTION 7

1 REVIEW OF EXISTING METHODS 10

1.1 Review of existing methods of text recognition 10

1.2 Review of existing methods and systems of searching duplicates 19

1.3 Conclusion for first section 33

2 TEXT RECOGNITION 34

2.1 Edge detection 35

2.2 Finding contours 37

2.3 Perspective transform 37

2.4 Cropping 39

2.5 OCR (Optical Character Recognition) 40

2.6 Conclusion for second section 40

3 SEARCH DUPLICATES 41

3.1 Shingle method 41

3.1.1 Text canonization 41

3.1.2 Splitting into shingles 42

3.1.3 Calculating hashes 43

3.1.4 Comparison, determination of the result 44

3.2 MinHash 47

3.3 TF-IDF with cosine similarity 48

3.4 Bag-of-words model with cosine similarity 49

3.5 Calculating final similarity 50

3.6 Conclusion for third section 52

4 PRACTICAL PART 53

4.1 Selected tools 53

4.1.1 Database 53

4.1.2 Backend 54

4.1.3 Frontend 58
4.1.4 Other tools 60
4.2 UML diagrams 61
4.3 General scheme of work 67
4.4 Experiments, comparisons and results 71
4.5 Conclusion for fourth section 72
CONCLUSION 73
REFERENCES 74

INTRODUCTION

Today one of the urgent tasks of the modern information society is the search for documents that are partially or completely similar to each other. The ability to determine the similarity between documents can solve important practical problems: eliminate identical texts from the output of search engines, identify matching web pages, exclude duplicate information from databases, prevent the publication of identical articles in scientific journals, detect copyright infringement.

Finding duplicates is one of the most important and difficult tasks. Full duplicates of documents can be easily detected with the help of specialized software or by comparing the hash of documents. The main difficulties arise when looking for fuzzy duplicates. The main problem of solving this task is a huge amount of data. Huge amount of data makes its direct solution (by pairwise comparing the texts of documents) practically impossible at reasonable time. Therefore, faster document comparison algorithms are required. At the same time, algorithms that successfully work for a specific statement of the problem are inapplicable or give poor results for other problems. One of the most common methods for reducing the computational complexity of algorithms is the choice of different heuristics. For example, hashing a certain fixed set of "main" words or document sentences, sampling a set of substrings of text, using dactylograms, and etc.

Resistance to small changes and ability to process short documents is the one of the key requirement for the quality of detection algorithm for duplicates.

To identify fuzzy duplicates, we can use the shingles method. To use shingles method we need to divide all the texts under consideration to shingles. The division into shingles can be carried out in different ways. A shingle can contain one letter or one word, or a sentence or several sentences. The division can go one after the other (butt-end), can overlap (overlap), can go some distance from each other, skipping a few characters or words. The quality and accuracy of document matching depends on the size of the shingle. Different methods of dividing the text into shingles can vary greatly in effectiveness. The larger the shingle, the greater the spread of the probability

of borrowing. This leads to incorrect comparison results. If the shingles go one after another (butt-end), then moving them with only one or two symbols can change the whole text beyond recognition for the document comparison system, since the shifted shingles will contain completely different words and symbols. This effect can be avoided by applying the method of imposing shingles (overlap), or a method of selective taking of shingles. If the shingle is small (1 or 2 characters or words), then the accuracy of the comparison can be very high, but the processing power required to implement a comparison system with such shingles may be too large, because one of the main tasks of using shingles is to reduce the complexity of comparing documents. Reduction of the compared text is achieved by encoding the shingle, converting it to some number, for example, using the algorithms CRC, MD5, SHA1, etc. All the shingles of the document form a list of hashes to be compared. By hashes obtained by converting the shingles, the comparison system compares the documents and determines the probability of their coincidence. To measure of how similar obtained set of hashes are we can use Jaccard Coefficient. It is calculated as intersection of sets divided by union of sets. Now we have a one method to compare documents for similarity. But this method is not efficient. Because if we divide documents by 5 words as shingles and our first document contains 30000 words and second document contains 50000 words, then our first set will contain 29996 shingles, second set will contain 49996 shingles. To compare and store them we need to use a lot of computer resources.

To speed up the comparison of the shingles, we can use MinHash algorithm. MinHash algorithm reduces the amount of data that must be stored and compared. For example, we can select 250 random selected shingles to compare. To compare and store we can convert shingles into checksums. Comparing and storing checksums are easier than comparing shingles. Because one shingle can contain a lot of sequence of words or symbols. After converting shingles into checksums, we have 250 integer checksums, instead of 250 randomly selected shingles. So, in order to compare 29996 and 49996 shingles, now we need to compare 250 integer hash values to another 250 integer hash values.

To identify fuzzy duplicate of image file, we need to extract text information from image. To extract text information from image files we can use tools like Tesseract, OpenCV and etc. After extracting we can easily use result of extraction as input of search using shingle method.

The practical significance of the thesis is determined by the effectiveness of the methods, algorithms implemented in the application, allowing to effectively search for the similarity between documents. This in turn can be applied in universities and schools.

The scientific novelty of the thesis is to solve the following problems:

– Effective methods and algorithms for finding copies and duplicates of text are proposed and developed;

– An effective method for recognizing text from an image is developed.

The thesis consists of an introduction, four sections, conclusion and list of used literature. The first section provides an analysis of existing methods and technologies for recognizing texts from images and searching for duplicates, describing their advantages and disadvantages. The second section describes the method of recognizing text from an image, examples are considered. The third section describes in detail the methods and algorithms for finding copies and duplicates of text with examples. In the last, fourth section, the results of experiments and scheme of work of the application are given.

1 REVIEW OF EXISTING METHODS

1.1 Review of existing methods of text recognition

Technologies which are used to recognize text inside images is called Optical Character Recognition (OCR). These technologies are used to convert any image containing written, typed, handwritten or printed text into machine-readable data. In OCR processing, the character recognition is achieved through important steps of segmentation, feature extraction and classification [1]. Optical Character Recognition (OCR) accuracy depends on the quality of input files.

Recognition is widely used to convert books and documents into electronic format, to automate accounting systems in business or to publish text on a web page. OCR allows you to edit the text, search for words or phrases, store it more compactly, display or print material without losing quality, analyze information, and to apply to the text of the electronic translation, formatting or conversion to speech. Optical text recognition is an explored problem in the areas of pattern recognition, artificial intelligence and computer vision.

Optical recognition systems require calibration to work with a specific font; in the early versions for programming, it was necessary to display each symbol, the program could simultaneously work with only one font. Currently, the most common are the so-called "intelligent" systems, with a high degree of accuracy recognizing the majority of fonts. Some OCR systems can restore the original text formatting, including images, columns, and other non-text components.

History

In 1929 Gustav Tauschek received a patent for the method of optical text recognition in Germany, after which he was followed by Paul W. Handel, obtaining a patent for his method in the United States in 1933. In 1935, Tauschek also received a United States patent on his method. The Tauschek machine was a mechanical device that used templates and a photodetector.

In 1950, David H. Shepard, a cryptanalyst from the United States Armed Forces Security Agency, analyzed the task of converting printed messages into

computer language for computer processing, and after analyzing he built a machine that solves this problem. After he received a United States patent, he reported this to the "Washington Daily News" on April 27, 1951 and to "New York Times" on December 26, 1953. Then Shepard founded a company that develops intelligent machines, which soon released the world's first commercial optical character recognition systems.

The first commercial system was installed on the "Reader's Digest" in 1955. The second system was sold to the company "Standard Oil" for reading credit cards for working with checks. Other systems supplied by Shepard were sold in the late 1950s, including a page scanner for the US National Air Force, designed to read and transmit typing messages on typewritten messages. IBM later received a license to use Shepard's patents.

Around 1965, Reader's Digest and RCA began collaborating to create a machine for reading documents using optical text recognition designed to digitize the serial numbers of the Reader's Digest coupons that had returned from advertisements. For printing on documents with the drum printer "RCA", a special OCR-A font was used. The machine for reading documents worked directly with the computer RCA 301 (one of the first semiconductor computers). The speed of the machine was 1500 documents per minute.

Postal Service of the United States since 1965 to sort mail uses machines that work on the principle of optical recognition of text, created on the basis of technologies developed by the researcher Jacob Rabinow. In Europe, the first organization using machines with optical text recognition was the British Post Office. Mail of Canada has been using optical character recognition systems since 1971. At the first stage, the recipient's name and address are read in the Sorting Center of the OCR system and the bar code is printed on the envelope. It is applied with special inks, which are clearly visible in ultraviolet light. This is done to avoid confusion with the address field filled with a person that can be anywhere on the envelope.

In 1974, Raymond Kurzweil created the company Kurzweil Computer Products, and began working on the development of the first optical character

recognition system capable of recognizing text printed in any font. Kurzweil believed that the best application of this technology is the creation of a reading machine for the blind, which would allow blind people to have a computer that can read the text aloud. This device required the invention of two technologies at once: a CCD flatbed scanner and a synthesizer that converts text to speech. The final product was presented January 13, 1976 during a press conference, headed by Kurzweil and the leaders of the National Federation of the Blind.

In 1978, Kurzweil Computer Products began selling a commercial version of the computer program for optical character recognition. Two years later, Kurzweil sold his company to Xerox Corporation, which was interested in further commercializing text recognition systems. Kurzweil Computer Products has become a subsidiary of Xerox, known as ScanSoft.

The first commercially successful program recognizing the cyrillic alphabet was the "AutoR" program of the Russian company "OCRUS". The program began to be distributed in 1992, operated under the DOS operating system and provided an acceptable speed and quality recognition even on personal computers IBM PC / XT with an Intel 8088 processor at a clock speed of 4.77 MHz. In the early 90's, Hewlett-Packard supplied its scanners to the Russian market with the AutoR program. The algorithm "AutoR" was compact, fast and fully font-independent. This algorithm was developed and tested in the late 60's by two young biophysics, graduates of MIPT - G.M. Zenkin and A.P. Petrov. Their method of recognition, they published in the journal "Biophysics" in Number 12, no. 3 for the year 1967. At present, the Zenkin-Petrov algorithm is applied in several applied systems that solve the problem of recognizing graphic symbols. Based on the this algorithm by Paragon Software Group in 1996, PenReader technology was created. G.M. Zenkin continued work on PenReader technology in the company Paragon Software Group.

Optical Character Recognition software

Accurate recognition of latin characters in printed text is currently possible only if clear images, such as scanned printed documents, are available. Accuracy in this formulation of the problem exceeds 99%. Absolute accuracy can be achieved

only by subsequent editing by a person. The problems of recognizing printed and standard handwritten text, as well as printed texts of other formats, are currently the subject of active research.

The accuracy of the methods can be measured in several ways and therefore can vary greatly. For example, if a specialized word is encountered that is not used for the corresponding software, when searching for nonexistent words, the error may increase.

Another widely researched task is handwriting recognition. Nowadays, the accuracy achieved is even lower than for a handwritten printed text. Higher indicators can only be achieved using contextual and grammatical information. For example, during recognition, searching for whole words in a dictionary is easier than trying to identify individual characters from the text. Knowledge of the grammar of the language can also help to determine whether a word is a verb or a noun. Forms of individual handwritten characters may sometimes not contain enough information to accurately recognize all handwritten text.

To solve more complex problems in the field of recognition, as a rule, intelligent recognition systems, such as artificial neural networks, are used.

Optical Character Recognition (OCR) process has several general steps. Every step of OCR is important. If any result of step is not correct then whole OCR process will fail.

– Loading an image from a given source. Generally OCR algorithms supports wide array of file formats like pdf, bmp, jpg, png and tiff. Once the image file is loaded, the OCR algorithm can start to work. Loading files can be scanned documents, photographs, and etc. Regardless of the original format, OCR algorithm will transform uploaded files into other format which is easily accessible and editable for algorithm.

– Extracting important features from image like resolution, inversion and etc. If necessary image must be inverted and rescaled before processing. Because many OCR algorithms only works with predefined background colors, foreground colors and font sizes.

– Image can contain a lot of noises or can be skewed. To improve the quality of image we need to denoise and deskew image using denoising and descewing algorithms.

– Generally OCR algorithms works only with bi-tonal images. Bi-tonal image is image which uses only two colors. For example, black and white. RGB and grayscaled images must be firstly converted to bi-tonal image. The process of converting RGB or grayscaled images to bi-tonal images is called binarization. This step is very important on OCR algorithms. Because if image binarization process is not correct then whole OCR process can fail.

– Finding lines and removing them from images. This step can improve page layout analysis. By finding and removing lines from image we can improve recognition quality to detect tables, for underlined texts and etc.

– Page layout analysis. By help of page layout analysis we can find important areas and their types, positions from image. This step is also called zoning.

– Detection of words and lines with text. It is one of the difficult tasks. Because text information can be on different font sizes, can contain different spaces between lines with texts and between words.

– Characters analysis (combined and broken). Some characters on text can touch each other or can be broken into several parts. To find correct position of characters on text we need to find and detect this kind of cases.

– Character recognition. After image pre-processing steps, every character will be converted to character code. But sometimes this process can produce several character codes for one character. Because some characters are similar to each other. For example, this step of OCR can produce "l", "I", "|", "1" for character "I". To improve character recognition step we can use dictionary.

Nowadays OCR has been succesfull and widely used. For example, automatic number plate reader (SERGEK), signature verification and identification (banks and etc.), car rentals, ticket validation and etc.

There are a lot of free, non-free OCR applications, softwares exists [2]. Examples for free software: Tesseract, Ocrad, CuneiForm, OCRFeeder, GOCR, OCRopus and etc. Some of free softwares also supports command line implementations. Examples for non-free, commercial software: VueScan, OmniPage, Microsoft Office Document Imaging, Nuance PaperPort Professional, ABBYY FineReader, ReadSoft and etc.

CuneiForm

CuneiForm is a free OCR system. It transforms image files and electronic documents like pdf into editable documents without changing document fonts and structure of document. CuneiForm was developed by Cognitive Technologies in 1993. Firstly CuneiForm was a commercial product. In 2008 it became free, an open-source program. It supports about 20 languages.

CuneiForm is able to instantly identify all the standard characters of all sorts of text and fonts. It can recognize books, magazines, newspapers, fax, xerox copies, texts from ancient typewriters and so on, excluding the decorative font and manuscript. In the CuneiForm program code, a number of unique innovative technologies of Optical Character Recognition are used, such as: adaptive recognition using font-independent instructions, neural-analytical normalization networks, cognitive analytics of alternative text interpretation options, special algorithms for dot matrix printer, poor results of photocopying, faxes and typewritten pages and other. CuneiForm can easily downloaded from internet and installed on operating systems like Windows 7, Windows 8, Windows 7, Windows Vista and Windows XP. CuneiForm is able to recreate an absolute copy of the source. Formatting and structuring, indents, footnotes, indices, number and size of columns, paragraphs, arrangement of separate fragments of text, table elements and illustrations, font styles and other elements of font design of original document are preserved.

GOCR

GOCR is free OCR program. It was written by Jörg Schulenburg. It also known as JOCR. It is used to convert image files to text data. GOCR can also translate

barcodes. GOCR supports command-line and different operating systems. It accepts image formats: pgm, ppm, pnm, tga, pbm. Other widely used image formats like png, jpg, tiff, giff, and bmp also acceptable. If user uploads formats like png, jpg, tiff, giff, or bmp, application automatically converts uploaded file formats using netpbm-progs, gzip and bzip2 to pnm, pbm, pgm, ppm or tga.

Ocrad

Ocrad is a free OCR system. Ocrad uses feature extraction method to convert image to text. It accepts image formats like pgm, pbm and ppm. After reading image Ocrad returns text in UTF-8 format or in byte. It can be used as backend of other programs or can be used as stand-alone application. Ocrad supports page layout analyser that can be used to detect text zones from images.

OCRFeeder

OCRFeeder is a program that provides a graphical user interface for optical character recognition systems. OCRFeeder is a free program for the Linux operating system. It can run in command-line mode. It supports image formats like tiff, pnm, pgm, ppm, pbm, png, jpg, jpeg, bmp, gif and etc. It also supports pdf format. OCRFeeder can process input image to improve the quality of recognition using noise filters and etc. After processing input documents it can return result as text file or in formats like odt, html or pdf. OCRFeeder is written in Python and uses libraries and components like PyGTK (for the graphical user interface), ReportLab (to import pdf files), Unpaper (for image processing), PIL (to work with images) and PyeEnchant, PyGtkSpell (to check orthography).

OCRopus

OCRopus is a collection of document analysis programs. It is OCR system for text recognition based on tesseract. It supports FreeBSD, Linux, Mac OS X. OCRopus is developed under the lead of Thomas Breuel from the German Research Centre for Artificial Intelligence in Kaiserslautern, Germany and was sponsored by Google [3]. It supports a large number of fonts and languages. OCRopus by default comes with English model. If necessary OCRopus can be train for new language

models or for new characters. To recognize text it uses recurrent neural networks. Also OCRopus can be used on command-line mode.

Tesseract

Tesseract is a free computer program for text recognition. Tesseract was originally developed at Hewlett-Packard Laboratories Bristol and at Hewlett-Packard Co, Greeley Colorado between 1985 and 1994, with some more changes made in 1996 to port to Windows, and some C++izing in 1998. In 2005 Tesseract was open sourced by HP. Since 2006 it is developed by Google [4]. Tesseract can recognize more than 100 languages. To recognize other languages Tesseract can be trained. Tesseract can return result of recognition as text or on formats like pdf, html, tsv. To get better recognition results, input image quality needs to be improved. Improvement can done by using OpenCV library or other tools like ImageMagick, Leptonica. Improvements steps contain: rescaling, binarization, noise and border removal, rotation and deskewing. By default Tesseract uses Leptonica to process image before doing actual OCR.

How Tesseract works:

– Analyzing contours and storing it

– Gathering contours as blobs

– Organizing blobs into text lines

– Breaking text lines into words

– Trying to recognize each word in turn at first pass of process of recognition

– Passing words which are satisfactory to adaptive trainer

– Trying to recognize not satisfactory words from first pass of recognition process on second pass using adaptive trainer

– Checking text for small caps and resolving fuzzy spaces

– Outputting digital text

During these processes, Tesseract uses:

– algorithms for detecting proportional and non proportional words (proportional word is a word where all letters are the same width)

– page layout analysis for detecting text lines

– algorithms for chopping joined characters and for associating broken characters

– linguistic analysis for determine the most probable word formed by a cluster of characters

– two character classifiers: a static classifier, and an adaptive classifier which employs training data, and which is better at distinguishing between upper and lower case letters

Example of text recognition

As an example used python implementation of Tesseract called pytesseract for OCR. As image preprocessing tool was used OpenCV library.

First step is image processing. So, we need to convert to grayscale, binarize, find edges, deskew, find text zone and crop image. To find edge from image first we need to convert rgb image to grayscale image using OpenCV function *cv2.cvtColor*. Then to find edges we can use *cv2.Canny* function. After we need to find contours using function *cv2.findContours* and approximate the each contour. We can assume that we have found text zone from image if approximated contour contains 4 points. Now, we need to deskew and denoise image. From deskewed and denoised image we need to find text regions. So, we can blur image and find text regions and crop them.

After cropping process we can send cropped image as input of python implementation of Tesseract OCR called *pytesseract*. Pytesseract returns text. Now we can test Tesseract OCR accuracy. To calculate accuracy we can use python library called *jellyfish*. This library contains different methods to measure difference between two strings like *Jaro distance* and *Jaro-Winkler distance.*

Jaro distance is a string-edit distance that gives a floating point response in (0,1) where 0 represents two completely dissimilar strings and 1 represents identical strings [5].

Jaro-Winkler is a modification/improvement to Jaro distance, like Jaro it gives a floating point response in (0,1) where 0 represents two completely dissimilar strings and 1 represents identical strings [5].

1.2 Review of existing methods and systems of searching duplicates

The search for near duplicates allows us to assume whether two objects are partially identical or not. An object can be understood as text files, webpages and other types of data. There are a lot of methods and algorithms to solve this problem. But some of them works well on small data, some of them on large data. So, we need to know when to use each algorithm.

Currently, software and algorithmic tools aimed at detecting non-original publications are able to solve important practical problems: eliminate identical texts from the output of search engines, identify matching web pages, exclude duplicate information from databases, prevent the publication of identical articles in scientific journals, detect borrowing of other people's results and ideas.

Despite the intensive development of methods for detecting fuzzy duplicates, at the present time it has not been possible to create a universal approach that ensures the best completeness and accuracy in different samples. The complexity of the development is due to the specifics of the processing of textual information: the lack of mechanisms for a logical and mathematical description of the meaning of the material being presented, the large dimension and complexity of the task, the small size of samples, and so on.

A large number of theoretical and experimental studies have now been published, aimed at studying the advantages and limitations of various procedures for identifying fuzzy duplicates [6, 7]. There are sufficiently complete reviews in which a description of the algorithms and the results of their comparative analysis are given [8].

In most publications procedures are generally divided into two classes: syntactic methods (analysis of sequences consisting of symbols, words or sentences) and lexical methods (analysis of informative terms). At the same time, it seems expedient to use more detailed systematization. One such systematization combines methods based on the principle of a general approach to decision-making when identifying fuzzy duplicates: specialized distances (Hamming distance, Levenshtein distance, Damerau–Levenshtein distance, Jaro distance, Jaro-Winkler distance),

shingles and their modifications (super singles, megashingles, Winnowing, SpotSigs), procedures based on the calculation of weights of terms (the method of reference words, I-match), proximity measures (associative coefficients, cosine measure, Monge-Elkan matching scheme, a two-level comparison function based on soft tfidf-weighting) [8, 9, 10].

One of the first studies in the field of finding fuzzy duplicates is the work of U. Manber [11] and N. Heintze [12]. In these works, sequences of neighboring letters are used to construct the sample. The file or document file dactylogram includes all text substrings of fixed length. The numerical value of the dactylograms is calculated using the random polynomial algorithm Rabin-Karp [13]. As a measure of similarity between the two documents, the ratio of the number of common substrings to the size of the file or document is used. A number of methods aimed at reducing the computational complexity of the algorithm are proposed. U. Manber used this approach to find similar files (utility sif), and N. Heintze - to detect fuzzy duplicates of documents (Koala system).

In 1997, A. Broder proposed a new method for assessing the similarity between documents, based on the representation of the document in the form of a set of all possible sequences of fixed length k consisting of neighboring words. Such sequences were called shingles. Two documents are considered similar if their sets of shingles significantly overlapped. Since the number of shingles is approximately equal to the length of the document in words, i.e. is quite large, the authors proposed two sampling methods for obtaining representative subsets [14, 15].

The first method was left only by those shingles, which dactylograms, calculated by the Rabin-Karp algorithm, were divided without a remainder by a certain number m. The main drawback is the dependence of the sample on the length of the document, and therefore documents of small size were represented either by very short samples or not at all. The second method selected only a fixed number s of shingles with the lowest values of the fingerprints or left all the shingles if their total number did not exceeds.

Further development of A. Broder's concepts is the research of D. Fetterly [16]. For each chain, 84 dactylograms are calculated by the Rabin-Karp algorithm using one-to-one and independent functions that use random sets ("minwise independent") of simple polynomials. As a result, each document was represented by 84 shingles, minimizing the value of the corresponding function. Then 84 shingles are divided into 6 groups of 14 (independent) shingles in each. These groups are called "super shingles". If two documents have similarities, for example, $p \sim 0.95$ (95%), then the corresponding 2 super-shingle in them coincide with the probability $p^{14} \sim 0.95^{14} \sim 0.49$ (49%).

Since each document is represented by 6 super-shingles, the probability that at least two super- shingles will match at least two documents is: $1 - (1-0.49)^6 - 6 * 0.49 * (1-0.49)^5 \sim 0.90$ (90%). For comparison: if two documents have a similarity only $p \sim 0.80$ (80%), then the probability of coincidence of at least two super-shingles is only 0.026 (2.6%).

Thus, for effective verification of the coincidence of at least 2 super-shingles, each document is represented by all possible pairwise combinations of 6 super-shingles, which are called "mega-shingles". The number of such mega-shingles is 15. Two documents are similar in content if they have at least one mega-shingles.

The key advantage of this algorithm over A. Broder's research is that, firstly, any document (including a very small one) is always represented by a fixed-length vector, and secondly, the similarity is determined by a simple comparison of the vector coordinates and does not require to perform set-theoretic operations as in A. Broder's research.

Another signature approach, based not on syntactic but on lexical principles, was proposed by A. Chowdhury in 2002 and improved in 2004. [17, 18]. The basic idea of this approach is to calculate the I-Match dactylogram for presenting the contents of documents. For this purpose, first a dictionary L is constructed for the original collection of documents, which includes words with mean IDF values, since such words provide, as a rule, more accurate results when fuzzy duplicates are detected. Words with large and small IDF values are discarded.

Then, for each document, a set of U different words is formed, and the intersection of U and the dictionary L is determined. If the size of this intersection is greater than some minimum threshold, then the list of words entering the intersection is ordered and an I-Match signature is calculated for it using hash function SHA1.

Two documents are considered similar if they have the same I-Match signatures. The algorithm has a high computational efficiency, exceeding the parameters of the algorithm A. Broder. Another advantage of the algorithm is its high efficiency when comparing small-sized documents. The main drawback is the instability to small changes in the content of the document.

To overcome this drawback, the original algorithm was modified by the authors, and the possibility of multiple random mixing of the main vocabulary was introduced into it. The essence of the new improvements is as follows. In addition to the basic dictionary L, K different L_1-L_K dictionaries are generated, which are obtained by accidentally removing from the source dictionary some small fixed part of words, which is about 30% -35% of the original volume L.

For each document, instead of one, the (K + 1) I-Match signature is calculated using the algorithm described above, i.e. the document is represented as a vector of dimension (K + 1), and two documents are considered duplicates if they have at least one of the coordinates. If the document undergoes minor changes (of the order of n words), then the probability that at least one of the K additional signatures will remain unchanged will be: $1 - (1-p^n)^K$.

A similar approach is described in the work of W. Pugh [19] from Google. The author suggests that a complete dictionary of the document be divided into a fixed number of word lists using a hash function. For example, using the remainder of dividing the hash code by the number of lists. Then, for each list, a fingerprint is calculated and two documents are considered similar if they have at least one common fingerprint. The author gives estimates of the stability of the algorithm to small changes in the content of the original document. There is no information on the effectiveness of practical application for reasons of confidentiality.

Another signature approach, also based on lexical principles (use of the dictionary) is the method of descriptive words, proposed by S. Ilyinsky, M. Kuzmin, A. Melkov, I. Segalovich [20].

The essence of the algorithm is as follows. First, from the index, a set of N words is selected from a certain rule (N is determined experimentally), called "descriptive" ones. Then each document is represented by an N-dimensional binary vector, where the i-th coordinate is 1 if the i-th "reference" word has in the document a relative frequency above a certain threshold (separately for each "reference" word) and equal to 0 otherwise the case. This binary vector is called the signature of the document. Two documents are similar if they have the same signatures.

General considerations for constructing a set of descriptive words are as follows:

1. The set of words should cover the maximum possible number of documents.
2. The "quality" of the word should be the highest.
3. The number of words in the set should be minimal.

Algorithm for constructing a set and selecting threshold frequencies. Let "frequency" be the normalized document frequency of the word in the TF document, lying in the range 0...1, where 1 is the frequency of the most frequent word in the document. For each word (once), the distribution of documents is based on such an document "frequency". We perform several iterations, each of which consists of two phases (1) and (2). In (1), the coverage of documents in the index is maximized for a fixed (bounded from below) accuracy; in (2) the accuracy is maximized for a fixed covering. After each iteration, we drop the most "bad" words. After the last iteration, leave enough words for a good coverage. This method allows, starting with a sample of hundreds of thousands of words, to leave a set of 3-5 thousand, the calculation of signatures by which with the use of the full-text index is carried out on the billions index for several minutes on the search cluster of Yandex.

There are a lot of algorithms and applications are exists. Some of them are described below with examples.

Shingle method

Shingle method can be used to check text for duplicate content. To use shingles method we need to divide all the texts under consideration to shingles. The division into shingles can be carried out in different ways. A shingle can contain one letter or one word, or a sentence or several sentences. For example, for the text "I did not see them at the station because Almas and Zhalgas arrived at the bus station before noon", the shingle size 5 would be:

- I did not see them;
- did not see them at;
- not see them at the;
- see them at the station;
- them at the station because;
- at the station because Almas;
- the station because Almas and;
- station because Almas and Zhalgas;
- because Almas and Zhalgas arrived;
- Almas and Zhalgas arrived at;
- and Zhalgas arrived at the;
- Zhalgas arrived at the bus;
- arrived at the bus station;
- at the bus station before;
- the bus station before noon;

In shingle method selecting word, character size for shingle can vary greatly in effectiveness. If we choose big size for shingle then it can lead us to incorrect comparison results. If the shingles size too small, then the accuracy of comparison can be high, but it needs more processing powers. Because of shingle method is used to reduce the complexity of comparing two texts, data, documents, we need to select shingles size more accurately.

Simple shingle method for duplicate detection contains several steps:

- canonization of text – process of removing useless signs, words, prepositions, points, symbols from text. We need only interested in a set of meaningful words.
- splitting into shingles – process of diving text into set of subsequences. Amount of set can be calculated by number of words minus size of shingle plus one. So, if we have 1000 words after canonization and size of shingle is 5, then set of shingles length will be 1000 – 5 + 1, 996.
- computing hashes for shingles – process of computing checksum for each shingle. Here we can use different types of hashing algorithms. Because of hash collision, we need to select hashing algorithm accurately. In shingle method hashing is used to minimize size of data and comparison.
- searching for identical subsequences – process of comparing hashes. In this step we can use metrics like Jaccard coefficient and etc.

Let's say we have two texts: text1 is "One of the urgent tasks of the modern information society is to find documents that are completely and partially similar to each other" and text2 is "Search for documents that are completely or partially similar to each other is one of the urgent tasks of the modern information society".

First step of shingles method is text canonization. So we need to lowercase, remove not letters, remove extra spaces, remove punctuation, remove symbols which length is smaller or equal to two and remove words like 'i', 'me', 'do', 'the', 'a', 'an', 'because' and etc. After text canonization step text1 becomes "one the urgent tasks the modern information society find documents that are completely and partially similar each other" and text2 becomes "search for documents that are completely partially similar each other one the urgent tasks the modern information society".

Second step is splitting our texts into shingles. Let's select three as shingle size. Result of splitting into shingles for text1 is equals to set ('one the urgent', 'the urgent tasks', 'urgent tasks the', 'tasks the modern', 'the modern information', 'modern information society', 'information society find', 'society find documents', 'find documents that', 'documents that are', 'that are completely', 'are completely and', 'completely and partially', 'and partially similar', 'partially similar each', 'similar each other'). Result of splitting into shingles for text2 is equals to set ('search for

documents', 'for documents that', 'documents that are', 'that are completely', 'are completely partially', 'completely partially similar', 'partially similar each', 'similar each other', 'each other one', 'other one the', 'one the urgent', 'the urgent tasks', 'urgent tasks the', 'tasks the modern', 'the modern information', 'modern information society').

Third step is computing hash values for shingles. As hash function used MurmurHash version 3 with seed value 5. Hash values for text1 is set (2092786056L, -117797895L, 1091229950L, -1719675260L, 1475834944L, -1933512722L, -919041499L, -1169735792L, -1963868842L, 1454935347L, 38303714L, 443125013L, 1634745116L, -1648892685L, -1401603515L, 774181926L) and for text2 is set (650162799L, 255259750L, 1454935347L, 38303714L, -1612114097L, -932889383L, -1401603515L, 774181926L, 511101553L, 907266865L, 2092786056L, -117797895L, 1091229950L, -1719675260L, 1475834944L, -1933512722L)

Last step is search for identical subsequences. To calculate similarity we can use different metrics, but on this example we will use Jaccard coefficient which calculated by formulae set A intersection set B divided by set A union set B. In our case intersection of hash values for text1 and text2 sets is equals to 20, and union of hash values for text1 and text2 is equals to 32. Jaccard similarity will be 20 / 32 * 100. So result of similarity between text1 and text2 is 62.5%.

To speed up shingles method, we can use MinHash algorithm, super-shingles, mega-shingles. Shingles method can be used on different areas like search engines [21], on clustering and etc.

Fuzzy String Matching

Fuzzy String Matching is the process of finding strings that matches to given pattern. The similarity of matching to pattern is measured in terms of distances to edit. Distances to edit is the number of operations needed to convert string into exact match. Operations are usually insertion, substitution and deletion. Insertion is used to insert new character to given position. Deletion is used to delete some character. Substitution is process of replacing some character with new character. Fuzzy String

Matching used to solve different tasks. For example: text plagiarism detection, spell-checking, spam filtering and etc.

Turnitin

Turnitin is online, commercial plagiarism detection service. Turnitin was launched in 1997. It is used as SaaS by schools and universities. Turnitin checks all submitted files by its database and returns plagiarism score as percentage. Turnitin database contains public internet resources, newspapers, books, student papers, and journals.

Copyleaks

It is online service which is used to find out if textual content is original and where it has been used before. Using copyleaks we can detect plagiarism in online content, webpages, text, images of text (images with text information). To use this service we can connect to copyleaks using copyleaks api using copyleaks account. We can use this service for education (for students, universities, schools), for businesses (for publishers, for SEO) [22].

Rabin-Karp Algorithm

The Rabin-Karp algorithm is a string search algorithm that looks for a pattern using hashing. Pattern is substring in the text. It was developed in 1987 by Michael Rabin and Richard Carp. One of the simplest practical applications of the Rabin-Karp algorithm is to detect plagiarism.

Say there is a word of length m and a document of length n. Both the word and the document are represented as strings. The naive way to search the document for the word would be to start at the beginning of the document and check every m sequential letters to see if it matches up with the letters in the word. If the m letter slice of the document matches with the m letters in the word, then a match has been found [23].

This kind of string checking is very slow. Because in this kind of checking algorithm complexity is O(nm). O(nm) means that algorithm needs n*m operations to complete. If document length n is very large, then this algorithms' complexity is O(nn), meaning it needs n*n operations to complete.

To speed-up this kind of algorithm is to reduce the number of unnecessary comparisons. For example, if substring we are checking is "BAG" and segment of text that we comparing it is "THB", then we can see that after first operation of comparison between first characters "B" and "T" that this will not be match. Because of this we can start to compare other segments of text without wasting time to comparing remaining two letters since we know that they won't match.

Rabin-Karp algorithm uses rolling hash technique and hash functions. Rolling hash used to calculate a hash value of string or substring without rehash the entire string. Rolling hash technique used to speed-up Rabin-Karp algorithm.

For example, let's say that we have hash function H which return unique prime number to each character of alphabet. And we are searching from string "loremipsum" of length 10 for substring "mip" of length 3. Here to search for substring "mip" we can use rolling hash technique. At each step we need compare 3 characters of string "loremipsum" to substring "mip". On first step we get hashes for first 3 characters of string "loremipsum" like H(l), H(o), H(r) and compute them to get hash value of substring "lor". To get hash value of substring "mip" we can make same operation. And then we need to compare hash values, it will be not match. On second step we need to calculate hash value of, next 3 characters of string "loremipsum", substring "ore". To calculate hash value substring "ore" we can get hash function of H(o), H(r), H(e), but we will repeat work which we already do. So we can calculate hash value of substring "ore" by dividing hash value of substring "lor" by hash function H(l) and multiplying it by H(e). Now we need to compare this hash value with hash value of substring "mip" and so on.

Rabin-Karp algorithm in rolling hash implementation uses only simple operations like addition, multiplication and substraction. There are constant number of operations at each step of algorithm, so these operations complexity is O(1). Complexity of operation of comparing substring searching for with substring of text will be also O(1). So if we have text of length k then complexity of step will be O(k).

If our text length is n and substring searching for length is m, then average complexity of Rabin-Carp algorithm is O(n+m). If there is hash collision on each step

then complexity of Rabin-Carp algorithm will become O(nm). Because at each step algorithm must verify each letters. To avoid hash collisions we need select good hash function.

LCS (Longest Common Substring)

LCS stands for Longest Common Substring. It is one of the ways for computing how two strings are similar. It searches the length of longest commom subsequence between strings. LCS can be used in simple plagiarism detection. LCS used for file comparison, searching similar DNA sequences, and etc.

TF-IDF

Tf-idf stands for term frequency-inverse document frequency. tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query [24]. Also Tf-idf can successfully used in plagiarism detection, filtering stop-words, text classification, text summarization and etc.

On Tf-idf, weight is composed by two terms. The first one is Term frequency (TF). Term frequency is the ratio of the number of occurrences of a word to the total number of words in a document. It shows importance of the word within document. The second one is the Inverse Document Frequency (IDF). Inverse Document Frequency is the frequency inversion, with which some word occurs in the collection documents. IDF reduces the weight of commonly used words. For each unique word within a specific collection of documents, there is only one IDF value. Words with a high frequency within a specific document and with a low frequency of use in other documents will take high Tf-idf weight.

To use tf-idf weight for checking for plagiarism, what we need to is to find tf-idf for two texts and calculate similarity between two vector using cosine similarity [25].

Example of using tf-idf weight for checking for plagiarism. Let's say we have two texts. Text1 is "One of the urgent tasks of the modern information society is to find documents that are completely and partially similar to each other" and text2 is "Search for documents that are completely or partially similar to each other is one of the urgent tasks of the modern information society" like on example for shingle method. Here we need canonize texts, then we should create tf-idf matrix. To calculate similarity we will use cosine similarity. Result of similarity between text1 and text2 using tf-idf weight with cosine similarity is 63.37%. Result of similarity using shingles method with Jaccard coefficient was 62.5%. We can say that two results are almost similar.

Bag of words

The Bag-of-words model is the most popular and simple model of text representation used in many Text Mining tasks. The model represents the text as a set of words without regard to their mutual arrangement and mutual relations. With its use, the semantic proximity of two texts (two sets of words) is estimated by the number of matching words. This means that two texts in which there are few common words or none at all, are considered semantically and thematically not close. Ignoring the semantic links between words is the main drawback of the Bag-of-words model.

The Bag-of-words model commonly used as feature generation tool. For example, classification of documents based on occurrence of each word as feature. This model also used in computer vision and plagiarism detection applications.

In this model text will transformed into bag-of-words model, then calculated various measures to characterize text. Main characteristics, features which we can calculate from the Bag-of-words model is frequency, meaning number of occurs of word in the text. Using this measures we can calculate similarity between two texts. To calculate similarity we can use cosine similarity.

Let's calculate similarity for two texts using Bag-of-words model and cosine similarity. Text1 is "One of the urgent tasks of the modern information society is to find documents that are completely and partially similar to each other" and text2 is "Search for documents that are completely or partially similar to each other is one of the urgent tasks of the modern information society" like on example for shingle method.

First step is to create Bag-of-words model for texts. Bag-of-words model for text1 will be ('the': 2, 'and': 1, 'information': 1, 'tasks': 1, 'society': 1, 'modern': 1, 'completely': 1, 'one': 1, 'urgent': 1, 'that': 1, 'documents': 1, 'are': 1, 'each': 1, 'other': 1, 'similar': 1, 'find': 1, 'partially': 1). Model for text2 will be ('the': 2, 'information': 1, 'search': 1, 'documents': 1, 'for': 1, 'that': 1, 'tasks': 1, 'modern': 1, 'one': 1, 'urgent': 1, 'completely': 1, 'other': 1, 'are': 1, 'each': 1, 'society': 1, 'similar': 1, 'partially': 1).

Second step is to calculate similarity between Bag-of-words models using cosine similarity. To calculate cosine similarity we need to calculate dot product and magnitude of two vectors, after we need to divide dot product by magnitude. To get percentage we need to calculate result of division by 100. For our example, magnitude equals to 20 and dot product equals to 10. So, result of similarity using Bag-of-words model using cosine similarity for text1 and text2 is equals to 90%.

Calculating similarity between texts

After extracting features from texts using various methods we need to calculate similarity between texts. To extract characteristics or features we can use TF-IDF, LCS (Longest Common Substring), Shingle method, Bag-of-words model and etc. To calculate similarity we can use metrics like cosine similarity, euclidean distance and etc. or if our text lengths are small we can use Jaro distance, Jaro-Winkler distance and etc.

Jaro distance

The Jaro distance is used to measure similarity between two strings. If Jaro distance is higher, then two strings is more similar. Jaro distance returns score between

0 and 1. 0 means there is no similarity between strings, 1 means exact match of two string.

Jaro-Winkler distance

The Jaro-Winkler distance is also used to measure similarity between two strings. Jaro-Winkler distance is a modification and improvement of Jaro distance. If Jaro-Winkler distance score is lower, then strings are more similar. But Jaro-Winkler score is normalized as in Jaro distance. It also returns score between 0 and 1 and 0 means there is no similarity between strings, 1 means exact match of two string. The Jaro-Winkler similarity is given by 1 − Jaro-Winkler distance.

Levenshtein distance

Levenshtein distance is the minimum number of operations for inserting one character, removing one character and replacing one character for another, necessary for converting one line to another. It is a measure of the "similarity" of the two strings. For the first time the problem was mentioned in 1965 by the Soviet mathematician Vladimir Iosifovich Levenshtein in the study of sequences 0-1. Subsequently, a more general problem for an arbitrary alphabet was associated with his name.

Levenshtein distance and its generalizations are actively used to correct errors in the word (in search engines, databases, when entering text, when recognizing scanned text or speech), for comparing text files using the diff utility and in bioinformatics for comparing genes, chromosomes and proteins.

Damerau-Levenshtein distance

Damerau-Levenshtein distance is the minimum number of operations to insert, delete, replace one character, and transpose two adjacent characters needed to transform one string to another. It is a modification of the Levenshtein distance, differs from it by adding a permutation operation.

The distance of the Damerau-Levenshtein, like Levenshtein metric, is a measure of the similarity of the two lines. The algorithm of its search finds application in realization of fuzzy search, and also in bioinformatics (DNA comparison), in spite of the fact that initially the algorithm was designed to compare texts typed by a person.

Damerau proved that 80% of human errors in typing texts are permutations of neighboring characters, skipping a character, adding a new character, and an error in the symbol. Therefore, the Damerau-Levenshtein distane is often used in spelling checker programs.

Cosine similarity

Cosine similarity is a measure of similarity between two vectors of the pre-hilbert space, which is used to measure the cosine of the angle between them. If two characteristic vectors, A and B, are given, cosine similarity, cos (θ), can be represented using a dot product and magnitude as in formula (1).

$$similarity = cos(\theta) = \frac{A \cdot B}{||A||\ ||B||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\ \sqrt{\sum_{i=1}^{n} B_i^2}} \quad (1)$$

One of the reasons for the popularity of cosine similarity is that it is effective as an evaluation measure, especially for sparse vectors, since only non-zero measurements need to be taken into account.

1.3 Conclusion for first section

This section of the thesis provides an explanation and a scheme of work for the most popular algorithms and methods for text recognition and searching for duplicate, and for some of them examples are given. There are a lot of methods, algorithms and software applications exists to extract text information from image files and to search for duplicate. Some of them are free to use, and some them are commercial.

Most popular applications, methods to extract text from image is Tesseract, and to search for duplicates are Shingles method, Bag-of-words model and TF-IDF. If documents contains a lot of words we can use Minhash to check for plagiarism.

2 TEXT RECOGNITION

To extract text information from image files we need some sort of page-layout analysis to find out where text is. When we looked at the image, we could immediately isolate the text region. For computer it is difficult to find text regions. On my application I have used OpenCV library to find and crop text region. Then I have used pytesseract (python implementation of Tesseract OCR) module to extract text from image.

OpenCV (Open Source Computer Vision Library) is a library of algorithms for computer vision, image processing and numerical algorithms of general purpose with open source. Implemented in C / C ++. Also, it exists for some other languages, for example, for Java, Python and etc. It includes various algorithms for computer vision, image recognition and much more, working in real time. All interested persons can use the OpenCV library for free, both for educational purposes and in commercial projects.

It includes the algorithms like recognition objects in the video stream, recognition of printed and handwritten text, elimination of picture distortion, identification the similarity and shape of objects, tracking the movement of an object, recognition of movements, gestures and much more. As an example, this library can be used to search for faces on an image or in a video stream from a camera phone or camera.

Nowadays recognition of objects in a multimedia video stream is becoming particularly relevant. There is a lot of research in this area. For example, German scientists developed software that recognized the figures of people, and depending on where the person moved, the program automatically turned the camera and watched for person.

The open license for OpenCV was designed in such a way that it was possible to create a commercial application using any OpenCV features. Since its alpha release in January 1999, OpenCV has been used in many applications and research projects, including: the imposition of conventional maps and photographs from the satellite, alignment of documents during scanning, removal of noise from medical images,

analysis of objects, systems security, automatic surveillance, quality control systems at work, calibration of cameras, as well as unmanned aerial, ground and underwater vehicles. It was even used to recognize sound and music, where image recognition methods were applied to images of sound spectrograms. The OpenCV library is a powerful tool for solving problems in both scientific and industrial fields.

2.1 Edge detection

The first step is to detect edges using OpenCV. We need to resize input image to have height 500 pixels. Resizing image makes edge detection step more accurate and speeds up image processing. Then we need to convert the image from RGB to grayscale. After we should denoise our image using Gaussian blurring and perform Canny edge detection. Result of edge detection is shown on Figure 1.

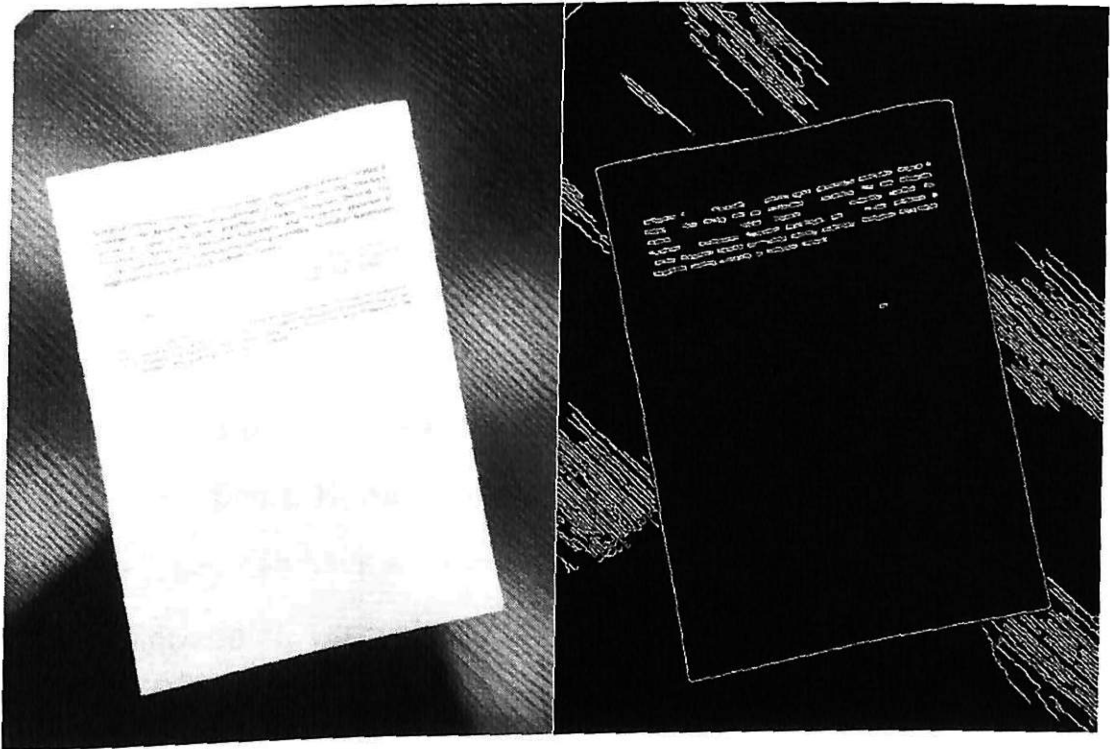


Figure 1. Original image and edges detected image

Canny Edge Detection is a popular edge detection algorithm. It was developed in 1986 by John F. Canny and uses a multi-step algorithm to detect a wide range of boundaries in images. Algorithm is not limited to calculating the gradient of the smoothed image. In the boundary contour, only the points of the maximum of the image gradient are left, and not the maximum points lying near the boundary are

removed. It also uses information about the direction of the border in order to delete the points exactly next to the boundary and not break the boundary itself near the local maximums of the gradient. Then, using the two thresholds, weak boundaries are removed. A border fragment is processed as a whole. If the gradient value somewhere on the trace to be tracked exceeds the upper threshold, then this fragment also remains an "admissible" boundary and in those places where the gradient value falls below this threshold, until it falls below the lower threshold. If, on the whole fragment, there is not a single point with a value greater than the upper threshold, then it is deleted. Such a hysteresis makes it possible to reduce the number of discontinuities in the output boundaries. The inclusion of noise cancellation in the Canny algorithm on the one hand increases the stability of the results, and on the other - increases computational costs and leads to distortion and even loss of details of the boundaries. So, for example, such an algorithm rounds the corners of objects and breaks the boundaries at the connection points.

Before applying the detector, the image is usually converted to grayscale in order to reduce computational costs. This stage is typical for many image processing methods.

The main stages of the algorithm consists:

- Smoothing. Bluring image to remove noise.
- Search for gradients. Borders are marked where the image gradient acquires the maximum value. They can have a different direction, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in a blurred image. The angle of the direction of the gradient vector is rounded off and can take on values such as 0, 45, 90, 135.
- Suppression of non-maximum. Only local maximum are marked as boundaries.
- Double threshold filtering. Potential boundaries are defined by thresholds.
- Tracing the ambiguity area. The resulting boundaries are determined by suppressing all edges that are not related to certain (strong) boundaries.

OpenCV puts all stages in single function called *cv2.Canny*.

2.2 Finding contours

After we need to find contours in edged image using function *cv2.findContours* and approximate number of points. We can assume that we have found text zone from image if approximated contour contains 4 points. Result of finding contours is shown in Figure 2.

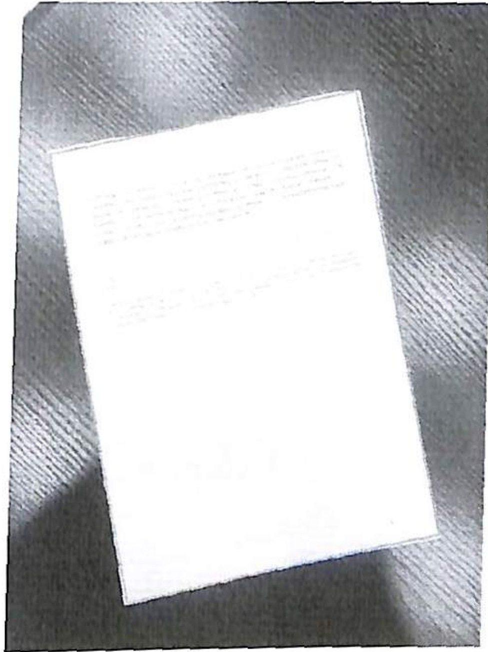


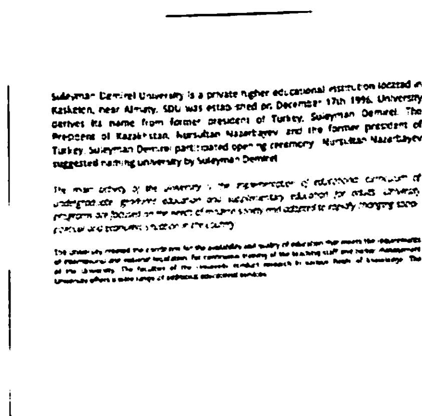
Figure 2. Result of finding contours

Contours are curve which joins all the continuous points with same color or intensity along the boundary. They are useful for object detection and recognition and shape analysis. We should use binary images to get better accurate. Object to be found should be white and background should be black. So before finding contours step we should apply Canny edge detection or threshold. In OpenCV, to find contours we can use function *cv2.findContours*. After finding contours we should to approximate contour to get page from image if page exists.

2.3 Perspective transform

In order to obtain a top-down view, we need to take four coordinates which represents outline of the document and apply a perspective transform. To apply perspective transform, first we should to determine the width of new image. Here width is the largest distance between the bottom-left and bottom-right x-coordinates or the top-left and top-right x-coordinates.

After we should determine height of new image. Here height is the maximum distance between the bottom-right and top-right y-coordinates or the bottom-left and top-left y-coordinates. Now we can perform the transformation using OpenCV functions. To obtain binarized image, meaning black and white image, we should get warped image and convert it to grayscale. Then we should apply adaptive thresholding filter to obtain black and white image. Result is shown Figure 3. Now we can send this image to pytesseract and get text or we can simplify work of pytesseract by cropping only text regions from deskewed image.



Suleyman Demirel University is a private higher educational institution located in Kaskelen, near Almaty. SDU was established on December 17th 1996. University derives its name from former president of Turkey, Suleyman Demirel. The President of Kazakhstan, Nursultan Nazarbayev and the former president of Turkey, Suleyman Demirel participated opening ceremony. Nursultan Nazarbayev suggested naming university by Suleyman Demirel

[illegible]

[illegible]

Figure 3. Result of perspective transformation and adaptive thresholding

In OpenCV there are two functions for transformation: *cv2.warpAffine* and *cv2.warpPerspective*. Using these functions we can make all kind of transformation.

For perspective transformation, we need a 3x3 transformation matrix. Straight lines will remain straight even after the transformation. To find this transformation matrix, we need 4 points on the input image and corresponding points on the output image. Among these 4 points, 3 of them should not be collinear. Then transformation matrix can be found by the function *cv2.getPerspectiveTransform*. Then we should apply *cv2.warpPerspective* with this 3x3 transformation matrix.

In affine transformation, all parallel lines in the original image will still be parallel in the output image. To find the transformation matrix, we need three points

from input image and their corresponding locations in output image. Then function *cv2.getAffineTransform* will create a 2x3 matrix which is to be passed to *cv2.warpAffine*.

Simplest method of image segmentation is called thresholding. It is used to get binary images from grayscale images. OpenCV uses function *cv2.threshold* to threshold image. First argument of function is the source image. Source image should be grayscale image. Second argument of function is threshold value which is used to classify the pixel values. Third argument is the maximum value which represents value to be given if pixel value is more than (sometimes less than) the threshold value. If pixel value is greater than a threshold value, it is assigned one value, else it is assigned another value. OpenCV provides different styles like *cv.THRESH_BINARY*, *cv.THRESH_BINARY_INV*, *cv.THRESH_TOZERO* of thresholding and it is decided by the fourth parameter of the function.

Thresholding using threshold value is not good solution in all conditions where image has different lighting conditions in different areas. In such situation, we should use adaptive thresholding. In adaptive thresholding algorithm calculate the threshold for each small parts of image. So as result we get different threshold values for different parts of the same image.

2.4 Cropping

When we want to crop set of all possible crops can be very large. Instead of a lot of cropping operations we can find chunks of text. To do this, we can apply binary dilation to deskewed image.

Figure 4. Result of cropping

This process concatenates text regions one into another. We do this step repeatly until there are only a few connected components. By including some of these components and rejecting others, we can form candidate crops. To solve this we can use F1 score [26]. After finding accepted set of components, we can crop image as shown in Figure 4.

2.5 OCR (Optical Character Recognition)

Now we can use Tesseract to convert text information from image to text. Pytesseract is a wrapper for 'Google's Tesseract OCR Engine. It is also useful as a stand-alone invocation script to tesseract, as it can read all image types supported by the Python Imaging Library, including jpeg, png, gif, bmp, tiff, and others, whereas tesseract-ocr by default only supports tiff and bmp [27].

To get text we should use function *image_to_string* with input parameters image and language.

2.6 Conclusion for second section

This section of thesis is about OpenCV library, Tesseract OCR. It provides information about how to use OpenCV library for image processing, image processing steps, how to find text zones on image, examples and provides information about Tesseract OCR for text recognition.

3 SEARCH DUPLICATES

Application consists several steps to search duplicates. To calculate similarity between texts application uses shingles method with hash, shingles method with minhash, tf-idf with cosine similarity and bag of words with cosine similarity. Then similarity percentage between two texts calculated as average of them.

3.1 Shingle method

Shingle method for duplicate detection contains several steps like canonization of text, splitting into shingles, computing hashes for shingles and searching for identical shingles.

Canonization of text is process of removing useless signs, words, prepositions, points, symbols from text. We only interested in a set of meaningful words.

Splitting into shingles is process of diving text into set of subsequences. Amount of set can be calculated by number of words minus size of shingle plus one. For example, if we have 500 words after canonization step and our shingle size is 3, then set of shingles length will be $500 - 3 + 1$, 498.

Computing hashes for shingles is process of computing checksum for each shingle. Here we can use different types of hashing algorithms. Because of hash collision, we need to select hashing algorithm accurately. In shingle method hashing is used to minimize size of data and comparison.

Searching for identical subsequences is process of comparing hashes. In this step we can use metrics like Jaccard coefficient and etc.

3.1.1 Text canonization

Checksums are very sensitive to changes. For example, the checksums for the following two texts will be different: checksum for "Hello world" is -1948869038, for the sentence "Hello world!" is 461707669. The difference between the first and second text is only in the character at the end of the text, but as we can see, the checksums are completely different.

When we searching for duplicates, we are not interested in any exclamation marks, points, commas and etc. We are only interested in words (not conjuctions,

prepositions, etc.) Therefore, we need to clear the text from useless signs and words that do not make sense when comparing, this process is called "canonization of text".

First of all we need to create a set of symbols and words that need to be excluded from both texts. Then we need to clear texts from unnecessary symbols and words. Example for stop-words: like i, me, my, myself and etc. Example for stopsymbols: .,!?:;-(). We can also add stemming of words.

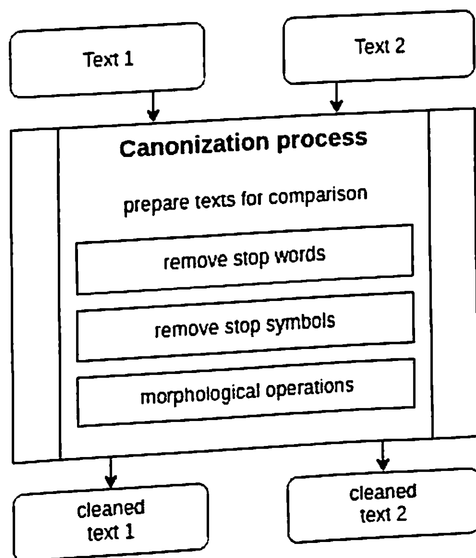


Figure 5. Text canonization step

On application to the stop symbols I have included punctuation marks, newlines and tabs. To the stop words I included conjuctions, prepositions and other words, which, when compared, should not influence the result.

Converting texts to a single canonical form is not limited. For example, we can use morphological analyzers of the languages and etc.

3.1.2 Splitting into shingles.

After canonization step we need to break texts into subsequences – shingles. The division into shingles can be carried out in different ways. A shingle can contain one letter or one word, or a sentence or several sentences.

In shingle method selecting word, character size for shingle can vary greatly in effectiveness. If we choose big size for shingle then it can lead us to incorrect comparison results. If the shingles size too small, then the accuracy of comparison can be high, but it needs more processing powers. Because of shingle method is used to

reduce the complexity of comparing two texts, data, documents, we need to select shingles size more accurately.

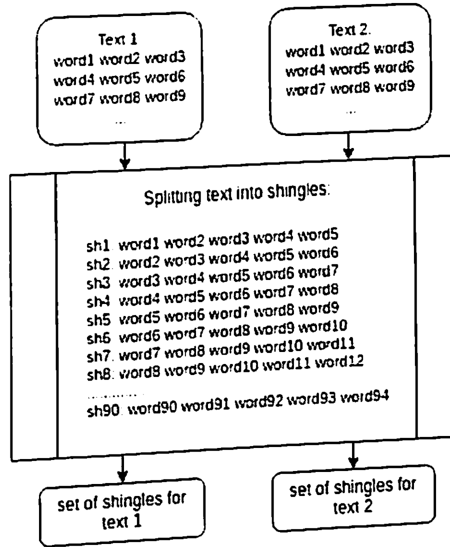


Figure 6. Splitting into shingles step

Amount of shingles can be calculated by number of words minus size of shingle plus one. If we have 2500 words after canonization and size of shingle is 10, then set of shingles length will be $2500 - 10 + 1 = 2491$.

3.1.3 Calculating hashes.

We can use substring as shingles, but we need more space to store them and more computer resources to compare them. Because of this we need to use hashing function to get hash value of set of strings instead of using set of strings directly as shingles.

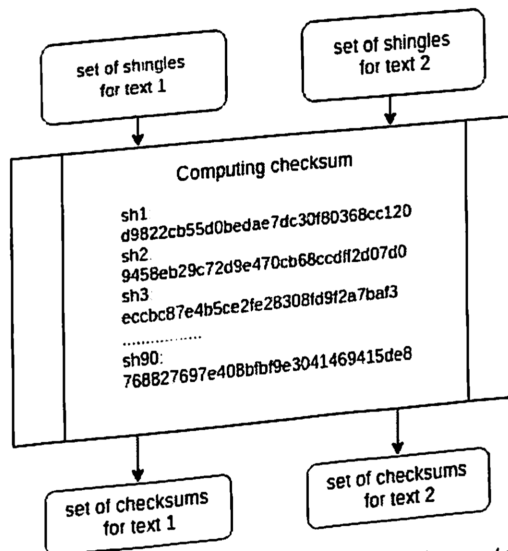


Figure 7. Calculation of hash values step

For instance, if our shingle size is equals to five words and each word contains 6 characters, then shingle size in bytes will be 6 * 5 + 4 = 34 bytes. Here 6 * 5 is number of characters, and each character equals to one byte. 4 is spaces between words. If we will use hashing function which gets string as input and returns number bigger than integer, for example long, then hash value size in bytes will be 64 bits, which means 8 byte. Thus, each shingle will be represented by 8 bytes instead of 30 bytes (approximately). But when we use hash functions hash collision may occur. Because of hash collision, we need to select hashing algorithm accurately. On application to find hashes I have used MurMurHash version 3.

3.1.4 Comparison, determination of the result

Last step is comparing hashes. Comparing can be different. On my application I have used Jaccard coefficient, which gives a similarity score between two sets. The coefficient is defined as the size of the intersection, the number of shingles present in both, divided by the size of the union, number of shingles present in either [28].

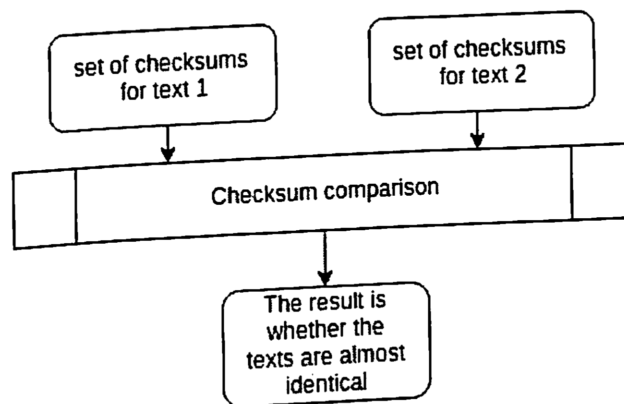


Figure 8. Checksum comparison

If we had this similarity score for a pair of documents we could define a threshhold, and say that any pairs of documents that are over a given score are near-duplicates [29].

Example

Let's suppose that we have two texts to compare. First one is 'Because Almas and Zhalgas arrived at the bus station before noon, I did not see them at the station.'

and second text is 'I did not see them at the station because Almas and Zhalgas arrived at the bus station before noon.'. First step is shown on Figure 9.

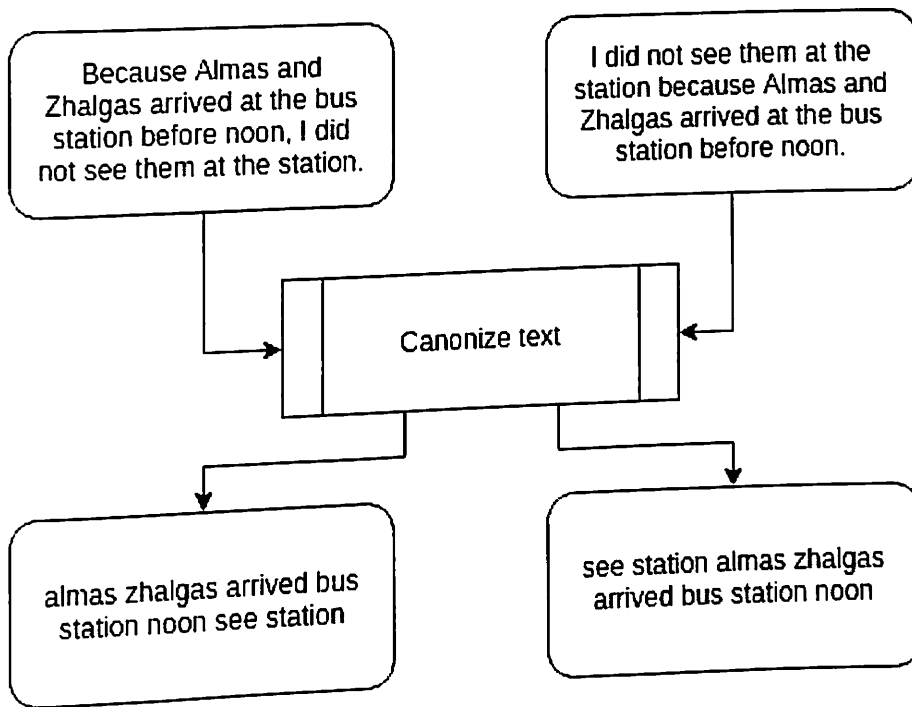


Figure 9. Text canonization step

After clearing text from stop words and stop symbols our text1 becomes 'almas zhalgas arrived bus station noon see station' and text 2 becomes 'see station almas zhalgas arrived bus station noon'.

Second step is breaking cleaned texts to shingles. In my example I have divided text by 3 words. So, we will have ('count of words in cleaned text' - 'shingle length' + 1 = 8 – 3 + 1) 6 shingles in each text. Second step is shown in Figure 10.

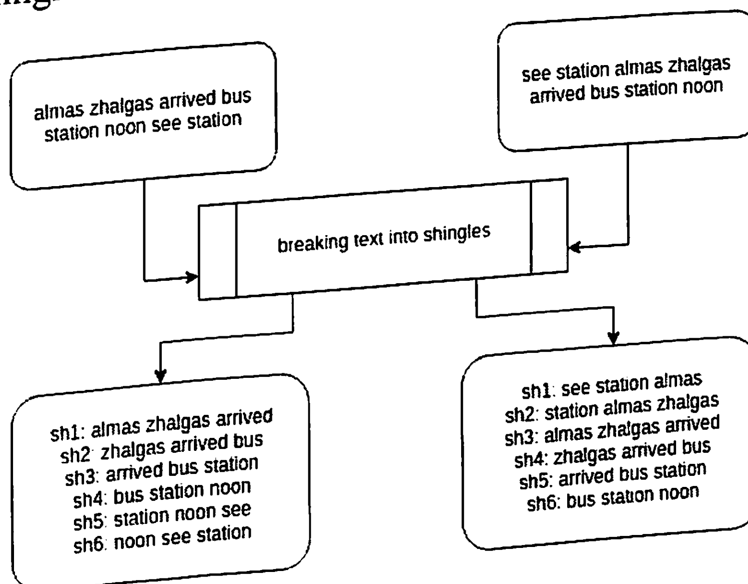


Figure 10. Splitting into shingles step

After breaking we need to compute checksums for each shingles. My example is simple implementation of shingles algorithm. Because of this I have used CRC32 checksum. This is shown on Figure 11.

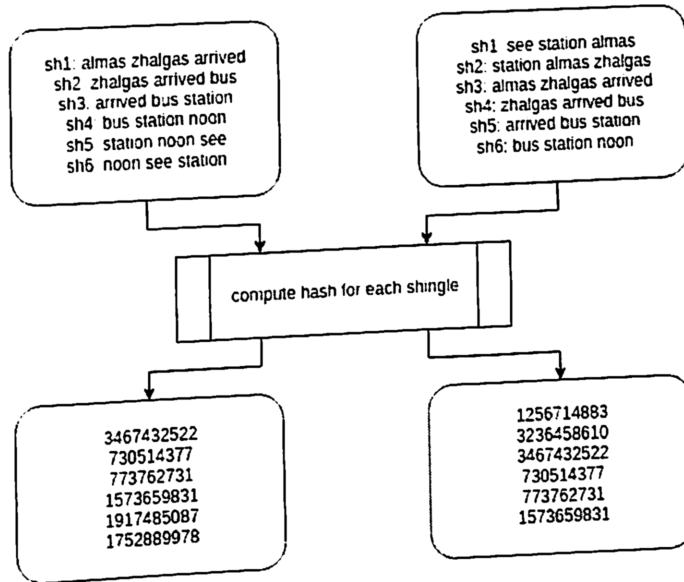


Figure 11. Calculation of hash values step

Last step is comparing our checksum sets and computing percentage of similarity using Jaccard coefficient. Last step is shown Figure 12. In our case similar checksum(intersection of checksum of text1 and checksum of text2) count will be 4.

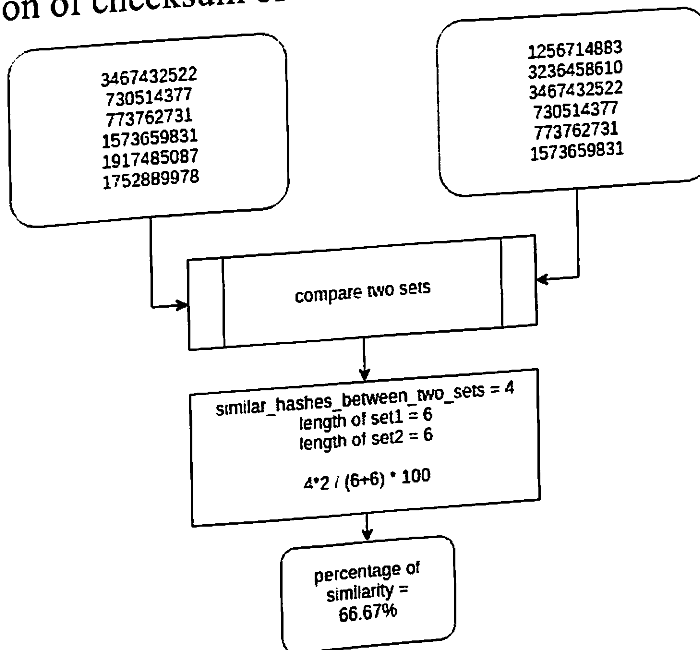


Figure 12. Checksum comparison and calculation similarity

We need to compute it by 2. Because we have 2 sets. After we need to divide intersection coefficient by union of checksum sets. To find percentage we need to

multiply division result by 100. So, percentage of similarity will be 8/12*100 = 66.67%.

3.2 Minhash

The MinHash algorithm is actually pretty easy to describe if you start with the implementation rather than the intuitive explanation. This algorithm can be used to add performance to shingles method. It also has canonization of text and splitting text into shingles step. But, on third step we should create some number hash functions which we will use to get hash value of shingles and select minimum hash value between them. Then we will use these minimum hash values to compare with other texts' minimum hash values.

The key ingredient to the algorithm is that it we will have hash function with no collision. To avoid hash collision this hash function will take as input some integer and will map it to different integer. Put another way, if you took the numbers 0 – (2^32 – 1) and applied this hash function to all of them, we would get back a list of the same numbers in random order. Here is the definition of the hash function, which takes an input integer "x": H (x) = (qx + w) % e. Coefficients q and w are randomly chosen integers which are less than maximum value of x. e is prime number which is bigger than maximum value of x. For different choices of q and w, this hash function will produce a different random mapping of the values. So we have the ability to generate as many of these random hash functions as we want by just picking different values of q and w.

So here's how we compute the MinHash signature for some document. Let's say we have generated 20 hash functions. Now we should to take first hash function and apply this hash function to all shingles of document. Then we should select minimum hash value between them. After we will use selected minimum hash value as the first component of MinHash signature. After selecting first component of MinHash signuature we should take second hash function, again compute hash values for shingles and select minimum hash value, and use it as the second component of Minhash signature. And so on.

So if we have 20 random hash functions, we'll get a MinHash signature with 20 values. We'll use the same 20 hash functions for every document in the dataset and generate their signatures as well. Then we can compare the documents by counting the number of signature components in which they match.

3.3 TF-IDF with cosine similarity

To use tf-idf to check for duplicates we should first create tf-idf matrix. Let's say we have two texts. Text1 is equals to 'this is a first function' and text2 is equals to 'this is a second function'.

First of all we should calculate term frequency (TF) for texts. Term frequency is the ratio of the number of occurrences of a word to the total number of words in a document. It shows importance of the word within document. Term frequency is calculated by formula *TF(a) = number of times word a was encountered in the text / number of words in the text*. TF for text1 will be *("this": 1/5, "is": 1/5, "a": 1/5, "first": 1/5, "function": 1/5)* and TF for text2 will be *("this": 1/5, "is": 1/5, "a": 1/5, "second": 1/5, "function": 1/5)*.

Second step is to calculate Inverse Document Frequency (IDF). Inverse Document Frequency is the frequency inversion, with which some word occurs in the collection documents. IDF reduces the weight of commonly used words. For each unique word within a specific collection of documents, there is only one IDF value. Words with a high frequency within a specific document and with a low frequency of use in other documents will take high Tf-idf weight. IDF is calculated by formula *IDF(a) = 1+ logarithm(total number of texts / number of text in which exists word a)*. So IDF for corpus will be *("this": 1+log10(2/2) => 1, "is": 1+log10(2/2) => 1, "a": 1+log10(2/2) => 1, "function": 1+log10(2/2) => 1, "first": 1+log10(2/1) => 1.3, "second": 1+log10(2/1) => 1.3)*. As logarithm we can use any kind of logarithm. Because TF-IDF is a relative measure. The weight of terms are not expressed in any units, but exist relative to each other.

Third step is to calculate TF-IDF. TF-IDF is calculated by formula *TFIDF(element) = TF(element) * IDF(element)*. Result of TF-IDF for text1 will be

("this": 0.2, "is": 0.2, "a": 0.2, "first": 0.26, "function": 0.2) and TF-IDF for text2 will be *("this": 0.2, "is": 0.2, "a": 0.2, "second": 0.26, "function": 0.2).*

Last step is calculating similarity between TF-IDF values using cosine similarity. Here dot product will be 0.16 and magnitude will be 0.2276. So result will be 0.702987. To get similarity percentage we should compute this value by 100. Result of similarity between texts using tf-idf with cosine similarity is 70.3%.

3.4 Bag-of-words model with cosine similarity

To use Bag-of-words model we should create model for texts. Bag-of-words model simply represents the text as a set of words without regard to their mutual arrangement and mutual relations.

Let's suppose that we have two texts to compare. Text1 is equals to 'Because Almas and Zhalgas arrived at the bus station before noon, I did not see them at the station' and text2 equals to 'I did not see them at the station because Almas and Zhalgas arrived at the bus station before noon'.

First step is to canonize texts. Let's just remove punctuations and lowercase all words in texts. After canonization step our text1 becomes 'because almas and zhalgas arrived at the bus station before noon i did not see them at the station' and text2 becomes 'i did not see them at the station because almas and zhalgas arrived at the bus station before noon'.

Second step is to create Bag-of-words model for texts. Bag-of-words model for text1 will be *('station': 2, 'the': 2, 'at': 2, 'and': 1, 'them': 1, 'because': 1, 'i': 1, 'bus': 1, 'noon': 1, 'zhalgas': 1, 'did': 1, 'see': 1, 'arrived': 1, 'almas': 1, 'not': 1, 'before': 1).* Model for text2 will be *('station': 2, 'at': 2, 'the': 2, 'and': 1, 'them': 1, 'because': 1, 'i': 1, 'bus': 1, 'noon': 1, 'zhalgas': 1, 'did': 1, 'see': 1, 'arrived': 1, 'not': 1, 'before': 1, 'almas': 1).*

Lasr step is to calculate similarity between Bag-of-words models using cosine similarity. To calculate cosine similarity we need to calculate dot product and magnitude of two vectors, after we need to divide dot product by magnitude. To get percentage we need to calculate result of division by 100.

For that example, magnitude equals to 25 and dot product equals to 25. So, result of similarity using Bag-of-words model using cosine similarity for text1 and text2 is equals to 100%.

3.5 Calculating final similarity

After calculating similarities using Shingle method, MinHash, TF-IDF with cosine similarity and Bag-of-words model with cosine similarity we will have four similarities. But we can have such a situation that the results of these methods and algorithms will be very different from each other. For example, let's say we have text1 and text2 to compare, and result of similarity using Shingle method is 50%, result of similarity using MinHash is 5%, result of similarity using TF-IDF with cosine similarity is 45%, and result of similarity using Bag-of-words model with cosine similarity is 70%. How we should calculate final similarity result? We can just calculate average of similarities. But it is not good solution. Because we do not know which similarity results are reliable, and which of them are not. In such kind of situation, we can use statistics interval estimate called confidence interval.

Confidence interval is a range of values so defined that there is a specified probability that the value of a parameter lies within it. To calculate confidence interval we should calculate mean and standard error of array of similarities. Mean is calculated by formula (2), which means division sum of similarities by number of similarities, where N is number of similarities and numerator is sum of similarities.

$$\mu = \frac{\sum_{i=1}^{n} x_i}{N} \quad (2)$$

After calculating mean, we should calculate standard error. But to calculate standard error we should calculate first variance by formula (3) where μ is mean, N is number of similarities and x_i is i^{th} similarity.

$$\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{N} \quad (3)$$

Then we should calculate standard deviation by just taking square root of formula (3), or by formula (4) where μ is mean, N is number of similarities and x_i is i[th] similarity.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \mu)^2}{N}} \quad (4)$$

Now we can calculate standard error using formula (5), where σ is standard deviation and N is number of similarities.

$$SE = \frac{\sigma}{\sqrt{N}} \quad (5)$$

After calculating standard error and mean, we can easily calculate confidence interval of similarities using formula (6), where Z is z-value (for 95% interval - 1.960, for 99% interval - 2.576, for 90% interval - 1.645), SE is standard error and μ is mean.

$$CI = [\mu - Z * SE, \mu + Z * SE] \quad (6)$$

Let's calculate confidence interval for situation when we have text1 and text2 to compare, and result of similarity using Shingle method is 50%, result of similarity using MinHash is 5%, result of similarity using TF-IDF with cosine similarity is 45%, and result of similarity using Bag-of-words model with cosine similarity is 70%.

First step is to calculate mean. For our example mean will be 42.5 (170 / 4). Second step is to calculate variance. Variance will be equals to 556.25. Third step is to calculate standard deviation. Standard deviation is square root of variance. So standard deviation will be approximately equals to 23.59. Fourth step is to calculate standard error. Standard error is calculated by standard deviation divided by square root of number of similarities. So standard error will be approximately equals to 11.79 (23.59 / 2). Fifth step is to select interval (z-value). If we choose z-value for 90%, and then choose another z-value, then confidence intervals will be different.
Last step is to calculate confidence interval.

For confidence level 95%. In that case, our z-value will be equals to 1.96 and confidence interval will be approximately between 19.39 and 65.61. Only similarities Shingle method (50%) and TF-IDF with cosine similarity (45%) are lies on that interval. So final similarity result will be 47.5% (95 / 2).

For confidence level 99%. In that case, our z-value will be equals to 2.576 and confidence interval will be approximately between 12.12 and 72.88. Similarities Shingle method (50%), TF-IDF with cosine similarity (45%) and Bag-of-words model with cosine similarity (70%) are lies on that interval. So final similarity result will be 55% (165 / 3).

3.6 Conclusion for third section

Third section of thesis is about methods and algorithms, which I have used in my application to search similarities between texts. On this section given information about methods and algorithms Shingle method, Minhash, TF-IDF with cosine similarity, Bag-of-words model with cosine similarity, and shown steps and simple examples about how to use these methods and algorithms in practice. Also given information and example about calculation of final similarity result using similarity results of Shingle method, TF-IDF with cosine similarity, MinHash and Bag-of-words model with cosine similarity.

4 PRACTICAL PART

4.1 Selected tools

Master's thesis application consists of two parts: backend and frontend. Backend and frontend communicates using REST API. All application data are stored in database. If user needs data from database, for example, list of documents, it sends request to backend, then backend sends query to database, finally backend sends to frontend requested data as json format.

4.1.1 Database

As database of application I have used PostgreSQL database. Because PostgreSQL is a powerful, open-source, one of the popular database in the world.

In application database there are 16 tables. Two of them are used to store main application data: dblist and stopwords.

Dblist table used to store information about document. This is main table of application. All documents are stored in this table. It contains 16 columns:

- id – primary key, integer; unique id number.
- user – foreing key, integer; information about who added document to database.
- createdon – datetime; creation date and time of document on database.
- name – varchar; name of document.
- language – varchar; language of document, can be either russian, english or kazakh.
- media_url – varchar; path to file of document.
- block_size – integer; shingle size.
- type – varchar; type of document.
- organization – varchar; information about organization.
- author_fullname – varchar; name and surname of author.
- advisor_fullname – varchar; name and surname of advisor. This field used when document type is thesis.
- year – integer; creating or released year of document.

- month – varchar; creating or released month of document.
- hashes_txt – text; hash array of document text.
- minhashes_txt – text; minhash array of document text.
- content_txt – text; canonized text of document.

Stopwords table used to store stop-words which are used on text canonization step. It contains 2 columns:

- word – varchar; lowercased text.
- language – varchar; language of word, can be either russian, english or kazakh.

4.1.2 Backend

As backend of application I have used Django. Django is free, open-source, a high-level Python Web framework. To create REST API I have used flexible toolkit called django-rest-framework. For authorization to REST API I have used JSON web tokens. To add documents or to check for plagiarism users needs to authorize. Without authorization application sends them unauthorized error code (401).

Backend contains several urls. Main of them are:

- POST /auth/token/ - generates jwt if user password and username is correct.
- POST /auth/refresh-token/ - generates new jwt if old jwt is valid.
- POST /dblist/ - save sent document to database.
- GET /dblist/ - get all list of documents from database.
- POST /dblist/search/ - used to check for plagiarism.

To check plagiarism user only needs to send post request to endpoint /dblist/search/ with fields file, language, type of document and minimum similarity percentage. It returns JSON response with result of checking.

On backend there are two additional classes which are used to check plagiarism: Plagiarism, ImageToText. And, one additional class to calculate final results of similarities: ConfidenceInterval.

Class *ImageToText* used to extract text information from image file. It uses OpenCV library and Tesseract. There are 12 methods:

__init__(url, language)

Constructor of class ImageToText. It takes two parameters as input: url and language. Default value for url is None, for language is "en".

get_text()

This method is one of the main methods of class. It takes url and checks for empty and checks for existence of file. If everything is correct, it calls other methods like read_and_select, crop, etc. and return text from text.

crop(img)

Crop method takes image as input, search for text region and crops. After cropping it stores cropped image to temporary directory and returns path to cropped image.

find_number_of_text_pixels_in_crop(img, crop)

Method which returns number of text pixels on given crop x and y parameters.

components_properties(components, edges)

Used to get component information

find_optimal_components_subset(components, edges)

Used to find optimal components from component information.

union_crops(crop1,crop2)

Method which returns union of coordinates crop1 and crop2

find_components(edges, max_component)

Used to search for connected components

read_and_select(path)

Method which returns text region. It reads image from path, converts image to grayscale image, adds median blur to remove noises, find edges and contours, then selects text region and returns as output of method.

four_point_transform(img, pts)

Returns transformed image.

order_points(pts)

Used to find top-left, bottom-left, top-right, and bottom-right from list of coordinates.

resize(img, width, height)

This method used to resize input image to given height and width.

Class *Plagiarism* used to search similarities between texts. Main logic of application is written on this class. It has 15 methods:

__init__(language)

Constructor of class Plagiarism. It takes one parameters as input – language. Default value for language is “en”.

read(url)

Reads given file and returns canonized text. If given file format is not doc, docx, pdf, jpg, png or jpeg, then it returns empty string.

get_stopwords()

Used to get all list of stop-words from database. Only admin can add stop-words to database.

read_doc(url)

Reads content of files with format doc, docx and pdf. Then converts content to lower, remove urls, remove symbols, remove stopsymbols, remove stop-words, remove words which length is smaller than 2, replace words to stems and returns result as output.

read_img(url)

Extract text from image file using class ImageToText. Then converts text content to lower, remove urls, remove symbols, remove stopsymbols, remove stop-words, remove words which length is smaller than 2, replace words to stems and returns result as output.

generate_random_seeds(minhash_size, seed)

Return generated random seeds with given size. Generated random seed is used to compute minhash for shingles.

compute_minhash(shingles)

Takes array of shingles as input and returns array of minhashes. MinHash array is used to calculate similary between texts.

compute_hash(shingles)

Takes array of shingles as input and returns array of hashes. Hashes are computed using MurMurHash version 3 library. Hash array is used to calculate similarity between texts.

get_word_count(text)

Divides text to dictionary of word-count pair. For example, if we pass text 'hello world hello', it returns us ('hello': 2, 'world': 1).

shingles(text, shingles_size)

Divide canonized text into shingles with given size. Default shingle size is 5 word. Returns array of shingles.

tfidf_matrix(text1, text2)

Generates term frequency-inverse document frequency matrix.

calculate_cosine_similarity_for_dicts(dict1, dict2)

Calculates cosine similarity for input dictionaries of word-count pair. Cosine similarity calculates by dot product of dictionary 1 and dictionary 2 divided by square root of sum of squares of dictionary 1 added by square root of sum of squares of dictionary 2.

calculate_hash_similarity(hashes1, hashes2)

Calculates similarity of two array of hashes using Jaccard index. Jaccard index is calculated by formulae – intersection of hashes 1 and hashes 2 divided by union of hashes 1 and hashes 2.

calculate_minhash_similarity(hashes1, hashes2)

Calculates similarity of two array of minhashes by formulae count of similar objects between minhash 1 and minhash 2 divided by length of minhash 1.

calculate_cosine_similarity(tfidf_matrix)

Calculates cosine similarity for given tf-idf matrix.

Class *ConfidenceInterval* is used to calculate confidence interval of similarities. There are 6 methods:

__init__(sample)

Constructor of class ConfidenceInterval. It takes one parameters as input: array of similarities. Default value for input parameter is empty array.

calculate_mean()

Calculates average of similarities. If length of array of similarities is equals to zero, then it returns zero.

calculate_variance()

Calculates and returns variance. Uses *calculate_mean()* method to calculate average. If length of array of similarities is equals to zero, then it returns zero.

calculate_standard_deviation()

Returns standard deviation for input array by calculating square root of variance. Uses *calculate_variance()* method to calculate variance.

calculate_standard_error()

Calculates standard error for sample by dividing standard deviation by sample length. Uses *calculate_standard_deviation()* method to calculate standard deviation.

get_interval()

Returns the start and finish interval, in which the true value of the detected value is located with a probability of 99%. Uses *calculate_standard_error()* method to calculate standard error and *calculate_mean()* to calculate mean for given sample.

4.1.3 Frontend

As frontend of application I have used VueJS. This is a progressive JavaScript framework with open source code, designed to develop a user interface. It is one of the most popular frameworks for simplifying web development. VueJS works mostly with the presentation level. It can easily be integrated into large projects for front-end development. I have choose this framework because it is reactive, fast and flexible.

On frontend there are eight components:

– App.vue – it is main component of application. All other components will be loaded to this main component.

– Header.vue – this component contains menu, logout button, logo and etc. It checks if user logged in to application.

– Footer.vue – component which shows information like copyright and etc.

– PageDashboard.vue – it is main page of application.

– Page404.vue – used to 404 not found error to users.

– PageLogin.vue – login page of application. It contains login form to log in to application. After entering credentials it send post request to backend to get token. If user credentials is valid, user will be redirected to main page of application.

– PageDatabase.vue – page which shows list of document in database. When user enters to this page, it send post request to backend to get list of documents on database. If backend responses with success message, it parses it and renders list of documents.

– PageAddToDatabase.vue – contains form which is used to add new document to database. After filling all required fields, it sends post request to database. If everything is correct, then user get notification and will be redirected to PageDatabase.

– PageSearchFromDatabase.vue – this page is used to check for plagiarism. After filling required fields and clicking search button, it sends post request to backend to find similar documents. After calculating for similarity, backend returns response to frontend. Then frontend parses JSON, and generates result of checking.

Frontend VueJS application uses vue-router for application routing. There are six routes in application:

– /dashboard – route to main page of application (PageDashboard.vue).

– * – route to 404 error page (Page404.vue).

– / – route login page of application (PageLogin.vue).

– /database – route to list of documents on database (PageDatabase.vue).

– /database/add – route to add new document to database (PageAddToDatabase.vue)

– /database/search – route to page to check for plagiarism (PageSearchFromDatabase.vue).

Only authorized users can see components PageDashboard.vue, PageDatabase.vue, PageAddToDatabase.vue, PageSearchFromDatabase.vue, Header.vue and Footer.vue. Components PageLogin.vue, Page404.vue are visible to all users.

4.1.4 Other tools

On application there are several other tools has been used like Bootstrap, Popper.js, Waves.js and etc.

Bootstrap

Bootstrap is a sleek, intuitive, and powerful front-end framework for faster and easier web development. It is a combination of HTML, CSS, and Javascript code designed to help build user interface components. Bootstrap was also programmed to support both HTML5 and CSS3. Bootstrap is a free collection of tools for creating a websites and web applications [30].

Reasons to choose Bootstrap:

- easy to write html code
- compartable grid system
- base styling for most HTML elements such as buttons, forms, tables, images, icons, code, typography and etc.
- wide list of components
- javascript plugins

There are a lot advantages of using Bootstrap. Bootstrap is compatible with all modern browsers like Chrome, Safaru, Firefox and etc. It supports responsive design. Website with bootstrap will have the ability to be seen on any device, that being a mobile phone, tablet, or desktop. Developers won't ever have to worry over their design not being compatible on multiple platforms.

On application bootstap used to construct grid system and style. Using bootstap it is easy to write code for all platforms like mobile, desktop and tablet.

Waves.js

Waves.js is a standalone JS library for creating Google Material Design styled click effects on any html elements. The html elements can be icons, divs, images, buttons or any other inline elements. This library used on application to give styled click effects.

Font Awesome

Font Awesome gives us scalable vector icons that can instantly be customized - size, color, drop shadow, and anything that can be done with the power of CSS. This library used to add some beautiful icons to application.

Material Design Icons

Material design icons is the official icon set from Google. Some of them has been used on application, because this icons are readable at both large and small sizes.

Themify

Themify is a free set of icons for use in web design. This library also used to add some beautiful icons to application.

4.2 UML diagrams

UML is a graphic description language for object modeling in the field of software development, business process modeling, system design and mapping of organizational structures. UML stands for Unified Modeling Language.

UML is an open standard that uses graphic symbols to create an abstract model of a system, called a UML model. UML was created to identify, visualize, design, and document, mainly, software systems. UML is not a programming language, but code generation is possible based on UML models.

The UML uses diagrams like class diagram, component diagram, composite structure diagram, deployment diagram, activity diagram, use case diagram and etc.

Use Case Diagram

Use case diagram in UML is a diagram that reflects the relationship between actors and use cases and is an integral part of the use case model that allows to describe the system at a conceptual level.

The main purpose of the diagram is a describe the functionality and behavior that allows the customer, the end user and the developer to jointly discuss the projected or existing system.

Work on the diagram can begin with a text description. In this case, non-functional requirements such as operation system, programming language will be omitted when constructing a use case model.

List of actors of application: administrator and teacher. Use case diagrams of actors are shown on Figure 13 and Figure 14.

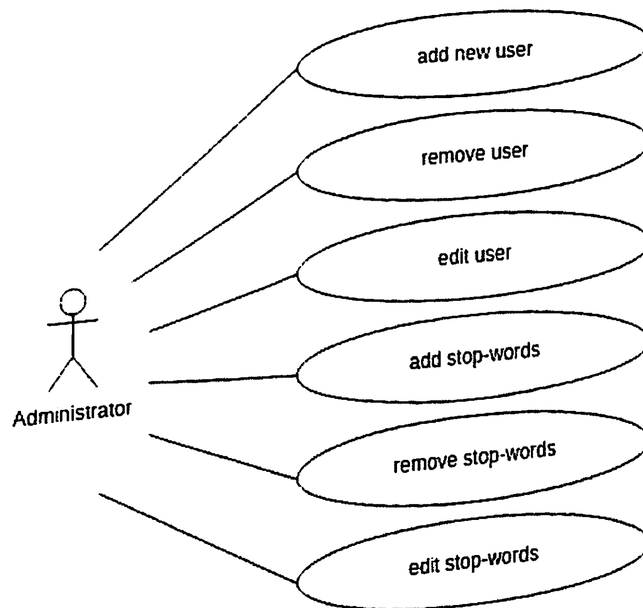


Figure 13. Use Case diagram of Administrator.

Use Cases of actors:

Administrator:

- can add new users to system
- can remove users from system
- can edit users
- can add new stop-words to system

- can remove stop-words from system
- can edit stop-words
- can see list of documents uploaded by users

Teacher:

- can add new document to database
- can check for plagiarism
- can see other teachers uploaded documents

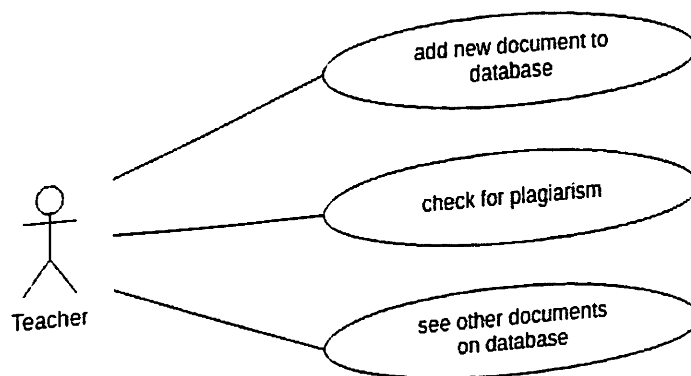


Figure 14. Use Case diagram of Teacher.

Sequence diagram

The sequence diagram is one of a variety of interaction diagrams and is intended for modeling the interaction of system objects in time, and also the exchange of messages between them. One of the basic principles of OOP is the way of information exchange between the elements of the System, expressed in sending and receiving messages from each other. Thus, the basic concepts of the sequence diagram are related to the concept of Object and Message.

The basic elements of the sequence diagram are the designations of objects (rectangles with the names of objects), vertical " lifeline ", reflecting the flow of time, rectangles reflecting the activity of an object or the performance of a certain function (rectangles on the dotted line of life), and arrows showing the exchange of signals or messages between objects.

Sequence diagrams of main scenarios of application are shown on Figure 15, 16, 17 and 18.

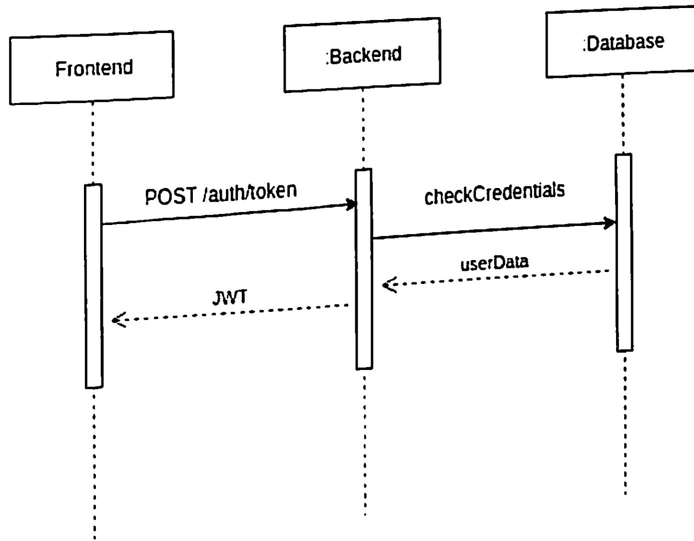


Figure 15. Sequence diagram for authorization

On first diagram shown authorization scenario:

a) user enters credentials and clicks login button
b) frontend framework sends POST request to backend
c) backend sends request to database for existence of user
d) database checks credentials and returns user information to backend
e) backend wraps user data to jwt and returns jwt to frontend
f) frontend redirects user to main page of application

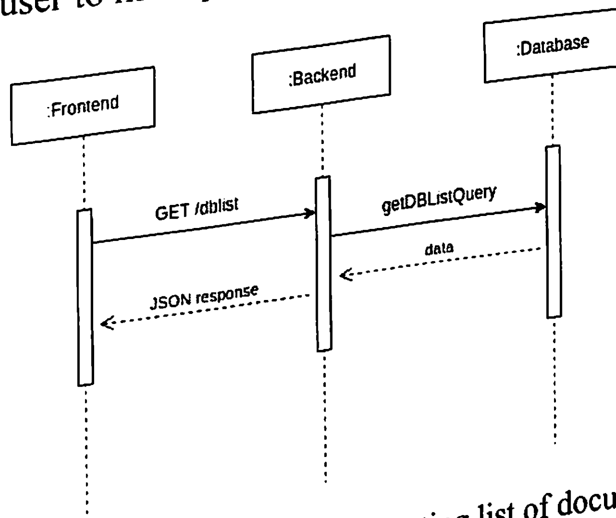


Figure 16. Sequence diagram for getting list of documents

On second diagram shown database page of frontend:

a) frontend sends GET request to backend with jwt authorization header
b) backend checks authorization header, if jwt is valid backend sends query to db

c) database returns list of documents

d) backend wraps list of documents to json and send to frontend

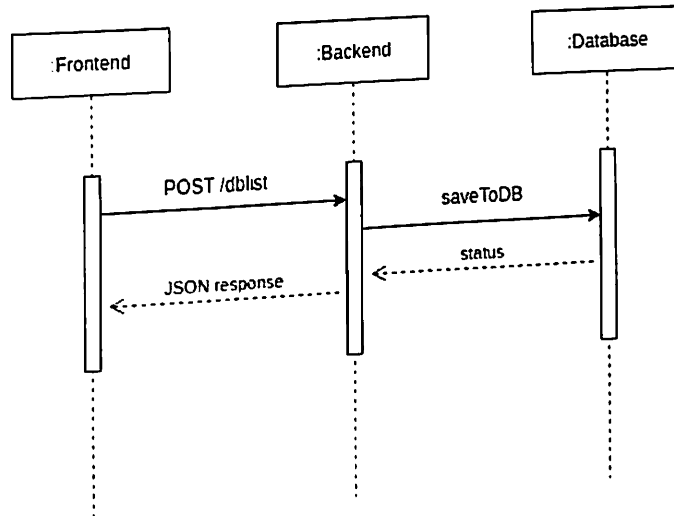


Figure 17. Sequence diagram for creating new document

Third diagram shows scenario of new document creation:

a) after filling all required fields, frontend sends POST request to backend with jwt authorization header

b) backend checks POST data and jwt, if data is valid it sends insert query to database

c) database returns query result to backend

d) backend sends JSON response with 201 code to frontend

e) frontend will notify user about result of insert query

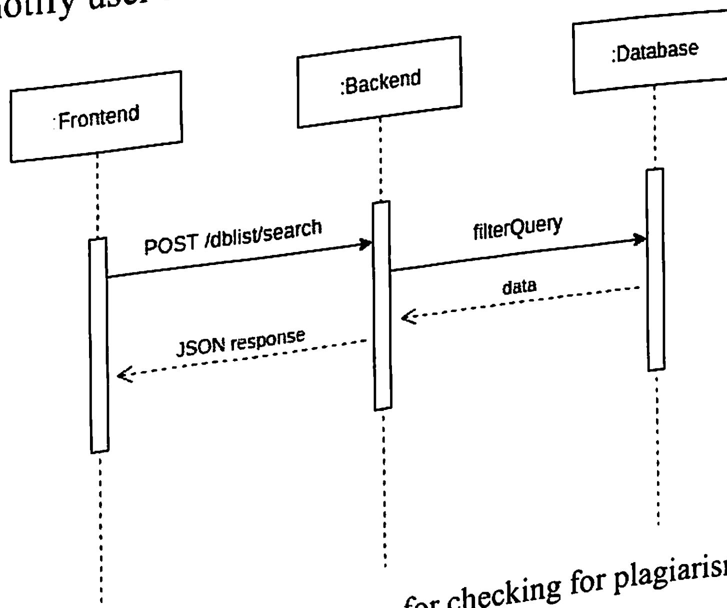


Figure 18. Sequence diagram for checking for plagiarism

Last diagram shows check for plagiarism scenario:

a) user fills required fields and frontend sends POST request to backend with jwt authorization header

b) backend checks POST data and jwt, if data is valid backend send query with filters like where document type equals some type and etc.

c) database returns list of documents to backend

d) backend checks for plagiarism, and returns near-duplicate document to frontend

e) frontend renders result to user

Class diagram

A class diagram is a diagram that demonstrates the classes of the system, their attributes, methods, and the relationships between them. It is central element to the design of the object-oriented system.

There are two types of class diagram:

- The static form of the diagram shows the logical relationships between classes.
- The analytical form of the diagram shows he general view and the interrelationships of the classes entering the system.

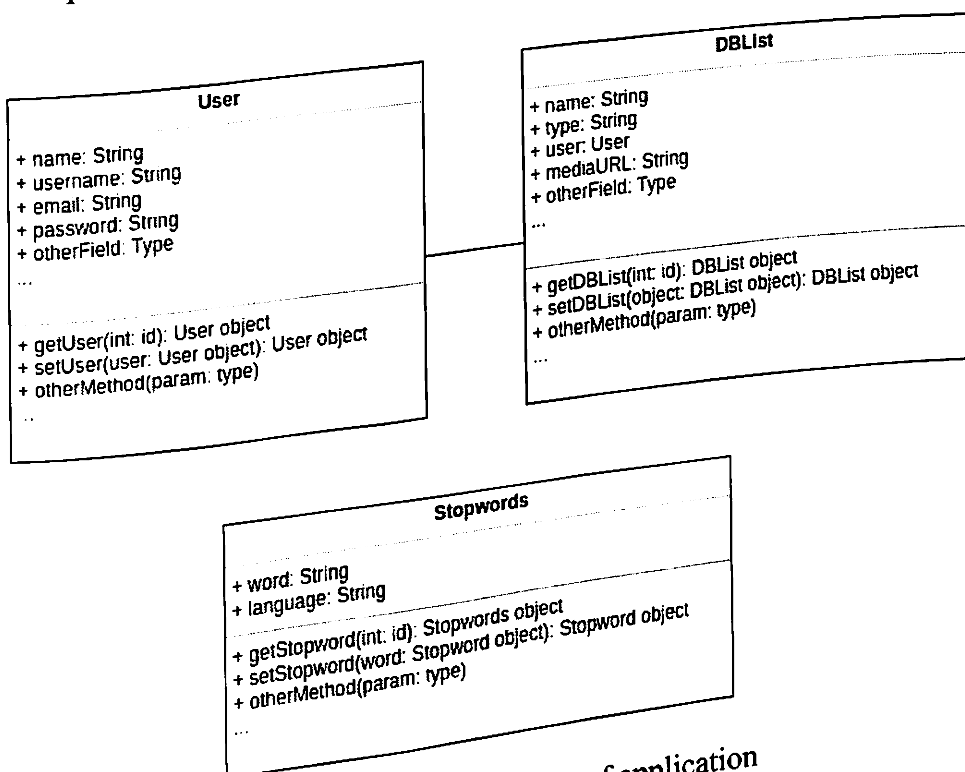


Figure 19. Class diagram of application

A class diagram is a key element in object-oriented modeling. In the diagram, classes are represented in a framework containing three components:

- Name of the class
- Attributes of class
- Class methods

Class diagram of main classes of application are shown on Figure 19. This diagram shows classes User, DBList and Stopwords main fields and methods. It also shows connection between DBList class and User class.

4.3 General scheme of work

To use application user needs to obtain JSON Web Token (jwt) from backend. Without jwt user can see only login page and 404 error page. Other pages of application requires jwt. Here obtaining jwt means authorization. Login page screenshot is shown of Figure 20.

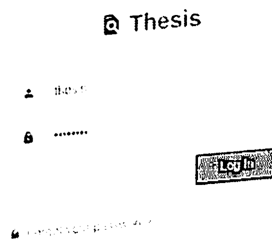


Figure 20. Login page of application

On login page user enters credentials and clicks login button. After clicking login button frontend sends POST request to backend of application. Backend sends query to database for existence user with given credentials. If credentials are not valid, then backend sends error to frontend, and frontend notifies user about incorrect credentials. If credentials are correct, database returns user object to backend, and backend generates jwt for user. By default jwt expires after 7 days. When jwt expiring

date is coming, frontend send POST request to backend to refresh jwt, and backend generates new jwt for user. Using jwt user can get access to other pages and operations.

After obtaining jwt, user redirects to main page of application. Main page of application is shown Figure 21. All routes are controlled by frontend framework.

Figure 21. Main page of application

Using menu at top of window user can add new document, check for plagiarism and can see list of document on system. On database page, user can see all list of documents with detailed information. Database page is shown on Figure 22.

Figure 22. Database page of application

To add new document to database user should click from menu 'Add new document' button. After clicking button, frontend renders AddNewDocument

component to user. On this page user can add new document to database using for
After filling required fields user should click 'Save' button.

After clicking 'Save' button frontend sends POST request to backend with jw
authorization header. Backend checks jwt and if jwt is invalid, user redirects to logi
page. If jwt is valid, backend checks POST data. If data is correct, it runs algorithm
which extracts text info from file, canonize it, divide canonized text into shingles, into
words, computes hashes for shingles, minhashes from shingles and save all data
database.

Figure 23. Add new document page of application

By precomputing hashes we can save time when user checks document for plagiarism. After saving to database it return JSON response to frontend, and frontend will show notification to user. Add new document page is shown on Figure 23.

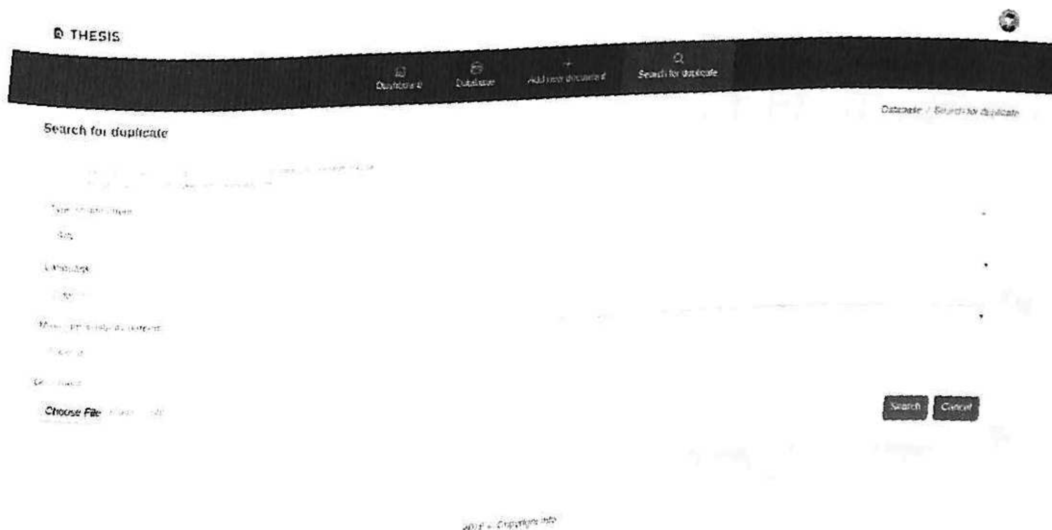


Figure 24. Search database page of application

To check for plagiarism user should click 'Search database' from top menu. When user clicks this button user will be redirected to search database page. Here user needs to fill all required fields and click 'Search' button. Search database page is shown on Figure 24.

After clicking 'Search' button, frontend send POST request to backend with jwt authorization header. Backend checks jwt header. If jwt is valid, it first runs algorithm which extracts text info from file, canonize it, divide canonized text into shingles, into words, computes hashes for shingles, minhashes for shingles.

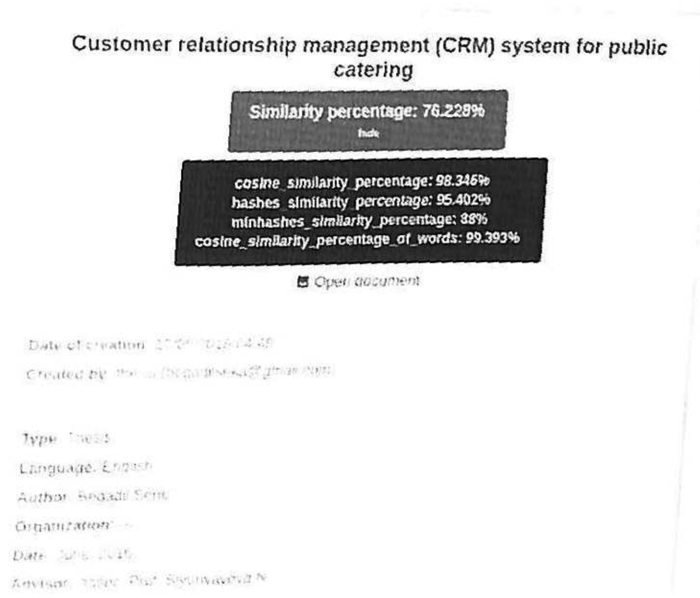


Figure 25. Similarity information

After this algorithm, backend sends query to database with filters like language, type of document and etc. Database returns list of documents to backend. Backend computes similarities between database documents and uploaded document. Then it calculates confidence interval for similarities. If calculated similarities are in confidence interval, then these similarities are used to calculate final average similarity value. If final similarity value is higher than value, which user sends with POST data, backend adds to some new list. After completing checking similarity step, backend sends list of data as JSON to frontend with list of similarities and confident interval. Then frontend renders result to user. By clicking 'show more...' button user can see all similarity values and confident interval as shown on Figure 25.

4.4 Experiments, comparisons and results

On development state of backend application I have tried different methods and algorithms for finding string similarities. Some of them I have implemented by myself by reading scientific papers, websites, and to some of them I have used already implemented libraries in python. One of methods works faster and accurate when texts are smaller, others works more accurate and faster when texts are bigger. I have tried to calculate similarities starting from small text to big text.

	exp1	exp2	exp3	exp4
text1	291	291	1321	31960
text2	505	1544	1544	1544

Figure 26. Experiments

Experiments are shown on Figure 26. Here values like 291, 1321 is word count in text1, and 505, 1544 is word count in text2. Exp1, exp2, exp3 and exp4 is experiment order.

	Shingles Hash	Shingles Minhash	Tf-idf Cosine similarity	bag of words	Difflib SequenceMatcher	Jellyfish Jaro_distance	Jellyfish Jaro_winkler	Fuzz Ratio	Fuzz Token_sort_ratio	Fuzz Token_set_ratio
exp1	0.002939	0.202202	0.013285	0.001124	0.067812	0.50832	0.507661	0.064857	0.065157	0.004856
exp2	0.010552	0.64201	0.042082	0.003802	0.084871	0.761914	0.752896	0.334767	0.309454	0.009202
exp3	0.017307	1.044477	0.050927	0.006723	0.072642	24.068413	23.955769	2.534251	2.289965	0.011951
exp4	0.360539	18.761202	2.053419	0.116886	56.594066	more than 120	more than 120	114.025676	95.636539	1.741437

Figure 27. Execution time for experiments

For each experiment, I run 10 different methods/algorithms and measure execution time. All methods/algorithms are written using python language. Result of time executions of methods are shown on Figure 27. Execution time measured on seconds. Execution time which works slower highlighted with red color.

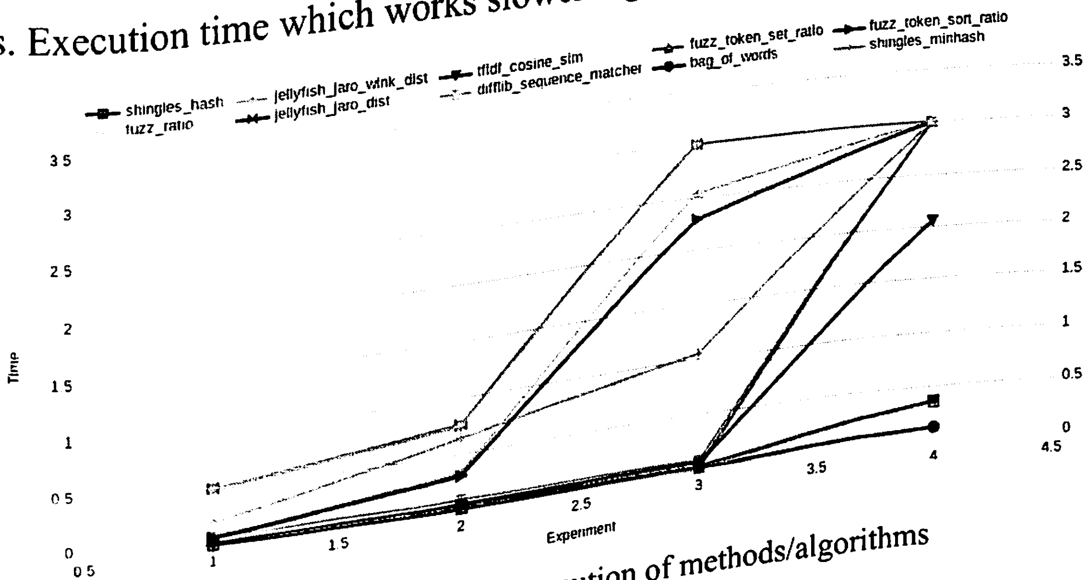


Figure 28. Chart of time execution of methods/algorithms

Chart for execution time of methods is shown on Figure 28. On chart X value is experiment and Y is execution time. To see differences I replace execution times which are bigger than 3 with 3. Chart shows us execution time of ten methods and algorithms. Execution time is calculated in seconds.

After this experiments to check for plagiarism on application I chose Shingle method, MinHash, TF-IDF with cosine similarity and Bag-of-words model with cosine similarity. To calculate final similarity between texts:

- calculate similarity for each selected method;
- calculate confidence interval for calculated similarities;
- removed similarity, if similarity not in confidence interval;
- calculate average similarity of remaining similarities.

4.5 Conclusion for fourth section

Fourth section of thesis is about application: UML diagrams of application, scheme of work of application, selected tools to write application, backend methods and endpoints of backend, frontend framework and different libraries used in frontend, and about experiments, comparisons and results.

CONCLUSION

This application is created for teachers. Application gives to teachers ability to check students works for plagiarism with the goal of improving education level at campus. I think application will be really interesting and useful.

Although, applications should be further developed. For example, we can use super-shingles, mega-shingles to increase performance of checking for plagiarism. Also we can use technologies like MapReduce, HBase, Hadoop to increase performance of searching for duplicates.

In conclusion, while doing this application I have learned a lot about string matching algorithms, similarity measures, REST API, image recognition using OpenCV, how to minimize execution time of functions and how to use useful libraries.

REFERENCES

1. Yu, F. T. S., Jutamulia, S. Optical Pattern Recognition. Cambridge University Press. 1998.
2. Optical character recognition - Wikipedia. https://en.wikipedia.org/wiki/Optical_character_recognition
3. Breuel, Thomas. "Recent Progress on the OCRopus OCR System". 2009.
4. Tesseract Open Source OCR Engine (main repository). https://github.com/tesseract-ocr/tesseract
5. jellyfish 0.5.6 documentation. http://jellyfish.readthedocs.io/en/latest/comparison.html
6. Chowdhury A., Frieder O., Grossman D., Mccabe C. 2002. Collection statistics for fast duplicate document detection. ACM Trans. Inform. Syst. 20(2):171–191.
7. Heintze N. 1996. Scalable document fingerprinting. 2nd USENIX Electronic Commerce Workshop Proceedings. 191–200.
8. Zelenkov Yuri. G., Segalovich Ilya V. 2007. Comparative analysis of near-duplicate detection methods of Web documents . IX All-Russian Scientific Conference “Digital Libraries: Advanced Methods and Technologies, Digital Collections” Proceedings. Pereslavl-Zalessky. 166–174.
9. Derbenev N. V., Tolcheev V. O. 2011. Using a method of detecting near duplicates in sciencemetric analysis. Information Technologies 12:24–29.
10. Broder A., Glassman S., Manasse M., Zweig G. 1997. Syntactic clustering of the Web. 6th World Wide Web Conference (International). 393–404.
11. U. Manber. Finding Similar Files in a Large File System. Winter USENIX Technical Conference, 1994.
12. N. Heintze. Scalable document fingerprinting. In Proc. of the 2nd USENIX Workshop on Electronic Commerce, Nov. 1996.
13. Д. Гасфилд. Строки, деревья и последовательности в алгоритмах. СПб.: Невский диалект, 2003.
14. A. Broder, S. Glassman, M. Manasse and G. Zweig. Syntactic clustering of the Web. Proc. of the 6th International World Wide Web Conference, April 1997.
15. A. Broder. On the resemblance and containment of documents. Compression and Complexity of Sequences (SEQUENCES'97), pages 21-29. IEEE Computer Society, 1998.
16. D. Fetterly, M. Manasse, M. Najork. A Large-Scale Study of the Evolution of Web Pages, WWW2003, May 20-24, 2003, Budapest, Hungary.

17. A. Chowdhury, O. Frieder, D. Grossman, M. McCabe. Collection statistics for fast duplicate document detection. ACM Transactions on Information Systems (TOIS), Vol. 20, Issue 2 (April 2002).
18. A. Kolcz, A. Chowdhury, J. Alspector. Improved Robustness of Signature-Based Near-Replica Detection via Lexicon Randomization. KDD 2004. http://ir.iit.edu/~abdur/publications/470-kolcz.pdf
19. W. Pugh. Detecting duplicate and near — duplicate files. http://www.cs.umd.edu/~pugh/google/Duplicates.pdf
20. S. Ilyinsky, M. Kuzmin, A. Melkov, I. Segalovich. An efficient method to detect duplicates of Web documents with the use of inverted index. WWW Conference 2002. https://pdfs.semanticscholar.org/c80a/b22173e78a4cfb6d4a74cbeae35831125b59.pdf
21. Форум технологий Mail.Ru Group: Поиск неточных дубликатов в рунете. https://www.searchengines.ru/tf_mailru_poisk_netochnyh_dublikatov_v_runete.html
22. Plagiarism Checker Tool Online - Copyleaks. https://copyleaks.com
23. String searching algorithm - Wikipedia. https://en.wikipedia.org/wiki/String_searching_algorithm#Na%C3%AFve_string_search
24. Tf-idf - A Single-Page Tutorial. http://www.tfidf.com
25. Machine Learning - Cosine Similarity for Vector Space Models. http://blog.christianperone.com/2013/09/machine-learning-cosine-similarity-for-vector-space-models-part-iii/
26. F1 score - Wikipedia. https://en.wikipedia.org/wiki/F1_score
27. pytesseract - PyPI. https://pypi.org/project/pytesseract/
28. Shingling - Near Duplicate Detection. http://phpir.com/shingling-near-duplicate-detection
29. Andrei Z. Broder. "Identifying and Filtering Near-Duplicate Documents". http://cs.brown.edu/courses/cs253/papers/nearduplicate.pdf
30. Peter Shaw. “Twitter Bootstrap 3 Succinctly: Design Attractive, Consistent UIs with Twitter Bootstrap”. 2014