Universität Rostock

Traditio et Innovatio

Bachelor thesis on the subject of

Detection of Similar Text Documents Based on Self-Organizing Maps

Degree course:	Informatik
Handed in by:	Kutiba Alahmad
Enrolment number:	219203159
Work period:	02. April 2024 – 01. October 2024
Supervisor:	Dr.-Ing. Hannes Grunert
Primary Reviewer:	Dr.-Ing. Hannes Grunert
Secondary Reviewer:	Felix Hauptmann, M.Sc.

Lehrstuhl für Datenbank- und Informationssysteme
Institut für Informatik
Fakultat für Informatik und Elektrotechnik
Universität Rostock

List of Acronyms

ANN	Artificial Neural Networks. 4–6, IV
NLP	Natural Language Processing. 8
SOM	Self-Organizing Maps. 6, 7, 13
TF-IDF	Term Frequency-Inverse Document Frequency. 10, 23, 24, 35, 40, 56

Contents

List of Figures

Abstract

Plagiarism of text has become a common occurrence today with difficulty in detecting forms such as paraphrasing being frequently practiced. This project presents an approach for detecting plagiarism in academic documents using Self-Organizing Maps (SOMs). The system leverages SOMs to cluster documents based on both word-level and context-level similarities, achieved through advanced text embeddings. Experimental results demonstrate the effectiveness of this approach in accurately detecting textual similarities and distinguishing between original and plagiarized content. Future enhancements include fine-tuning the embedding models and expanding the system's capabilities to handle multilingual.

keywords: Plagiarism Detection, Self-Organizing Maps, Sentence Embeddings, Document Clustering, Natural Language Processing, Academic Integrity

Chapter 1

Introduction

The theft of information in the form of computer data has significantly increased in the modern era. Plagiarism is defined as "construction, distortion, copying, or any other practice that seriously deviates from practices commonly accepted in the discipline or the educational and research communities generally in proposing, performing, reviewing, or reporting research and inventive activities" [14] and is a phenomenon that also occurs in the academic or educational era.

According to paper [14] Plagiarism comes in various forms, ranging from intentional to accidental. Direct plagiarism involves copying someone else's work word-for-word without credit, while self-plagiarism occurs when individuals reuse their previous work without acknowledgment. Mosaic plagiarism, or patchwriting, blends different sources without proper attribution, and paraphrasing plagiarism involves rewording someone else's ideas without credit. Even accidental plagiarism, due to improper citation, is still a serious issue. These forms highlight the need for integrity and proper referencing in academic and professional writing. Literal plagiarism includes copy-paste operations and is usually easy to detect. More sophisticated forms of plagiarism may involve translation, summarization, and paraphrasing and are more difficult to recognize

Research on the automated identification of possible plagiarism instances falls under the category of plagiarism detection techniques . This layer's papers usually offer techniques for analyzing textual similarity at the lexical, syntactic, and semantic levels in addition to similarities between non-textual content elements like mathematical formulas, figures, tables, and citations [6].

Generally, plagiarism detection techniques are based on four types; Lexical detection algorithms take into account a document's characters. Syntax-based detection methods consider the sentence structure, i.e., the parts of speech and their relationships. Sentences, paragraphs, or documents are compared for meaning in semantic-based detection techniques

[6]. Idea-based detection techniques take into account non-textual content elements such as images, citations, and mathematical content in addition to textual content analysis [6, 3]. Using Self-Organizing Maps (SOMs), a plagiarism detection system is the main goal of this project. Based on a technique from a paper [11], it offers a thorough analysis of the approaches, resources, and difficulties involved in identifying plagiarism in academic papers. The authors assess different methods for detecting plagiarism, such as machine learning and text-matching algorithms.

1.1 Objectives

The current work has been motivated by the widespread use of paraphrasing techniques for text plagiarism. This work aims to investigate the suitability of using a paraphrase recognition system based on machine learning for plagiarism detection.

The main objective of this project is to develop an efficient plagiarism detection system by leveraging Self-Organizing Maps (SOMs) to identify textual similarities between academic papers in terms of Copy-Paste problems and paraphrasing. The system aims to analyze and compare documents, distinguishing between original and plagiarized content. The core goals include:

1. Text Representation: Convert text data from research papers into meaningful vector representations using techniques like word embeddings for context matching and traditional word matching methods.
2. Similarity Measurement: Use SOMs to group and cluster similar documents based on both word-level and context-level similarities. This involves combining word and context similarity to form a comprehensive measure of document similarity.
3. Plagiarism Detection: Train the SOM model to detect clusters of documents that are too similar, indicating potential plagiarism. The system will compare original documents with their potentially plagiarized counterparts and highlight suspicious matches.
4. Evaluation: Assess the effectiveness of the system by providing it with plagiarized papers and checking whether it correctly identifies the original documents from which the content has been copied.

1.2 Structure of theses

The remainder of this thesis consists of 5 chapters structured as follows: Chapter 2 sets out the basic theory and background discussed in our research topic, including the different techniques of Text representation in the literature. Furthermore, Chapter 3 presents the proposed approach through the global architecture and the articulation between the different components (contextual model, semantic model, and clustering model). Then, Chapter 4 presents the development of the overall architecture of the proposed approach. Next, we present different metrics and use test cases to evaluate and validate the approach in Chapter 5. We end with a conclusion (Chapter 6) and an outlook on some perspectives and future works for interesting research directions.

Chapter 2

Study Background

In this chapter, we will explore artificial neural network concepts in section 2.1, especially self-organizing maps (SOM) (section 2.2). Next, in section 2.3, we will define Natural Language Processing (NLP) and its preprocessing techniques as well as the text representation methods as pre-trained models for word embeddings by introducing some popular models as well as Frequency-Based Methods. Finally, section 2.4 represents the vector preprocessing techniques we will utilize for that data before passing it to the SOM model, and section 2.5 presents the cosine similarity functionality that well be needed in the evaluation phase.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are a class of machine learning algorithms that are inspired by the way biological neural networks work (see Figure 2.1) [1]. ANN consists of multiple layers of interconnected nodes, or neurons, that process and transmit information. Each neuron receives input from other neurons and uses that input to compute its own output, which is then passed on to other neurons in the network [1].

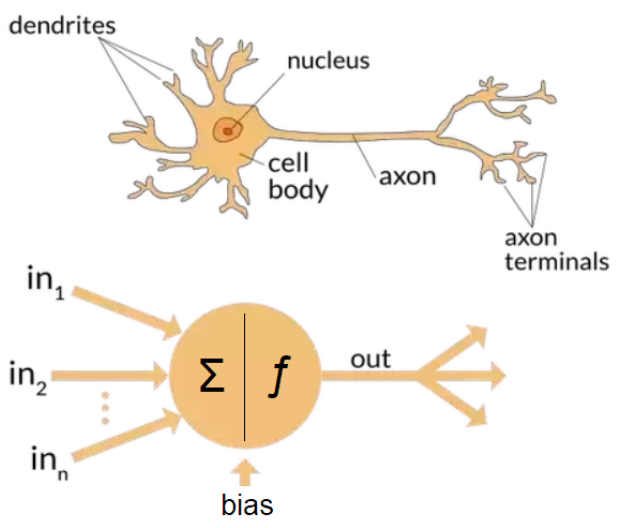


Figure 2.1: biological neural networks vs ANN taken from [1]

An artificial neural network (ANN) is generally structured into three types of layers: input, hidden, and output layers (as shown in Figure 2.2). The input layer is responsible for receiving the initial data, whether it's images, text, or other forms of raw information, and then forwarding it to the hidden layers. In these hidden layers, the data undergoes various transformations and processing steps. Finally, the output layer generates the final result, such as a prediction or classification, based on the processed data from the hidden layers.

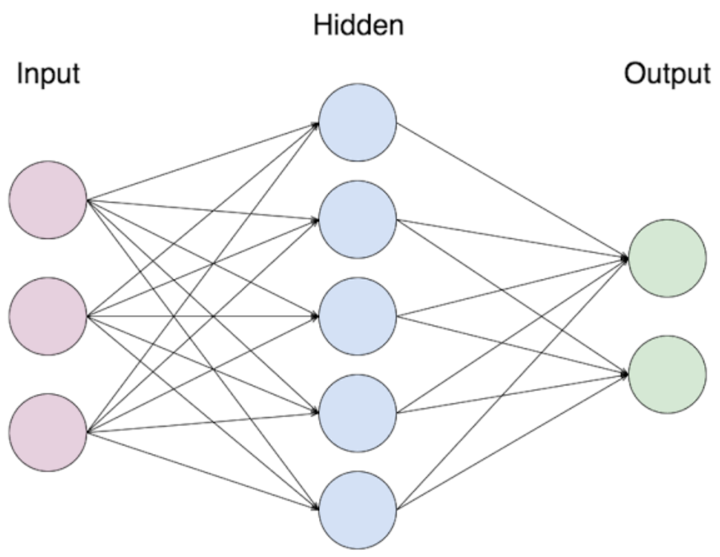


Figure 2.2: ANN architecture [1]

ANN are often used in supervised learning tasks, where the network is trained on labeled data to predict or classify new, unseen data [1]. During training, the weights and biases of the neurons in the network are adjusted based on the error between the predicted output and the true output. This process is repeated many times until the network is able to accurately predict or classify new data.

Weights are parameters associated with the connections between neurons in a neural network. Each connection between two neurons has an associated weight, which represents the strength or importance of that connection. The weights determine how much influence the input from one neuron has on the activation of the next neuron.

Biases are additional parameters in neural networks that provide an offset or a constant value to the input of a neuron. Each neuron in a network typically has an associated bias, which allows the network to introduce flexibility and account for variations in the data.

2.2 Self-Organizing Map

Self-organizing maps (SOMs) are a type of ANN that was developed by Teuvo Kohonen [10], which is inspired by biological models of neural systems from the 1970s. Kohonen's self-organizing map (SOM) is an abstract mathematical model of topographic mapping from the (motor, visual, auditory, etc.) sensors to the cerebral cortex. By accident, machine-based pattern recognition benefits from modeling and analyzing the mapping since it helps us understand how the brain interprets, encodes, identifies, and processes patterns [10].

The SOM is based on unsupervised learning, which means that no human intervention is needed during the training and those little needs to be known about characterized by the input data.

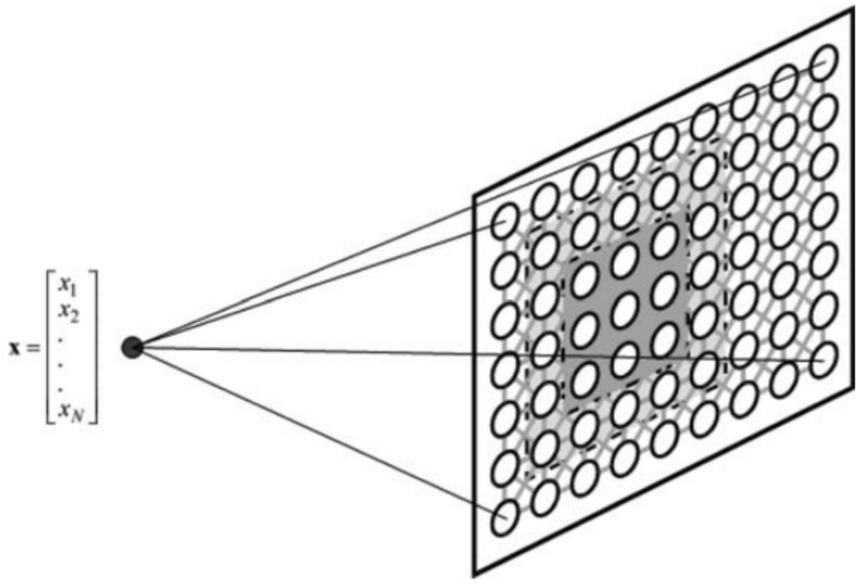

$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_N \end{bmatrix}$

Figure 2.3: Kohonen's self-organizing map model taken from [20]

Figure 2.3 represents Kohonen's self-organizing map model. The input is connected to every cell in the postsynaptic sheet (the map). Through the learning process, the map becomes localized, meaning that different local regions respond to distinct ranges of inputs. Lateral excitation and inhibition are modeled mathematically using a technique called local sharing, which modifies the learning mechanism [20]. As a result, there are no physical connections between the cells—grey lines are used to represent these virtual connections.

Also, SOM is based on competitive learning, which is a subset of machine learning that falls under the unsupervised learning algorithms. In competitive learning, a network of artificial neurons competes amongst themselves to be activated, with the result that only one is activated at any one time [20]. The "winning" neuron, which typically is the one that best matches the given input, is then updated while the others are left unchanged. The significance of this learning method lies in its power to automatically cluster similar data inputs, enabling us to find patterns and groupings in data where no prior knowledge or labels are given. This activated neuron is called a winner-takes-all neuron or the Best Matching Unit (BMU).

The form of SOM map, known as a topographic map, has two important properties:

1. Every piece of incoming information is retained in its appropriate context or neighborhood at each stage of representation, or processing.
2. Neurons handling similar pieces of information are maintained in close proximity to each other in order to facilitate short synaptic connections between them.

The Self-Organizing Map (SOM) uses a set of neurons, often arranged in a 2-D rectangular or hexagonal grid, to form a discrete topological mapping of input space, $X \in \mathbb{R}^n$ [20].

At the start of the learning, all the weights $\{w_1, w_2, \ldots, w_M\}$ are initialized to small random numbers. w_i is the weight vector associated with neuron i and is a vector of the same dimension n as the input. M is the total number of neurons, let r_i be the location vector of neuron i on the grid, , and Ω is the set of neuron indexes..

Then the algorithm repeats the steps shown in Algorithm 1 [20], where The winner neuron $\nu(t)$ is selected by minimizing the distance between the input vector $x(t)$ and the weight vectors $w_k(t)$ 2.1. The weights of the winner and its neighbors are updated based on the learning rate $\alpha(t)$ and the neighborhood function $\eta(\nu, k, t)$ 2.2. The algorithm repeats until the map converges. The coefficients $\{\alpha(t), t \geq 0\}$, termed the 'learning rate', are scalar-valued, decrease monotonically, and satisfy appropriate conditions for convergence. $\eta(\nu, k, t)$ is the neighborhood function that used a Gaussian form in practice. More specifically:

$$\eta(\nu, k, t) = \exp\left[-\frac{\|r_\nu - r_k\|^2}{2\sigma(t)^2}\right],$$

with σ representing the effective range of the neighborhood, which often decreases with time.

Algorithm 1 Self-Organizing Map Algorithm

```
1: repeat
2:     At each time step $t$, present an input vector $x(t)$, and select the winner:
           $$\nu(t) = \arg\min_{k \in \Omega} \|x(t) - w_k(t)\| \quad (2.1)$$
3:     Update the weights of the winner and its neighbors:
           $$\Delta w_k(t) = \alpha(t)\eta(\nu, k, t)[x(t) - w_{\nu(t)}] \quad (2.2)$$
4: until the map converges
```

straightforwardly, we can present the Algorithm 1 as :

Algorithm 2 Simplified Self-Organizing Map (SOM) Algorithm

```
1: repeat
2:     Select a random input vector.
3:     Find the best matching unit (BMU) on the map.
4:     Update the weight vectors of the BMU and its neighbors.
5:     Reduce the learning rate and neighborhood size over time.
6: until convergence or a predefined number of iterations is reached.
```

Self-Organizing Maps are a useful tool in many machine learning and data analysis domains because of their remarkable capacity to map high-dimensional, complex facts into a form that is visually insightful, topologically structured, and understandable.

2.3 Natural Language Processing

Natural Language Processing (NLP) is a sub-field of computer science, artificial intelligence, and linguistics that aims at understanding natural language using computers [2]. NLP uses computational linguistics, which is the study of how language works, and various models based on statistics, machine learning, and deep learning. These technologies allow computers to analyze and process text or voice data, and to grasp their full meaning, including the speaker's or writer's intentions and emotions.

The NLP concept consists of two main steps: first is the representation of the input text (raw data) into numerical format (vectors or matrix), and second is the design of models for processing the numerical data to achieve a desired goal or task [16].

NLP encompasses a wide array of techniques that are aimed at enabling computers to process and understand human language. These tasks can be categorized into several broad areas, each addressing different aspects of language processing.

2.3.1 Text Preprocessing

Text Preprocessing is the first step in the pipeline of Natural Language Processing (NLP), with potential impact in its final process. Text Preprocessing is the process of bringing the text into a form that is predictable and analyzable for a specific task.

1. Noise removal: Noise removal is about removing digits, characters, and pieces of text that interfere with the process of text analysis. It is one of the most important steps of text preprocessing. It is highly domain-dependent. For example, in the sentiment analysis, noise could be all the special characters but emojis have a significant sentiment index. The problem with noise is that it can produce inconsistent results if noisy, i.e., if uncleaned data is fed to the machine learning models.

   There are various ways to remove noise. This includes punctuation removal, special character removal, numbers removal, HTML formatting removal, URL removal, source code removal, header removal, and more. It all depends on which domain and what is categorized as noise for the task.

2. Tokenization: Tokenization is the task of breaking a character sequence up into pieces (words/phrases) called tokens. The list of tokens then is used to further processing [2].

3. Filtering: Filtering is usually done on documents to remove some of the words. A common filtering is stop-word removal. Stop words are the words that frequently appear in the text without having much content information (e.g. prepositions, conjunctions, etc.) [2].

4. Lemmatization: Lemmatization is the process of grouping together a word's various inflected forms so they can be examined as a single item, taking into account the morphological analysis of the words. Put differently, lemmatization techniques aim to associate nouns with a single form and verb forms with the infinite tense [2].

5. Text Normalization: Standardizing text format, including correcting spelling errors, expanding contractions, and handling special characters.

2.3.2 Text Representation

In Natural Language Processing (NLP), text representation describes the process of transforming textual input into a format that can be processed by machine learning models and algorithms. Textual stuff (such as sentences or documents) needs to be converted into a numerical form because machines only work with numerical data. A variety of natural language processing (NLP) activities, including text categorization, sentiment analysis, and language production, are directly impacted by the representation that is selected.

Statistical methods

In statistical methods, words are represented using vectors of numbers, and the corpus is represented as a collection of such vectors, forming a matrix [16]. These statistical techniques convert arbitrary-length documents into lists of numbers with a set length. These vector representations were useful because they allowed researchers to manipulate the vectors and calculate similarities and distances using linear algebra operations. This assisted in solving a far wider variety of issues than would have otherwise needed more manual coding of nested conditional rules and regular expressions.

Vectorization converts words into a numerical format, capturing not only their presence but also their importance using the TF-IDF (Term Frequency-Inverse Document Frequency) method. This method assigns weights to words based on how often they appear in a document relative to their frequency across all documents. Words that are frequent in a document but rare across the dataset receive higher weights, highlighting their unique contribution to the document's semantic content [16].

The TF-IDF values in the matrix indicate the importance of each term within each document.

TF-IDF Formula:

TF-IDF =TF $\times$ IDF

Where:

TF or Term Frequency represents how often a word appears in a document.
IDF is Inverse Document Frequency calculated as [19]:

$$\text{IDF(word)} = \log\left(\frac{\text{Total Number of Documents}}{\text{Number of Documents Containing the Word}}\right)$$

TF-IDF vectorization involves converting preprocessed documents into numerical features using the `TfidfVectorizer`. This process starts by calculating the term frequency (TF), which measures how often each word appears in a document. Next, the inverse document frequency (IDF) is computed for each term across the entire document set, giving higher importance to less common words. Finally, the TF-IDF score is determined by multiplying the TF and IDF values for each term, resulting in a weighted feature representation for the documents.

By employing TF-IDF vectorization, we capture the nuanced semantic relationships between words in the document corpus, enabling more sophisticated analysis and interpretation of text data.

Pre-trained models

A pre-trained model is a machine learning model that has been previously trained on a large dataset and is available for use in different tasks or applications. pre-trained models are a valuable resource allowing practitioners to build on existing knowledge and achieve high performance with less effort and computational cost.

The pre-trained word and sentence embeddings show good performance for NLP tasks due to their ability to retain the semantics and the syntax of the words in the sentence [16].

- **Global Vectors for Word Representation (Glove):**

  GloVe is a popular method for generating word embeddings, developed by researchers at Stanford University [17]. It is used to represent words in a continuous vector space, capturing semantic meanings based on the context in which words appear.

- **BERT (Bidirectional Encoder Representations from Transformers)**:

  BERT is a state-of-the-art model for natural language understanding, introduced by Google in 2018 [5]. It represents a significant advancement in how language models handle context and semantics in text.

- **SciBERT:**

  SciBERT is a variant of BERT specifically trained on scientific text. Developed by the Allen Institute for AI, SciBERT adapts the BERT architecture to handle the vocabulary and language patterns typical in scientific literature, making it particularly useful for tasks involving scientific documents [8].

- **Scientific Paper Embeddings using Contextualized Transformer (SPECTER**)

  SPECTER is another model designed for scientific text. Developed by researchers at the Allen Institute for AI, SPECTER extends the BERT architecture to generate embeddings specifically optimized for scientific documents [4].

2.4 Vector Preprocessing

In machine learning, the effectiveness of predictive models is greatly influenced by the quality of the input data. However, before raw data can be efficiently utilized by machine learning algorithms, it frequently needs to undergo extensive preparation. Vector preprocessing, which is modifying the dataset's features to make sure they are ready for model training, is an essential component of this architecture. In our project, this preparation stage involves dimensionality reduction, data scaling, and feature engineering. The effectiveness, stability, and performance of machine learning models are all improved by each of these procedures.

2.4.1 Feature Engineering

One of the main tasks in preparing data for machine learning is feature engineering. It is the practice of constructing suitable features from given features that lead to improved predictive performance [15]. Feature engineering is creating new features by applying transformation functions, like arithmetic and aggregate operators, to preexisting ones. A feature can be scaled or a non-linear relationship between a feature and a target class can be changed into a linear, more easily learned relation with the aid of transformations. The success of machine learning models heavily depends on the quality of the features used to train them.

Types of Feature Engineering:

- Domain-Specific: Creating new features based on domain knowledge, such as creating features based on business rules or industry standards.
- Data-Driven: Creating new features by observing patterns in the data, such as calculating aggregations or creating interaction features.
- Synthetic: Generating new features by combining existing features or synthesizing new data points.

While machine learning algorithms are designed to identify patterns and relationships within data, their effectiveness largely depends on the quality and relevance of the features (input variables) provided to them. Below are key reasons why feature engineering is important:

- Improves Model Performance: By providing additional and more relevant information to the model, feature creation can increase the accuracy and precision of the model.
- Increases Model Robustness: By adding additional features, the model can become more robust to outliers and other anomalies.
- Improves Model Interpretability: By creating new features, it can be easier to understand the model's predictions.
- Increases Model Flexibility: By adding new features, the model can be made more flexible to handle different types of data.

2.4.2 Data scaling

One method to standardize the independent features in the data within a predetermined range is feature scaling. It is done as part of the pre-processing of the data to handle greatly varied magnitudes, values, or units. In the absence of feature scaling, a machine learning algorithm will typically treat all values, regardless of unit, as higher and weigh larger values as such.

Scaling ensures that every feature has a comparable range and is on a comparable scale. Feature normalization is the term for this procedure. This is important because a lot of machine learning techniques depend on the size of the features. Larger scale elements might control the learning process and significantly affect the results. By scaling the features, you can prevent this issue and ensure that every feature makes an equal contribution to the learning process.

Moreover, When the features are scaled, several machine learning methods, including gradient descent-based algorithms, and distance-based algorithms (such SOM), perform better or converge more quickly. The algorithm's performance can be enhanced by scaling the features, which can hasten the convergence of the algorithm to the ideal outcome.

Avoiding large-scale differences between features is one way to stop numerical instability from occurring. Numerical overflow or underflow issues can arise from features with drastically different scales in, for instance, distance computations or matrix operations. Scaling the features helps to mitigate these issues and guarantees stable computations.

Ensuring that every characteristic receives equal consideration throughout the learning process is made possible by scaling features. Without scaling, learning could be dominated by features at a larger scale, leading to skewed results. Scaling ensures that each feature contributes equally to model predictions while also eliminating this bias.

2.4.3 Dimensionality Reduction

dimensionality needs to be reduced to handle large real-world data adequately. The process of converting high-dimensional data into a comprehensible representation with decreased dimensionality is known as dimensionality reduction. The dimensionality of the reduced representation should ideally match the dimensionality of the data intrinsically. The minimum number of parameters required to account for the observed properties of the data is known as the intrinsic dimensionality of the data [12].

Principal Components Analysis (PCA) is a linear technique for dimensionality reduction, which means that it performs dimensionality reduction by embedding the data into a linear subspace of lower dimensionality [12]. While there are several ways to accomplish this, PCA is the most often used (unsupervised) linear method. Therefore, we only use PCA in our technique.

A low-dimensional representation of the data is created via parse PCA, which attempts to capture as much of the variance in the data as feasible. To do this, the data must be reduced in dimensionality to a linear basis where the maximum variance can be found.

2.5 Cosine similarity

In the initial similarity assessment, the prepared word vectors from each document are compared using similarity metrics, such as cosine similarity. This comparison aims to identify documents that share a significant number of important words. Essentially, it helps to find documents with similar content or themes. This initial comparison serves as the foundation for grouping or linking documents that may be about similar topics.

The cosine similarity takes Vector representations of documents as input for the initial similarity assessment. The output of the initial similarity assessment is a set of initial similarity scores between pairs of documents. These scores indicate the degree of similarity between documents in terms of their word usage.

Let's calculate the cosine similarity between two vectors $\vec{A}$ and $\vec{B}$, which represent two documents. We will walk through the steps to calculate the **dot product**, **magnitude**, and the final **cosine similarity**.

Step 1: Vectors $\vec{A}$ and $\vec{B}$

Assume we have the following two vectors:

$$\vec{A} = [1, 2, 3]$$
$$\vec{B} = [4, 5, 6]$$

Step 2: Dot Product of $\vec{A}$ and $\vec{B}$

The dot product of two vectors is calculated by multiplying corresponding elements and summing them up:

$$\vec{A} \cdot \vec{B} = (1 \cdot 4) + (2 \cdot 5) + (3 \cdot 6)$$
$$\vec{A} \cdot \vec{B} = 4 + 10 + 18 = 32$$

Step 3: Magnitudes of $\vec{A}$ and $\vec{B}$

The magnitude (or length) of a vector $\vec{A}$ is calculated as the square root of the sum of the squares of its elements:

For $\vec{A} = [1, 2, 3]$:

$$\|\vec{A}\| = \sqrt{(1^2) + (2^2) + (3^2)} = \sqrt{1 + 4 + 9} = \sqrt{14} \approx 3.74$$

For $\vec{B} = [4, 5, 6]$:

$$\|\vec{B}\| = \sqrt{(4^2) + (5^2) + (6^2)} = \sqrt{16 + 25 + 36} = \sqrt{77} \approx 8.77$$

Step 4: Cosine Similarity

Finally, cosine similarity is the dot product of the vectors divided by the product of their magnitudes:

$$\text{Cosine Similarity} = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|}$$

$$\text{Cosine Similarity} = \frac{32}{3.74 \times 8.77} \approx \frac{32}{32.82} \approx 0.975$$

The cosine similarity between the two vectors $\vec{A}$ and $\vec{B}$ is approximately 0.975, which is very close to 1. This indicates that the two vectors (or documents) are highly similar in terms of their word usage or feature space.

2.6 Related Work

Recent advancements in plagiarism detection and document analysis have leveraged Self-Organizing Maps (SOMs) to enhance the accuracy and efficiency of these processes. Intrinsic Plagiarism Detection with Kohonen Self-Organizing Maps [18] explores the application of SOMs in detecting intrinsic plagiarism by analyzing variations in writing styles within documents. The study employs 49 enhanced style markers, including syntactic and parts-of-speech features obtained from dependency parsers. The research demonstrates the efficacy of SOMs in clustering documents based on stylistic characteristics, achieving a remarkable true positive rate of 100% for text segments of 450 words. However, the authors note that the ability to accurately identify true negatives—segments that do not share the same author—is not explicitly addressed, indicating a primary focus on maximizing true positive detection. The potential for false positives, which could lead to wrongful accusations, is also acknowledged, highlighting the importance of minimizing false negatives to ensure comprehensive detection of potential plagiarism cases.

In parallel, the work of Lensu and Koikkalainen [11] further illustrates the capabilities of SOMs in the domain of document retrieval. Their study, Similar Document Detection using Self-Organizing Maps, aims to develop a system for matching similar free-form textual documents. Utilizing a three-stage methodology that encompasses word matching, context matching, and document matching, the findings indicate that SOMs effectively cluster textual data to identify similar documents. This functionality not only enhances retrieval processes but also proves valuable for qualitative research, showcasing the efficiency of SOMs in handling diverse linguistic features.

These studies underscore the transformative role of Self-Organizing Maps in both intrinsic plagiarism detection and similar document identification, paving the way for broader applications of unsupervised learning techniques in complex textual analysis tasks.

2.7 Conclusion

Throughout this chapter, we have studied the basic concepts we use to develop our model. First, we presented the notion of self-organized map-based functionality, Text representation, and Text and vector preprocessing. The next chapter will be devoted to the study of the related works.

Chapter 3

Plagiarism detection proposed approach

This chapter proposed an approach for detecting similar text documents using a Self-Organizing Map (SOM). The design process encompasses various stages, starting from word-level analysis to contextual and document-level integration. The primary goal is to establish a robust framework that can accurately cluster and visualize documents based on their content and context.

3.1 Architecture overview

The diagram 3.1 illustrates a comprehensive approach to identifying plagiarism in academic papers. It highlights the process of collecting data from diverse sources, along with examples of plagiarized content. The core of the process involves meticulous text prepossessing to normalize and prepare the text for analysis.

The prepossessed text will pass through two mean methods which are word matching and context matching, those methods are used to assess the similarities between texts at both the word level and within their contextual terms. However, a text representation was completed beforehand in a different manner for the two methods to meet each method's requirements.

The combined score from these methods is then used to train a Self-Organizing Map model that clusters similar documents, highlighting potential instances of plagiarism. Finally, the model's performance is evaluated through metrics such as quantization error and unified distance matrix, ensuring its effectiveness in identifying plagiarized content. The remainder of the chapter will include a detailed presentation of each step.

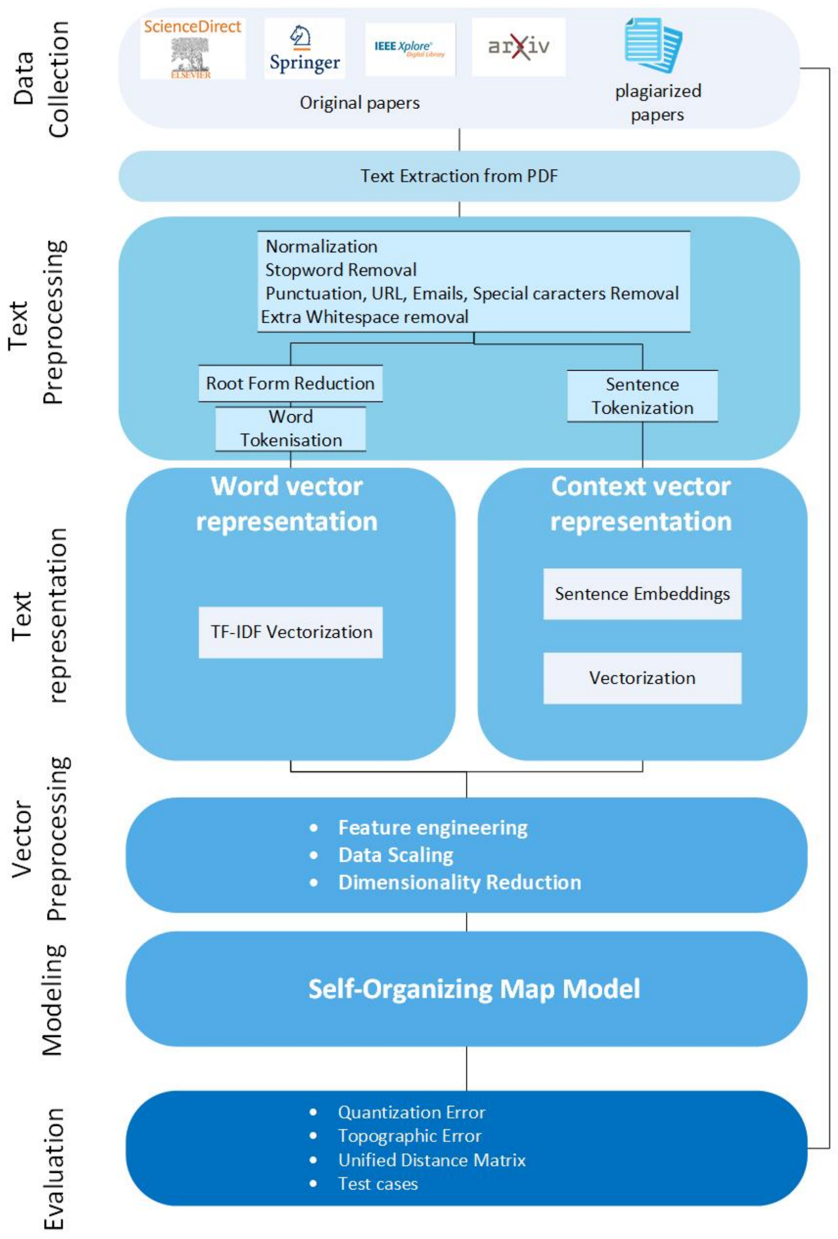


Figure 3.1: Architecture chart flow

3.2 Data Collection

The first step in any machine learning project is data collection, which is critical given its importance. A robust data collection strategy should include an efficient model. we gather the data from different scholarly databases and publishers such as ScienceDirect, Springer, IEEE Xplore, and arXiv.

We generate manually synthetic plagiarized papers that mirror real-world plagiarism techniques for effective training for the model. This involves two methods: direct copying and paraphrasing. Direct copying involves selecting text segments from original sources and directly pasting them into a new document. On the other hand, paraphrasing involves rewriting portions of the original text while maintaining the core meaning but altering word choice and sentence structure. This simulates a subtler form of plagiarism, often employed by students to avoid detection. By generating plagiarized papers using both these methods, the training data becomes more robust, enabling the system to identify a wider range of plagiarism techniques. the total number of documents is 100, Each generated document is categorized as either ”original” or ”plagiarized”, the data partition is as present in the figure 3.2. our data is unbalanced; 90 original papers and 10 plagiarized papers.

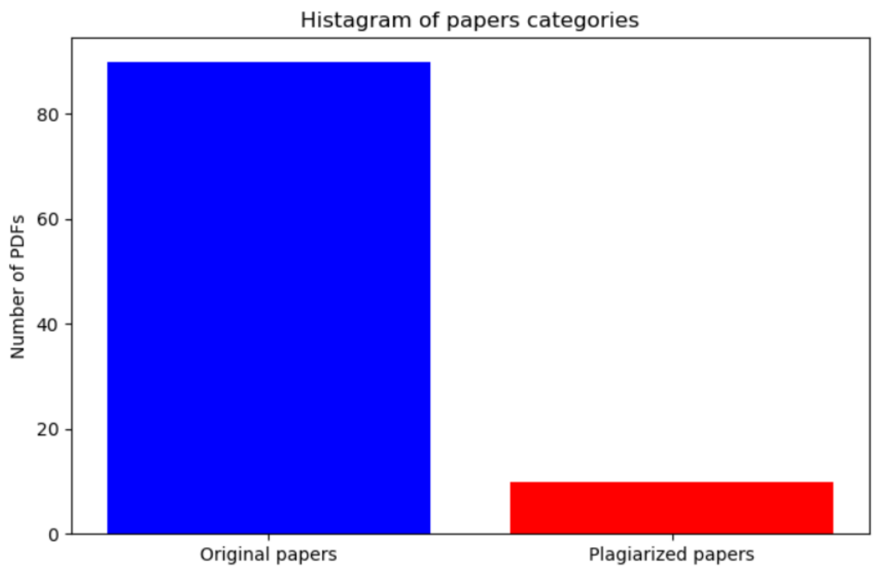


Figure 3.2: Data partition

3.3 Data Preprocessing

Pre-processing the data is the process of cleaning and preparing the text for the model. Data preprocessing is a critical phase in any data-driven project, especially when dealing with text data in natural language processing (NLP) tasks [2]. The quality of the data fed into a machine learning model or any analytical process significantly impacts the results. Thus, effective data preprocessing ensures that the text data is clean, structured, and ready for analysis.

3.3.1 Text Extraction and Normalization

The text extraction and normalization phases are for preparing raw text data for plagiarism detection. Text extraction involves converting raw documents, often in formats of PDF into plain text. This process removes any formatting, images, or other non-textual elements, leaving only the textual content for analysis. Normalization takes this extracted text and standardizes it for consistency by converting all characters to lowercase.

This transformation eliminates discrepancies caused by variations in capitalization, ensuring uniformity in the text corpus. By converting all words to lowercase, we create a level playing field for subsequent analysis, where distinctions based on case sensitivity are no longer relevant. It is the groundwork for unbiased text analysis by extracting meaningful words, filtering out extraneous characters, and standardizing the text dataset through lowercase conversion. This preparatory step sets the stage for more accurate and consistent text processing and analysis downstream.

3.3.2 Stopword Removal

The objective of stopword removal is to enhance the relevance and accuracy of text analysis by eliminating common words that carry little semantic value. By removing stopwords, we aim to sharpen the focus on terms that are more meaningful and informative, thereby improving the quality of similarity assessments and other text analysis tasks. These stopwords include frequently occurring words such as ”the”, ”is”, ”and”, etc., which are ubiquitous across texts but contribute little to the understanding of document content. By filtering out these stopwords from our text dataset, we ensure that our analysis prioritizes words that carry more unique information about the content of the documents. This filtering process helps to enhance the discriminative power of our analysis, allowing us to focus on terms that are more indicative of the underlying themes or topics within the documents. Example:

Consider the sentence: ”Machine learning algorithms are often used to analyze large datasets and make predictions.”

Stopwords List (from NLTK): [”are”, ”to”, ”and”, ”often”, and ”to”]

After removing the stopwords, the sentence is transformed to:

Processed Sentence: ”Machine learning algorithms used analyze large datasets make predictions.”

In this example, common words like ”to” and ”and” have been removed, leaving only the more informative words that contribute significantly to understanding the content. Stopword Removal serves to improve the quality and relevance of text analysis by eliminating common, low-information words from the dataset. By leveraging NLTK’s stopwords list, we enhance the discriminative power of our analysis, enabling more accurate and insightful assessments of document similarity and content.

3.3.3 Noise removal

URLs, emails, and special characters are removed to eliminate irrelevant information that could interfere with the analysis. Removing punctuation, URLs, emails, and special characters generally helps in cleaning the text, avoiding complications in parsing and tokenization processes, and reducing the number of irrelevant tokens that NLP algorithms have to process.

Example:

Text: "Machine learning models require vast amounts of data. However, not all data is useful; some of it can be noisy or irrelevant. Contact us at info@company.com for more details."

Preprocessed Output: ['machine', 'learning', 'models', 'require', 'vast', 'amounts', 'data', 'however', 'data', 'useful', 'noisy', 'irrelevant', 'contact', 'us', 'details']

3.3.4 Root Form Reduction:

The objective of Root Form Reduction is to normalize words to their base or root form, thereby facilitating more accurate comparisons between different texts. By reducing inflected or derived forms of words to their base forms, we aim to eliminate variations that may arise due to differences in tense, plurality, or other grammatical forms.

To achieve this, we use Lemmatization considers the part-of-speech (POS) of each word and relies on actual language models to ensure that the transformed words are valid and meaningful. Lemmatization is a more sophisticated approach than stemming because it considers the grammatical context of words. By analyzing the POS tags associated with each word, lemmatization accurately identifies the base or dictionary form of the word.

Root Form Reduction through Lemmatization is a sophisticated method for normalizing words to their base forms. By considering the part of speech and utilizing language models, lemmatization produces transformed words that are both linguistically valid and semantically meaningful, facilitating more accurate and insightful comparisons between different texts.

Example:

Processed text from Stopword Removal: "Machine learning algorithms used analyze large datasets make predictions"

Applying root form reduction (stemming or lemmatization) results in:

- **Machine** → *Machine* (often not reduced as it's already a base form)
- **Learning** → *Learn* (base form)
- **Algorithms** → *Algorithm* (base form)
- **Used** → *Use* (base form of the verb)
- **Analyze** → *Analyze* (already a base form)
- **Large** → *Large* (already a base form)
- **Datasets** → *Dataset* (base form)
- **Make** → *Make* (already a base form)
- **Predictions** → *Prediction* (base form)

By normalizing these words to their base forms, we reduce variations and make the text more consistent for analysis, enhancing the accuracy of comparisons and other NLP tasks.

The root from reduction technique is used only for word matching because Word matching methods rely on comparing the presence and frequency of words in documents. Root form reduction helps ensure that words with different inflections (e.g., "run", "running", "ran") are treated as the same base word, increasing the likelihood of detecting lexical similarities. instead, Context matching methods, which focus on deeper semantic relationships, may not need root form reduction as much and could even be hindered by it. Context Matching method requires techniques that understand and preserve the meaning and relationships between words in context, which root form reduction does not provide.

3.3.5 Tokenization

- **Word Tokenization:** This task involves breaking down the text into its individual word components. Each word becomes a basic unit for analysis, known as a token. Word tokenization is essential for tasks that require a granular understanding of language, such as word frequency analysis, lexical diversity studies, and most common NLP tasks like part-of-speech tagging.

  Example:

  - Input Text: "Hello World! This is a test."
  - Word Tokens: ["Hello", "World", "This", "is", "a", "test"]

- **Sentence Tokenization:** This process divides text into its constituent sentences. Understanding the boundary and context of each sentence is vital for tasks that analyze the structure and flow of texts, such as summarization, sentiment analysis, or when the context provided by sentence-level semantics is crucial.

  Example:

  – Input Text: ”Hello World! This is a test. Preprocessing and tokenization are critical for NLP.”

  – Sentence Tokens: [”Hello World!”, ”This is a test.”, ”Preprocessing and tokenization are critical for NLP.”]

3.4 Word Vector Representation

The Word vector representation method is an essential step in our NLP project that will be adapted to the next steps. The primary objective of this method is to represent the text as a vector so that the similarity can be calculated. TF-IDF vectorization refines the representation of a word numerically by considering both the term frequency and its importance across documents.

TF-IDF Calculation

Given the text: ”Machine learning algorithms used analyze large datasets make predictions”

Term Frequency (TF) Calculation

First, we preprocess the text and tokenize it: `["machine", "learning", "algorithm", "analyze", "large", "dataset", "prediction"]`

The term frequency (TF) is calculated as follows:

$$\text{TF}(t) = \frac{\text{Number of times term } t \text{ appears in the document}}{\text{Total number of terms in the document}}$$

For our document, the total number of words is 7. The frequency for each term is:

$$\text{TF(”machine”)} = \frac{1}{7} \approx 0.143$$

$$\text{TF(”learning”)} = \frac{1}{7} \approx 0.143$$

$$\text{TF("algorithm")} = \frac{1}{7} \approx 0.143$$

$$\text{TF("analyze")} = \frac{1}{7} \approx 0.143$$

$$\text{TF("large")} = \frac{1}{7} \approx 0.143$$

$$\text{TF("dataset")} = \frac{1}{7} \approx 0.143$$

$$\text{TF("prediction")} = \frac{1}{7} \approx 0.143$$

Inverse Document Frequency (IDF) Calculation

Assuming we have two documents:

1. Document 1: "Machine learning algorithms used analyze large datasets make predictions"
2. Document 2: "Machine learning is a field of artificial intelligence that uses algorithms to make predictions based on data"

The term "machine" appears in both documents. Hence:

$$\text{IDF("machine")} = \log\left(\frac{2}{2}\right) = \log(1) = 0$$

TF-IDF is calculated as:

$$\text{TF-IDF}(t) = \text{TF}(t) \times \text{IDF}(t)$$

For each term in Document 1:

$$\text{TF-IDF("machine")} = 0.143 \times 0 = 0$$

$$\text{TF-IDF("learning")} = 0.143 \times 0 = 0$$

$$\text{TF-IDF("algorithm")} = 0.143 \times 0 = 0$$

$$\text{TF-IDF("analyze")} = 0.143 \times \log\left(\frac{1}{2}\right) = 0.043$$

$$\text{TF-IDF("large")} = 0.143 \times \log\left(\frac{1}{2}\right) = 0.043$$

$$\text{TF-IDF("dataset")} = 0.143 \times \log\left(\frac{1}{2}\right) = 0.043$$

$$\text{TF-IDF("prediction")} = 0.143 \times 0 = 0$$

3.4.1 Context Vector Representation

During the vector representation stage, we tried several pre-trained models to determine which one was the best.

In this project, the pre-trained models are utilized to generate word embeddings considering the context of the sentence. The criteria considered for this comparative Comparative Analysis are the running time and the effectiveness in disguising context similarity.

The experience involves three papers: the first paper (paper(1)) presents a novel perspective on adversarial machine learning [21], the second paper (paper(2)) examines the use of traditional Chinese medicine in managing neuropsychiatric symptoms associated with Alzheimer's disease [9], and the third paper (paper(3)) discusses the concept of modularity in machine learning solution development [13]. Consequently, papers (1) and (2) differ significantly in their topics, while papers (1) and (3) also address distinct subject areas.

Pre-trained model	Context Similarty score		Running Time (in seconds)	
	paper(1) vs paper(2)	paper(1) vs paper(3)	paper(1) vs paper(2)	paper(1) vs paper(3)
GloVe	0.92	0.96	25	115
BERT	0.99	0.99	35	170
SciBERT	0.78	0.81	8	4
SPECTER	0.64	0.8	5	8

Table 3.1: Comparative Analysis of Pre-trained Models for Sentence Embeddings

GloVe and BERT show relatively high context similarity scores for both paper(1) vs paper(2) and paper(1) vs paper(3) (3.1), implying that these models are more general-purpose and can identify broader semantic relationships. SciBERT and SPECTER show lower context similarity scores, especially for paper(1) vs paper(2), which suggests that these models may be more sensitive to domain-specific content. SciBERT, being specialized for scientific text, still shows moderate similarity, but SPECTER, which is tailored for academic publications, shows the least similarity, reflecting its focus on specific academic content.

On the other hand, SciBERT and SPECTER have significantly shorter running times compared to GloVe and BERT, making them more efficient for quick analysis. BERT is the most computationally expensive model, indicating that its superior context similarity comes with a trade-off in processing time.

SPECTER is the best choice as it is designed to understand the nuances of academic papers or scientific content. Its efficiency in running time also makes it practical for large-scale analyses.

3.4.2 Vector Aggregation

Aggregate vectors in both representation, word and context, into a single document vector, which represents the entire document in a fixed-dimensional space. This aggregation can be achieved through simple averaging.

Example:

- For the sentence "The impact of climate change on urban planning."
  For demonstration purposes, let's assume we use SPECTER to generate embeddings for the following words (note: these values are hypothetical and simplified):
  - "impact": [0.45, 0.32, 0.54, 0.26]
  - "climate": [0.37, 0.48, 0.61, 0.22]
  - "change": [0.50, 0.30, 0.55, 0.30]
  - "urban": [0.42, 0.37, 0.50, 0.28]
  - "planning": [0.39, 0.45, 0.57, 0.33]
- To create a single vector representing the sentence, we compute the average of these word embeddings:
  - Average vector:

$$\left[\frac{0.45+0.37+0.50+0.42+0.39}{5}, \frac{0.32+0.48+0.30+0.37+0.45}{5},\right.$$
$$\left.\frac{0.54+0.61+0.55+0.50+0.57}{5}, \frac{0.26+0.22+0.30+0.28+0.33}{5}\right]$$

  - Resulting document vector: [0.43, 0.38, 0.55, 0.28]

3.5 Document matching using Self Organizing Map

Self Organizing Map (SOM) is an advanced neural network model designed to organize high-dimensional data into two-dimensional maps. Our interest is in building artificial topographic maps that learn through self-organization. During competitive learning, the neurons selectively adapt to different input patterns or classes of input patterns. A meaningful coordinate system for the input features is created on the graph by sorting the locations of the neurons that are tuned, so the neurons become the winning neurons.

In the context of text analysis, Document SOM leverages the capabilities of Self-Organizing Maps to visualize and cluster documents based on their vector representation.

This approach encapsulates both the word and the context, providing a holistic view of the document dataset. This method facilitates a nuanced exploration, revealing underlying patterns, themes, and groupings that might not be apparent with more traditional methods. The process involves combining feature vectors derived from both word-level analyses and contextual embeddings and then processing these combined vectors using the SOM to uncover the inherent structure within the document collection.

3.5.1 Feature engineering

Feature engineering in the context of a Document Self-Organizing Map (SOM) involves combining the word vector and context vector. Calculating combined similarity provides a way to compute a comprehensive similarity score between two documents by integrating both word-level and context-level information. This function then calculates a weighted average of these vectors, effectively combining the individual contributions to produce a more nuanced representation of overall similarity.

This approach allows for a more holistic understanding of the relationship between entities by considering both their semantic meaning and the context in which they are used. By combining word-level and context-level similarity, we're essentially constructing a new, richer feature that captures a more nuanced understanding of the relationships between entities.

3.5.2 Data Scaling

Scaling the combined matrix data before feeding it into the SOM is important for optimal performance. The SOM learns by iteratively adjusting its weights based on the input data. If features have vastly different ranges, those with larger ranges will dominate the learning process, leading to an imbalanced map representation. By standardizing the data, we ensure that all features contribute equally to the weight adjustments, enabling the SOM to learn more accurately and produce a map that reflects the underlying relationships between entities based on their combined matrix. This scaling process effectively levels the playing field for all features, allowing the SOM to learn a more balanced and representative map of the data. Standardization of a dataset is a common requirement for many machine learning estimators. The formula is used to normalize data, transforming it so that it has a mean of 0 and a standard deviation of 1.

$$X_{\text{scaled}} = \frac{X_i - X_{\text{mean}}}{\sigma}$$

- X_{scaled}: The scaled value or Z-score of the data point.
- X_i: The original value of the data point.

- X_{mean}: The mean (average) of the data set.
- σ: The standard deviation of the data set.

Example

Consider the matrix:

$$\text{combined\_vector\_matrix} = \begin{bmatrix} 0.75 & 0.55 & 0.65 \\ 0.35 & 0.45 & 0.55 \end{bmatrix}$$

To standardize this matrix, follow these steps:

1. Calculate the Mean and Standard Deviation for Each Column:

For column 1:

$$x_{\text{mean}_1} = \frac{0.75 + 0.35}{2} = 0.55$$

$$\sigma_1 = \sqrt{\frac{(0.75 - 0.55)^2 + (0.35 - 0.55)^2}{2}} = 0.28$$

For column 2:

$$x_{\text{mean}_2} = \frac{0.55 + 0.45}{2} = 0.50$$

$$\sigma_2 = \sqrt{\frac{(0.55 - 0.50)^2 + (0.45 - 0.50)^2}{2}} = 0.07$$

For column 3:

$$x_{\text{mean}_3} = \frac{0.65 + 0.55}{2} = 0.60$$

$$\sigma_3 = \sqrt{\frac{(0.65 - 0.60)^2 + (0.55 - 0.60)^2}{2}} = 0.07$$

2. Apply the Scaling Formula to Each Element:

For column 1:

$$x_{\text{scaled}_{11}} = \frac{0.75 - 0.55}{0.28} = 0.71$$

$$x_{\text{scaled}_{21}} = \frac{0.35 - 0.55}{0.28} = -0.71$$

For column 2:

$$x_{\text{scaled}_{12}} = \frac{0.55 - 0.50}{0.07} = 0.71$$

$$x_{\text{scaled}_{22}} = \frac{0.45 - 0.50}{0.07} = -0.71$$

For column 3:

$$x_{\text{scaled}_{13}} = \frac{0.65 - 0.60}{0.07} = 0.71$$

$$x_{\text{scaled}_{23}} = \frac{0.55 - 0.60}{0.07} = -0.71$$

Thus, the scaled matrix is:

$$x_{\text{scaled}} = \begin{bmatrix} 0.71 & 0.71 & 0.71 \\ -0.71 & -0.71 & -0.71 \end{bmatrix}$$

Scaling features ensure that each characteristic is given the same consideration during the learning process. Without scaling, bigger scale features could dominate the learning, producing skewed outcomes. This bias is removed through scaling, which also guarantees that each feature contributes fairly to model predictions.

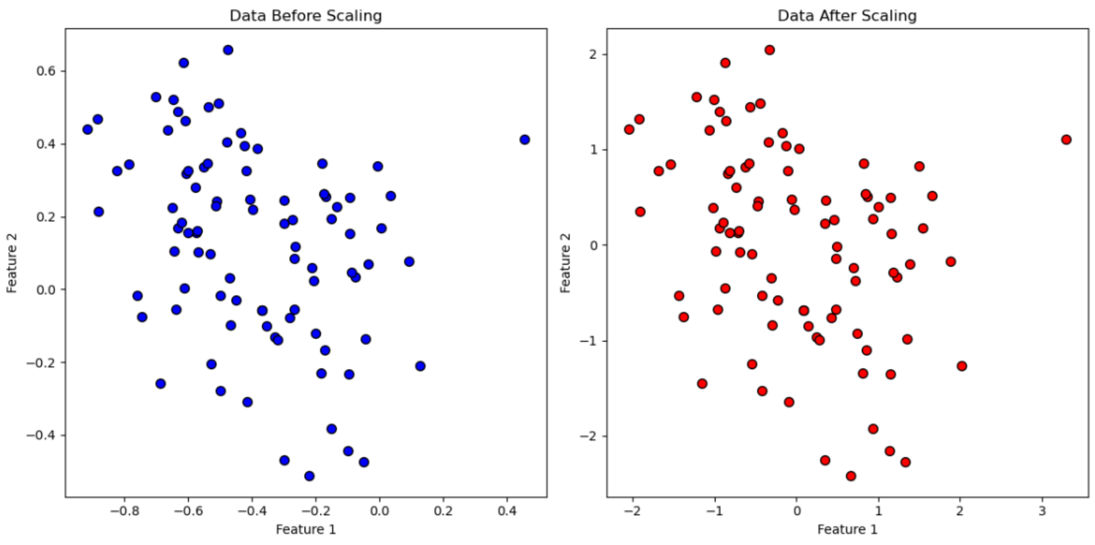


Figure 3.3: data before and after scaling

Figure 3.3 shows the effect of scaling data before applying a machine learning model. The left plot, showing the data before scaling, actually represents data that is already in a relatively small range. The data points are clustered closely together, indicating a smaller scale. This is often the case with real-world datasets.

The right plot, representing the data after scaling, shows data that has been stretched or compressed to have a more uniform range. This is done to ensure that all features contribute equally to the learning process and prevent features with larger scales from dominating the model. Scaling doesn't necessarily mean making data points closer together. It means adjusting the range of values to be more consistent across all features.

3.5.3 Dimensionality Reduction

In Machine Learning, it is known that the more the number of features the better the prediction, but it is not always working. If we keep on increasing the number of features, after a certain point, the performance of our machine learning algorithm tends to decrease.

PCA is a linear dimensionality reduction technique that converts a set of correlated features in the high dimensional space into a series of uncorrelated features in the low dimensional space. These uncorrelated features are also called principal components.

PCA is an orthogonal linear transformation which means that all the principal components are perpendicular to each other. It transforms the data in such a way that the first component tries to explain the maximum variance from the original data. It is an unsupervised algorithm i.e. it does not take into consideration the class labels.

One of the most popular approaches is to select the number of components that explain a large portion of the variance (e.g., 90%). it can be determined by plotting a scree plot or using the cumulative sum of the explained variance ratio.

In our case, we choose n_components From the plot, Figure 3.4, where the curve starts to flatten out, or where the cumulative explained variance reaches your desired threshold (e.g., 90%) which presents 28 components.

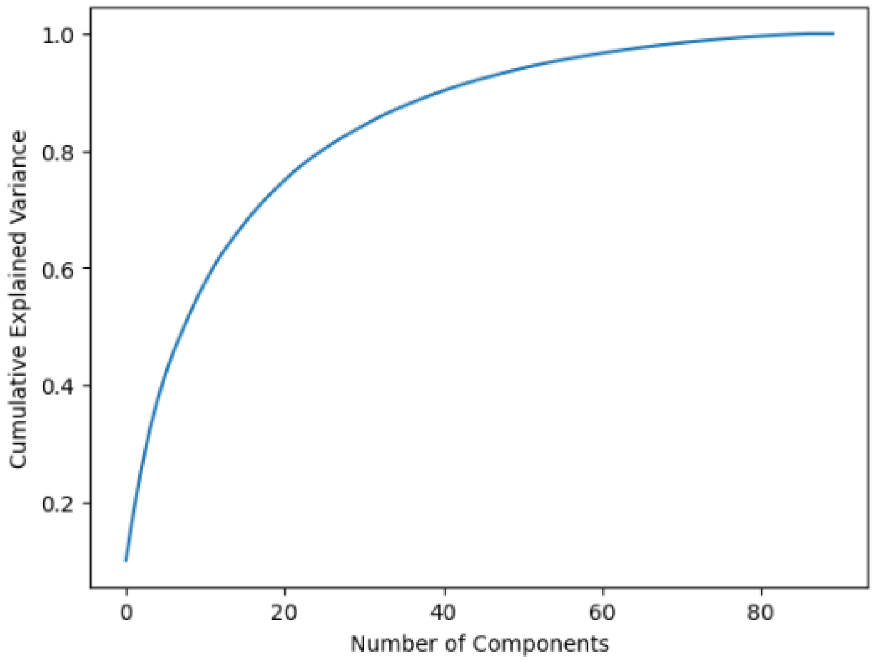


Figure 3.4: Cumulative Explained Variance vs. Number of Components

3.5.4 Training the SOM

Training a Self-Organizing Map (SOM) with the integrated feature set of document vectors involves several key stages, each crucial for effectively capturing the complex relationships and patterns within your data. The process is designed to map high-dimensional vectors onto a two-dimensional grid where similar items are clustered together spatially. Here's a structured approach to executing this process This process involves several key stages:

1. Choosing Dimensions:

   The first step is to select an appropriate size for the SOM grid. This involves balancing computational efficiency with the desired granularity of clustering. According to Minisom source code, the rule of thumb that set the size of the grid for a dimensionality reduction task is that it should contain 5*sqrt(N) neurons. where N is the number of samples in the dataset to analyze.

   Example: if your dataset has 150 samples, 5*sqrt(150) = 61.23 hence a map 8-by-8 should perform well.

   For 90 samples:
   $$\text{Grid size} = 5 \times \sqrt{N}$$

   Substituting $N = 90$:

   $$\text{Grid size} = 5 \times \sqrt{90} \approx 5 \times 9.49 \approx 47.45$$

   A grid with around 49 neurons would be ideal. Therefore, we could use a 7-by-7 grid (49 neurons) for a close approximation.

2. Initialization:

   The node weights in the SOM grid must be initialized. This is done randomly or by using a heuristic based on the properties of the data. we give the random state an integer. so the random initialization be the same for each run.

3. Competitive Learning:

   During training, each document vector is processed to identify the closest node in the grid, known as the Best Matching Unit (BMU). The BMU and its neighboring nodes are then adjusted to more closely resemble the document vector. This adjustment helps the map to learn the structure of the data gradually.

4. Iterative Refinement:

   Refine the positions of nodes through repeated exposure to the training data, allowing the SOM to represent the underlying structure of the data better.

5. Cycle through the entire dataset multiple times:

   Each cycle (epoch) refines the arrangement of the nodes. The learning rate, controlling how much node weights change in response to each vector, typically starts higher and decreases over time to stabilize learning. The process continues until changes to the node weights are minimal, indicating that the map has largely stabilized and reflects the data's structure effectively.

3.6 Conclusion

In this chapter, we have detailed the proposed approach. After describing the similarity detection model architecture as well as the case of study architecture and the overall process to detect plagiarism. We introduced the various components of plagiarism detection, which include the Pre-Processing Component, the text representation Component, and the SOM modeling Component. Additionally, we provided information on the dataset utilized in our case study.

Chapter 4

Implementation

The analysis and processing of textual data in this project leverage several advanced tools and methodologies. These tools are integral to the preprocessing, text representation, and clustering phases of our approach. They enable efficient and accurate handling of large volumes of text data, ensuring that our analysis captures both the syntactic and semantic nuances of the documents. In this chapter, we will justify our technical choices (programming language, libraries) used for developing and running the model of our project. Then, we will discuss the concrete implementation details of a prototype.

4.1 Tools and libraries

The analysis and processing of textual data in this project involve several advanced tools and methodologies. These tools are integral to the preprocessing, text representation, and clustering phases of our approach. Below is a detailed overview of the tools and techniques used.

4.1.1 Text Extraction

- os(3.9.5)[1]
  The 'os' module provides a way of using operating system-dependent functionality such as reading or writing to the file system, managing paths, and working with environment variables.
- PyPDF2(3.0.1)[2]
  PyPDF2 is a library for reading and manipulating PDF files. It allows splitting, merging, encrypting, decrypting, and extracting text or metadata from PDF documents.

[1] os library : https://docs.python.org/3/library/os.html
[2] PyPDF2 library: https://pypdf2.readthedocs.io/en/latest/

4.1.2 Text Preprocessing

- re[3] The re module provides regular expression matching operations similar to those found in Perl. It is useful for pattern matching and string searching within texts.
- nltk.tokenize(3.9.1)[4] The nltk.tokenize module provides functions for dividing text into sentences or words, a fundamental task in natural language processing (NLP). It includes tokenizers for both word and sentence tokenization.
- nltk.corpus.stopwords(3.9.1)[5] This part of NLTK provides a list of common stopwords for different languages, which can be removed from texts to improve the performance of NLP tasks by reducing the noise from frequently occurring words.
- nltk.stem.WordNetLemmatizer(3.9.1)[6]
  The WordNetLemmatizer is a tool for word lemmatization, which reduces words to their base or dictionary form. It helps in handling different forms of a word during text processing, such as "running" to "run."

4.1.3 Vector Preprocessing

- NumPy (1.21.0)[7]
  NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays. It is widely used for scientific and numerical computing.
- StandardScaler(1.5.1)[8]
  'StandardScaler' is a preprocessing tool in scikit-learn that standardizes features by removing the mean and scaling them to unit variance. This is commonly used to normalize data before applying machine learning algorithms that assume normally distributed data.
- PCA[9]
  'PCA' (Principal Component Analysis) is a technique from scikit-learn's decomposition module that is used for dimensionality reduction. It transforms the original features into a set of linearly uncorrelated components while preserving as much variance in the data as possible. This method is widely used to simplify datasets, improve visualization, and reduce noise in machine learning applications.

[3] re library: https://docs.python.org/3/library/re.html
[4] NLTK Tokenize: https://www.nltk.org/api/nltk.tokenize.html
[5] NLTK Stopwords: https://www.nltk.org/nltk_data/
[6] WordNetLemmatizer: https://www.nltk.org/_modules/nltk/stem/wordnet.html
[7] NumPy library: https://numpy.org/
[8] StandardScaler:https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
[9] PCA : https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

4.1.4 Text Representation

- TfidfVectorizer[10]
  The 'TfidfVectorizer' in scikit-learn converts a collection of raw documents into a matrix of TF-IDF (Term Frequency-Inverse Document Frequency) features. It is used in information retrieval and text mining.

- SentenceTransformer(3.0.1)[11]
  SentenceTransformers is a Python framework for computing dense vector representations for sentences and paragraphs. It uses transformer-based models like BERT, SciBERT, or Specter to generate sentence embeddings for text similarity tasks. .

- spaCy (3.0.0)[12] spaCy is an open-source library for advanced natural language processing (NLP) in Python. we use GloVe to convert the text into numerical data that can be processed by SOM algorithm. torch[13]
  PyTorch is an open-source machine learning library used for applications such as natural language processing. It provides tensor computation with strong GPU acceleration and deep learning neural network capabilities.

4.1.5 SOM Modeling

- SOM (Self-Organizing Map 0.2.3)[14]
  The 'MiniSom' class implements Self-Organizing Maps (SOMs), an unsupervised learning method for visualizing high-dimensional data. SOMs reduce data dimensionality and cluster similar data points on a grid, making it useful for tasks like document clustering and similarity detection.

4.1.6 Evaluation

- matplotlib (3.4.3)[15]
  Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is often used for plotting graphs and visualizing data for scientific research and analysis.

[10]TfidfVectorizer: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html
[11]SentenceTransformer: https://www.sbert.net/
[12]spaCy library: https://spacy.io/
[13]PyTorch: https://pytorch.org/
[14]MiniSom: https://pypi.org/project/MiniSom/
[15]matplotlib library: https://matplotlib.org/

- cosine_similarity(1.5.1)[16]
  Cosine Similarity is a metric used to measure how similar two documents or vectors are, based on the cosine of the angle between them. It is widely used in text similarity tasks.

4.1.7 Text Extraction phase

This phase provides two functions for loading text from PDF files using the 'PyPDF2' library as shown in the source code. The first function, 'load_pdf_texts_from_folder', processes all PDF files within a specified folder. It iterates through the folder, identifies files with a '.pdf' extension, and extracts the text from each page of every PDF. The extracted text, along with the corresponding filename, is stored in a list of tuples and returned. The second function, 'load_pdf_text_from_file', is designed to extract text from a single PDF file. It reads each page in the file, extracting the text and appending it to a string, which is then returned. Both functions ensure the extracted text is safe to process, handling cases where text extraction may fail on certain pages. These functions are useful for reading and processing large volumes of PDF documents for tasks like text mining or document similarity analysis.

Source Code

```python
def load_pdf_texts_from_folder(folder_path):
    """Loads text from PDF files in a given folder"""
    pdf_files = []
    for filename in os.listdir(folder_path):
        if filename.endswith(".pdf"):
            file_path = os.path.join(folder_path, filename)
            with open(file_path, "rb") as pdf_file:
                pdf_reader = PyPDF2.PdfReader(pdf_file)
                num_pages = len(pdf_reader.pages)
                text = ""
                for page_num in range(num_pages):
                    page = pdf_reader.pages[page_num]
                    text += page.extract_text()
                pdf_files.append((filename, text))  # Store filename and text
    return pdf_files

def load_pdf_text_from_file(file_path):
    """Loads text from PDF files"""
    reader = PdfReader(file_path)
    text = ""
    for page in reader.pages:
        text += page.extract_text()
    return text
```

[16]Cosine Similarity : https://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html

4.1.8 Preprocessing phase

The `preprocess_text()` function performs a series of text preprocessing tasks on a given input string. The process includes text normalization, tokenization, stopword removal, and lemmatization, applied to both words and sentences. Below is a breakdown of each step:

1. Normalization: The input text is converted to lowercase to standardize word comparison. Punctuation is removed, and URLs, email addresses, and special characters are filtered out. Consecutive spaces are also reduced to a single space.
2. Tokenization: The text is split into individual words using word_tokenize, and into sentences using sent_tokenize. This creates a list of words and sentences for further processing.
3. Stopword Removal: Commonly used words (stopwords) like "the," "is," and "and" are removed from the tokenized words and sentences. The stopword list is provided by NLTK.
4. Lemmatization: Words in the filtered sentences are reduced to their base forms using the WordNetLemmatizer. For instance, "running" becomes "run," ensuring consistency in text processing.

Source Code

```python
def preprocess_text(text):
    """
    Process the input text: normalize, tokenize, remove stopwords, and
    lemmatize from both words and sentences.

    Parameters:
    text (str): The text to preprocess.

    Returns:
    tuple: A tuple containing a list of words without stopwords and a list of
    sentences without stopwords.
    """
    # Normalize text: convert to lowercase and remove punctuation
    text = text.lower()
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove emails
    text = re.sub(r'\S+@\S+', '', text)

    # Remove special characters (keep punctuation like . , ? !)
    text = re.sub(r"[^a-zA-Z0-9.,!?\'\`]", " ", text)

    # Replace multiple spaces with a single space
    text = re.sub(r'\s+', ' ', text).strip()

    # Tokenization: split into words or sentences
    words = word_tokenize(text)
    original_sentences = sent_tokenize(text)

    # Stopwords removal for words
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in words if word not in stop_words]

    lemmatizer = WordNetLemmatizer()

    # Stopwords removal for sentences
    filtered_sentences = []
    for sentence in original_sentences:
        # Tokenize the sentence into words
        sentence_words = word_tokenize(sentence)
        # Filter out the stopwords
        filtered_sentence_words = [word for word in sentence_words if word not in stop_words]
        # Lemmatize words in the sentence
        filtered_sentence_words = [lemmatizer.lemmatize(word) for word in filtered_sentence_words]
        # Reconstruct the sentence (convert words back to strings)
        filtered_sentence = ' '.join(str(word) for word in filtered_sentence_words)
        filtered_sentences.append(filtered_sentence)

    return filtered_words, filtered_sentences
```

While the filtered sentences will go to the Context vectorization method because they contain more meaning than words, the filtered words will go to the Word vectorization method.

Example:

text:

"Natural Language Processing (NLP) is a field of artificial intelligence. It helps computers understand, interpret, and respond to human language. Visit https://www.example.com for more information or email us at info@example.com! It's an exciting area with many applications, like chatbots and language translation."

Output:
Filtered Words:

```
['natural', 'language', 'processing', 'nlp', 'field', 'artificial', 'intelligence',
'helps', 'computers', 'understand', 'interpret','respond', 'human', 'language',
'exciting', 'area','many','applications', 'like', 'chatbots', 'language',
'translation']
```

Filtered Sentences:

```
['natural language processing nlp field artificial intelligence',
 'helps computer understand interpret respond human language',
 'exciting area many application like chatbot language translation']
```

4.1.9 Text Representation

Word vectorization method

The compare_word function takes a list of words, converts it into a single string, and then applies Term Frequency-Inverse Document Frequency (TF-IDF) vectorization. It transforms the input text into a numerical vector representation based on the frequency of words in the text. The function can be useful for comparing texts based on the importance of words within a document.

Source Code

```python
def compare_word(text):
    # Convert list of words into a single string
    text = ' '.join(text)

    # Pass the single document as a list to the vectorizer
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform([text])

    # Extract the TF-IDF vector for the single document
    tfidf_result = tfidf_matrix[0]

    return tfidf_result
```

This would output the TF-IDF vector representation of the words in the list. The values represent the importance of each word in the text based on the frequency and inverse document frequency concept.

Input Words:

```
words = ['natural', 'language', 'processing',
'intelligence']
```

Output:

```
  (0, 5)  0.47493398290539906
  (0, 2)  0.47493398290539906
  (0, 3)  0.33179900815220113
```

For example, in ”(0, 5) 0.47493398290539906”, 0 refers to the index of the document (since we only have one document). 5 refers to the index of the word in the vocabulary created by TfidfVectorizer. 0.47493398290539906 is the TF-IDF value for the word at index 5.
The sparse matrix indicates which words have non-zero TF-IDF values, with the corresponding importance of each word in the document.

Context vectorization method

The context_vec function you provided generates a vector representation of a text by encoding it into embeddings and then aggregating them. The function uses model.encode() to convert the input text into a tensor of embeddings. However, the model is a pre-trained sentence transformer model such as (BERT, GloVe, SciBERT, and Specter).

The embeddings are aggregated into a single vector using mean aggregation (torch.mean(embeddings, dim=0)). This step combines the individual word embeddings into a single document vector. The resulting tensor is converted to a NumPy array (document_vector_np) for further numerical operations.

Source Code

```python
def comparing_context(text):

    # Encode texts into embeddings
    embeddings = model.encode(text, convert_to_tensor=True)

    # Aggregate embeddings to form a single vector
    document_vector = torch.mean(embeddings, dim=0).unsqueeze(0)

    # Convert to NumPy arrays for cosine similarity
    document_vector_np = document_vector.detach().numpy()

    return document_vector_np
```

The output will be a NumPy array representing the vector for the input text. This vector can then be used to compare with other text vectors or for various other text analysis tasks.

4.1.10 Vector Preprocessing

Feature Engineering:

The calculate_combined_vector function is designed to combine two types of vectors—word vectors and context vectors—using specified weights. This function aligns the dimensions of the input vectors and then computes a weighted sum to produce a combined vector.

Source Code

```
def calculate_combined_vector(word_vector, context_vector, word_weight=0.5,
    context_weight=0.5):
    # Resize to match dimensions
    min_dim = min(word_vector.shape[1], context_vector.shape[1])
    word_vector_resized = word_vector[:, :min_dim]
    context_vector_resized = context_vector[:, :min_dim]

    # Combine similarities
    combined_vector = word_weight * word_vector_resized + context_weight *
    context_vector_resized

    return combined_vector
```

1. **Resizing Vectors:**
    - Calculate `min_dim`, the smaller dimension between `word_vector` and `context_vector`, to ensure both matrices have the same number of columns.
    - Slice both vectors to this common dimension to align them for combination.

2. **Combining Vectors:**
   - Compute a weighted sum of the resized vectors. The weights for word and context vectors are provided as arguments (`word_weight` and `context_weight`), defaulting to 0.5 each.
   - The combined vector is computed as shown in ligne 8 in the source code.
3. **Return:**
   - The function returns the combined vector.

Example:

```
word_vector = np.array([[0.8, 0.6, 0.7], [0.4, 0.5, 0.6]])
context_vector = np.array([[0.7, 0.5, 0.6], [0.3, 0.4, 0.5]])
```

The output will be a NumPy array representing the combined vector based on the specified weights.
Combined Vector: [[0.75 0.55 0.65] [0.35 0.45 0.55]]

Data Scaling
Scaling the combined vectors guarantees that all features are on a comparable scale and have comparable ranges. This is significant because the magnitude of the features has an impact on many machine learning techniques including SOM. Larger scale features may dominate the learning process and have an excessive impact on the outcomes. You can avoid this problem and make sure that each feature contributes equally to the learning process by scaling the features.
When the features are scaled, our distance-based algorithms perform better or converge more quickly. The algorithm's performance can be enhanced by scaling the features, which can hasten the convergence of the algorithm to the ideal outcome.

To standardize this matrix, we use the StandardScaler from scikit-learn. The scaling process involves the following steps:

```
# Scale the data
scaler = StandardScaler()
combined_vector_scaled = scaler.fit_transform(combined_vector_matrix)
```

Example:

$$\text{combined\_vector\_matrix} = \begin{bmatrix} 0.75 & 0.55 & 0.65 \\ 0.35 & 0.45 & 0.55 \end{bmatrix}$$

The scaled matrix will be

$$x_{\text{scaled}} = \begin{bmatrix} 0.71 & 0.71 & 0.71 \\ -0.71 & -0.71 & -0.71 \end{bmatrix}$$

Dimensionality Reduction

We apply Principal Component Analysis (PCA) to reduce the dimensionality of the 'scaled_data' to 28 components. We start with initializing PCA with the number of components set to 28 that be chosen based on a large portion of the variance (e.g., 90%)(Figure 3.4). The data will be reduced to 28 features while retaining as much variance as possible. Then, we Fit the PCA model to the 'scaled_data' and transform the data, returning the reduced dimensional data.

This approach will enhance SOM by reducing the complexity of the input while retaining most of the meaningful variance.

```
# Scale the data
pca = PCA(n_components=28)
reduced_data = pca.fit_transform(scaled_data)
```

4.2 Self Organizing Map Implementation

Document Self-Organizing Map (SOM) involves combining different types of data representations—specifically, word-level features and contextual features—into a single, unified representation.

Code Implementation:

```
# Train SOM
x_dim, y_dim = 10,10
som = MiniSom(x=x_dim, y=y_dim, input_len=reduced_data.shape[1], sigma=1.0,
    learning_rate=0.8, random_seed=0)
som.train_random(reduced_data, num_iteration=1000)
```

The Self-Organizing Map (SOM) is initialized with dimensions $x = x\_dim$ and $y = y\_dim$, which represent the number of nodes in the SOM grid. The input vectors have dimensionality equal to the number of columns in the scaled matrix, `combined_vector_scaled`. The parameter $\sigma = 1.0$ defines the neighborhood radius, controlling how far the influence of a winning node spreads across neighboring nodes, and the learning rate is set to $learning\_rate = 0.8$.

Additionally, random_seed $= 0$ is used to ensure reproducibility by fixing the random initialization of the weights.

The SOM is then trained using the function `som.train_random` on the scaled data, `combined_vector_scaled`, for 1000 epochs. During training, the SOM adjusts its weights by finding the Best Matching Unit (BMU) for each input vector and updating the weights of the BMU and its neighboring nodes, iteratively clustering the data in an unsupervised manner.

4.3 Conclusion

In this implementation chapter, we detailed the steps involved in processing and analyzing textual data using various natural language processing (NLP) tools and techniques. In the next section, we will evaluate the performance of our approach through a series of experiments and case studies.

Chapter 5

Evaluation and Case Studies

In this chapter, we delve into the evaluation of our Self-Organizing Map (SOM) model's performance in detecting plagiarism. Evaluation is a crucial step in assessing the effectiveness of any machine learning model, providing insights into its accuracy, robustness, and overall utility. We define the key metrics used to gauge the model's performance, including silhouette score, quantization error, and topographic error.

We then explore validating the SOM by comparing the results of the model's clustering with known original and plagiarized documents. This includes analyzing how well the SOM clusters the plagiarized documents about their original counterparts and assessing the model's ability to identify similarities accurately. Finally, we examine the model's performance visually and quantitatively, using various plots and metrics to ensure that the SOM effectively captures the underlying structure of the data and performs reliably in practical scenarios.

5.1 Evaluation Metrics

To assess the performance of the Self-Organizing Map (SOM) in detecting plagiarism and clustering document similarity, several evaluation metrics are utilized. This section discusses the metrics employed, namely Quantization Error, Topographic Error, and Silhouette Score.

5.1.1 Quantization Error

Quantization error measures the average distance between the input vectors and their respective Best Matching Units (BMUs) on the SOM grid [7]. It is a common measure of the quality of representation of the input data by the map. A lower quantization error indicates that the SOM neurons are effectively capturing the input data distribution.

The quantization error is computed as follows:

$$QE = \frac{1}{N} \sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{w}_{bmu}\| \tag{5.1}$$

where N is the number of input samples, $\mathbf{x}_i$ is the i-th input vector, and $\mathbf{w}_{bmu}$ is the weight vector of the BMU corresponding to $\mathbf{x}_i$. Reducing the quantization error improves the accuracy of the SOM in representing the data.

5.1.2 Topographic Error

Topographic error evaluates how well the SOM preserves the topological relationships between input data points. This metric calculates the proportion of data points for which the first and second Best Matching Units (BMUs) are not adjacent on the SOM grid [7]. A lower topographic error indicates better preservation of the input space structure.

The topographic error is defined as:

$$TE = \frac{1}{N} \sum_{i=1}^{N} \delta(\mathbf{x}_i) \tag{5.2}$$

where $\delta(\mathbf{x}_i) = 1$ if the first and second BMUs of $\mathbf{x}_i$ are not adjacent, and $\delta(\mathbf{x}_i) = 0$ otherwise. The topographic error ranges between 0 and 1, with lower values indicating better topological preservation.

A low topographic error signifies that the map has successfully captured the structure of the data, with most neighboring data points in the input space being mapped to adjacent units on the SOM grid. Conversely, if the topographic error approaches 1, it suggests that many neighboring data points are being mapped to distant units, which means the map fails to maintain the data's topological structure.

5.2 Hyperparameter Optimization

Hyperparameter optimization is the process of finding the best values for the hyperparameters of a machine learning model.SOM hyperparameters are learning rate, sigma, size of map and number of epochs. Therefore, SOM are typically optimized by a process of trial and error where different hyperparameter combinations are tested and the one with the best performance is chosen.

Figure 5.1 shows the SOM results for different values of alpha and sigma. Alpha is a learning rate parameter that controls how much the weights of the SOM are adjusted during training. Sigma is a neighborhood parameter that controls the size of the neighborhood of neurons that are updated during training. Based on the rule of thumb mentioned in the Minsom code source, the grid size is 7X7.

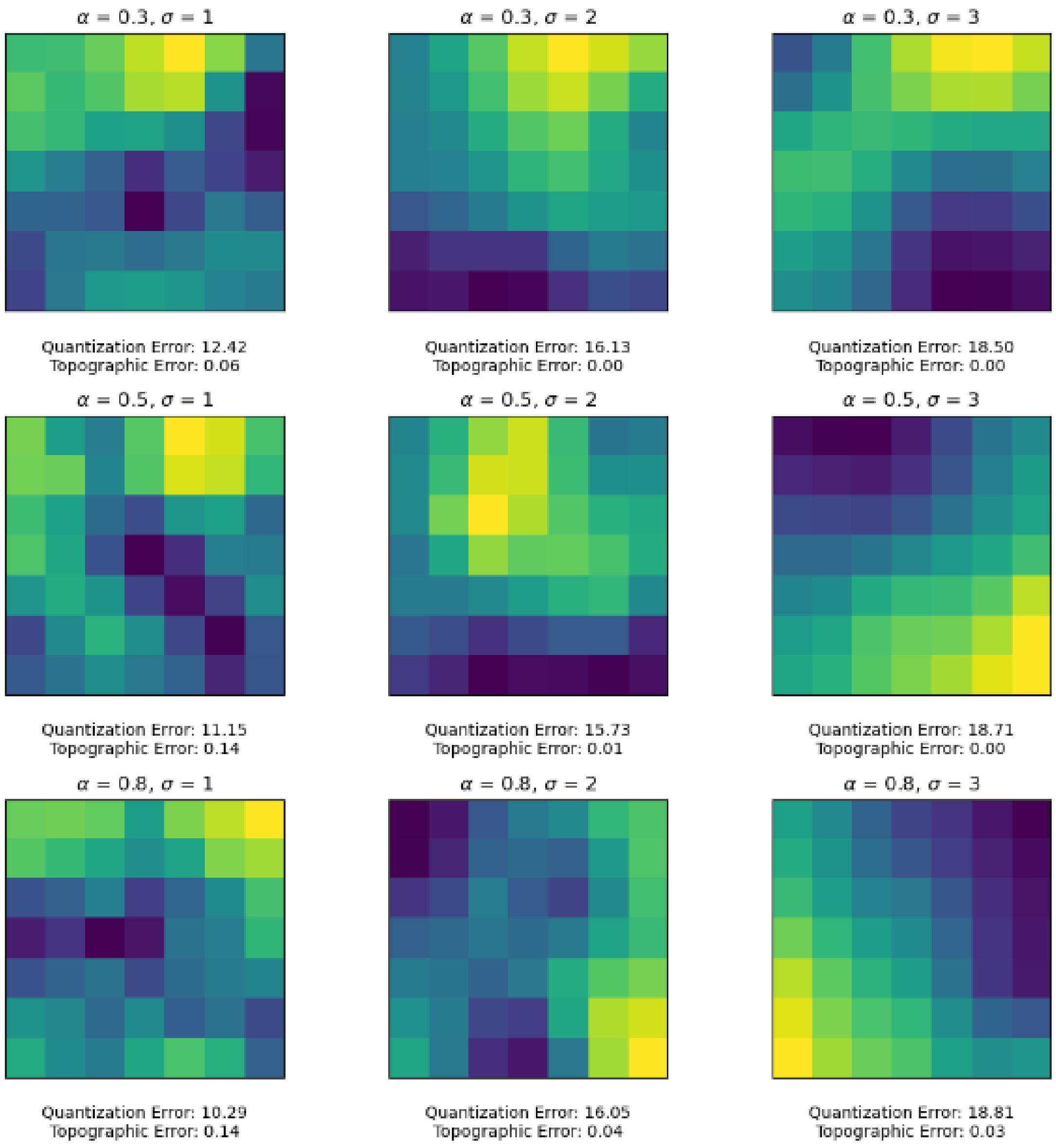


Figure 5.1: Hyperparameters learning rate and sigma Optimization (map size: 7x7)

The Figure 5.1 shows that the best parameters for the SOM in this case are alpha = 0.8 and sigma = 1. This combination of hyperparameters results in a SOM with low quantization error and low topographic error compared to the other SOMs. This indicates that the SOM with these hyperparameters is better at representing the data and preserving the spatial relationships between the data points.

When increasing the size of the Self-Organizing Map (SOM), the quantization error tends to decrease as shown in Figure 5.2 till we reach 20X20, which is desirable. Balancing the map

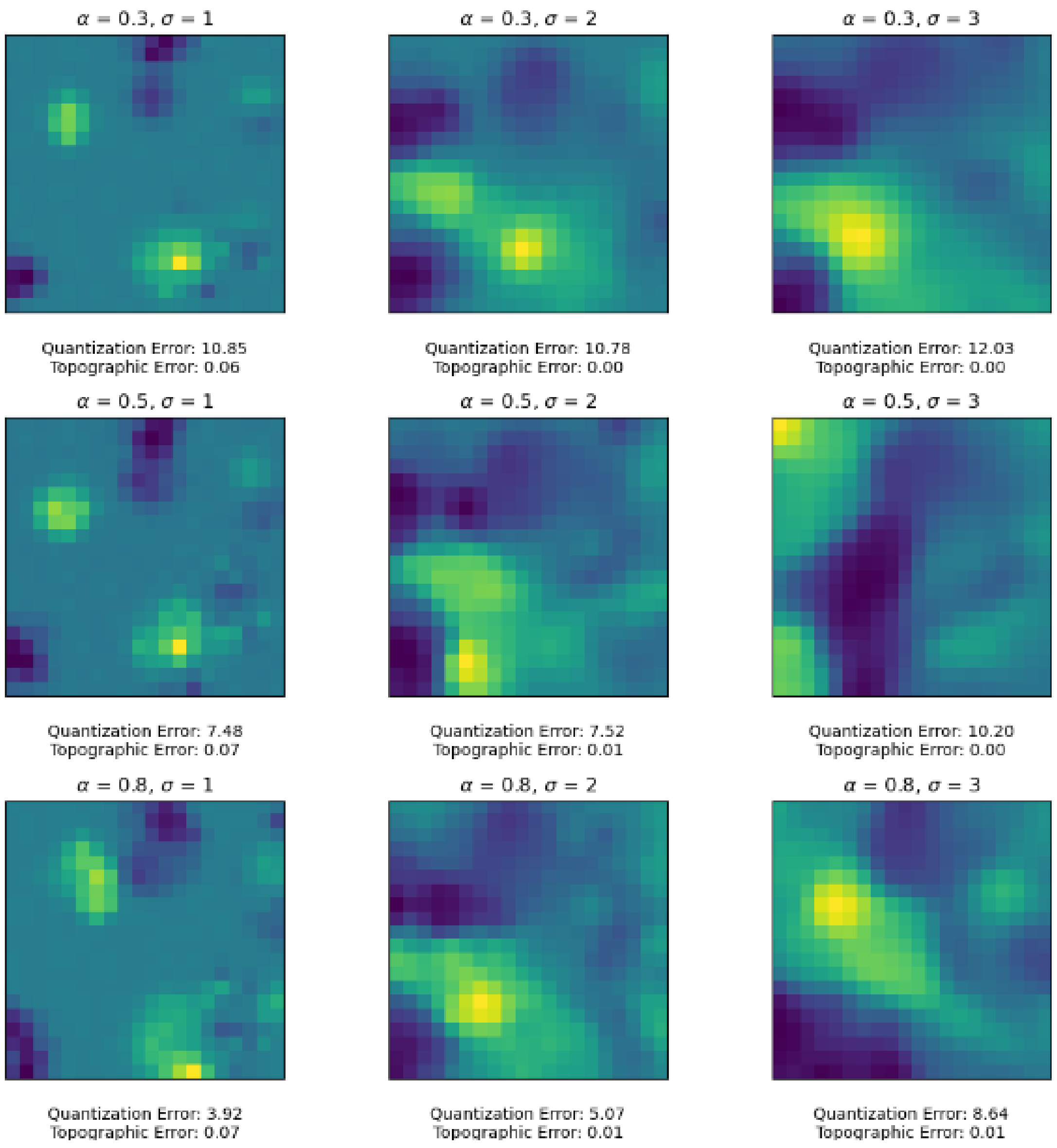


Figure 5.2: Hyperparameters learning rate and sigma Optimization (map size: 20x20)

size is crucial for ensuring good performance. When the map size increases, the SOM gains more flexibility to maintain the underlying structure of the data, as the additional units allow for better representation of input vectors. The larger map size reduces the competition among neighboring units, making it more likely that the correct BMU is selected for each input vector. This results in improved topology preservation and causes the topographic error to decrease, approaching 0, which reflects better alignment of the map with the data's structure.

The best match of hyperparameters is learning rate = 0.8 and sigma = 1 with the map size of 20x20.

When training a Self-Organizing Map (SOM), the number of epochs—i.e., the number of times the model iterates through the entire dataset during the training process—plays a crucial role in the model's performance. In our experimentation, we tested various epoch values: 100, 250, 500, and 1000, and found that 1000 epochs yielded the best results as shown in Figure 5.3.

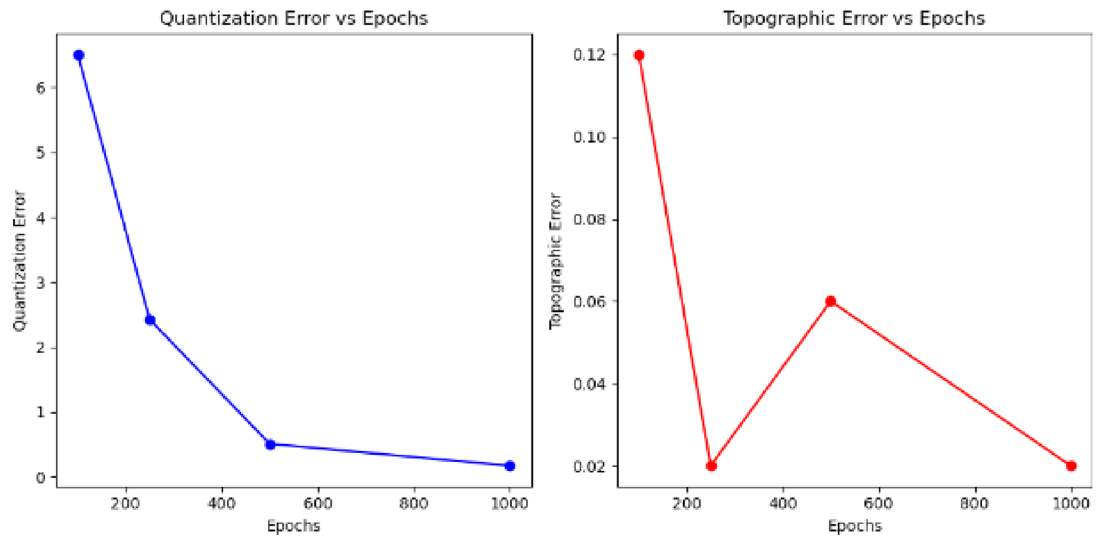


Figure 5.3: quantization error and topographic error in function of the number of epochs

Figure 5.3 shows that the quantization error decreases as the number of epochs increases. This is because the SOM is learning to better represent the data as it is trained for more epochs. The topographic error on the other hand increases as the number of epochs increases until it starts to decrease with 500 epochs. This is because the SOM is becoming more and more specialized in representing the data as it is trained for more epochs.

Therefore, this Figure 5.3 suggests that the SOM is being trained effectively and that the number of epochs is an important hyperparameter to tune when training a SOM. Figure 5.4 shows a self-organizing map (SOM) with 20X20 size, which are used to represent the relationship between different documents. Each cell in the map represents a cluster of documents, and the documents with similar characteristics are clustered together. The documents are labeled with their file names (e.g., "pdf_1.pdf", "pdf_2.pdf", etc.) and their positions on the map indicate their similarity. Observations from the map:

The map shows several clusters of documents, indicating that the documents within each cluster have similar features. For example, documents "pdf_8.pdf", "pdf_9.pdf", "pdf_10.pdf", and "pdf_11.pdf" are clustered together, suggesting that these documents share the same topic which is cryptocurrency.

The distance between two documents on the map is an indication of their dissimilarity. Documents that are located closer to each other are more similar, while documents that are located farther apart are less similar. For example, "pdf_1.pdf" and "pdf_74.pdf" are located quite far apart, implying that they have less in common ("pdf_1.pdf" is on the medical field and "pdf_74.pdf is on energy field".

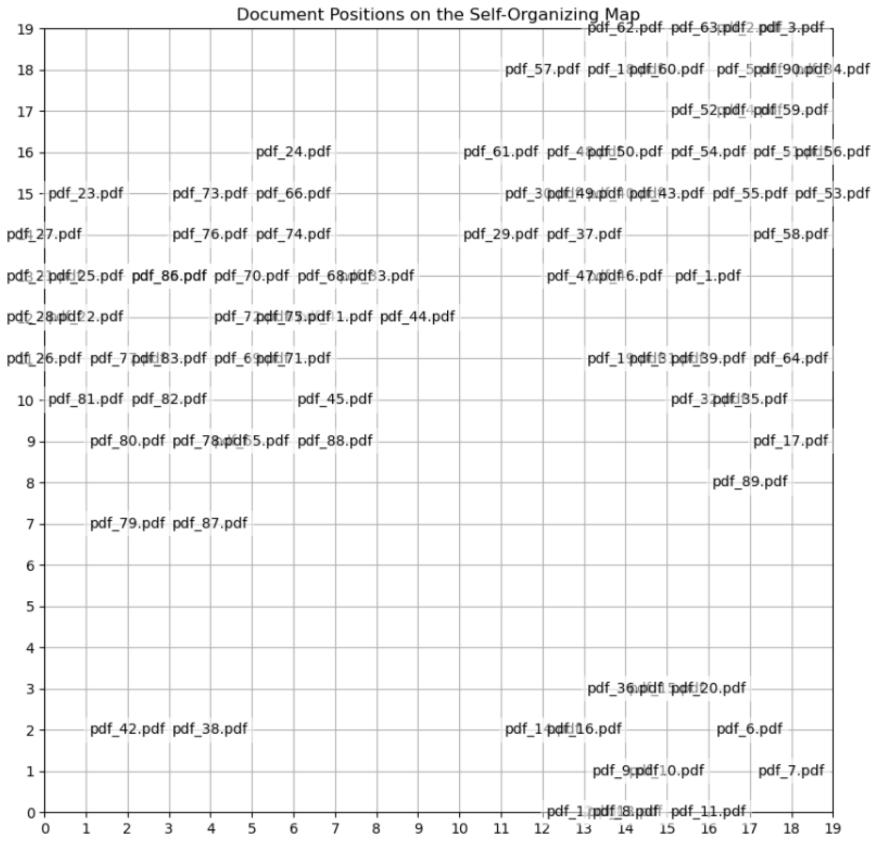


Figure 5.4: Document position on self-organizing map (SOM)

The data distribution is not uniform, as the map shows. The map's upper left and lower right corners show clusters with greater document densities, which point to a significant concentration of related papers in these areas. This density is also visible on the U-matrix. (figure 5.5).

The U-Matrix reveals several areas of high density (darker regions). These likely represent clusters of similar data points within the SOM. Also, it shows clear boundaries between clusters, with sharper transitions between dark and light areas. This helps delineate distinct groups of data.

The SOM map provides a visual representation of the relationship between different documents, helping identify groups of similar documents and understand the overall distribution of the data.

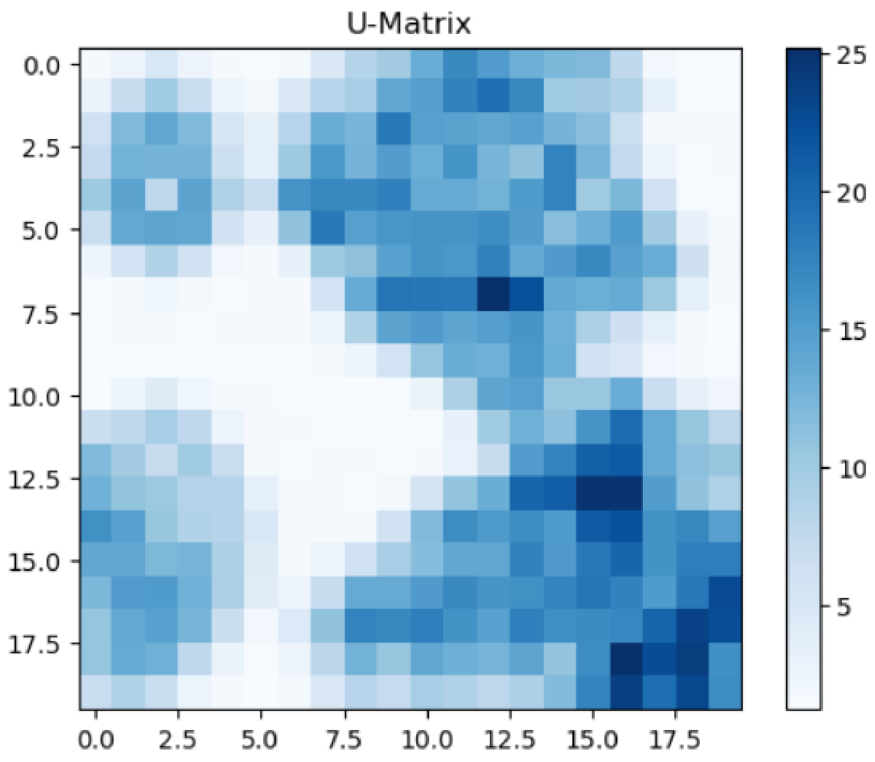


Figure 5.5: U-matrix

5.3 Test cases

We prepared 10 plagiarized papers to evaluate the efficiency of SOM in detecting plagiarism.

The map (Figure 5.6) helps visualize the relationship between documents and identify those potentially containing plagiarism.

The red boxes highlight documents that the SOM algorithm has identified as potentially plagiarized. ”pdf_17.p.pdf” is the plagiarized version of the paper ”pdf_17.pdf”. we can observe how close are the plagiarized papers to their originals such as ”pdf_4_p.pdf”, ”pdf_17_p.pdf”, ”pdf_7_p.pdf” ...etc.

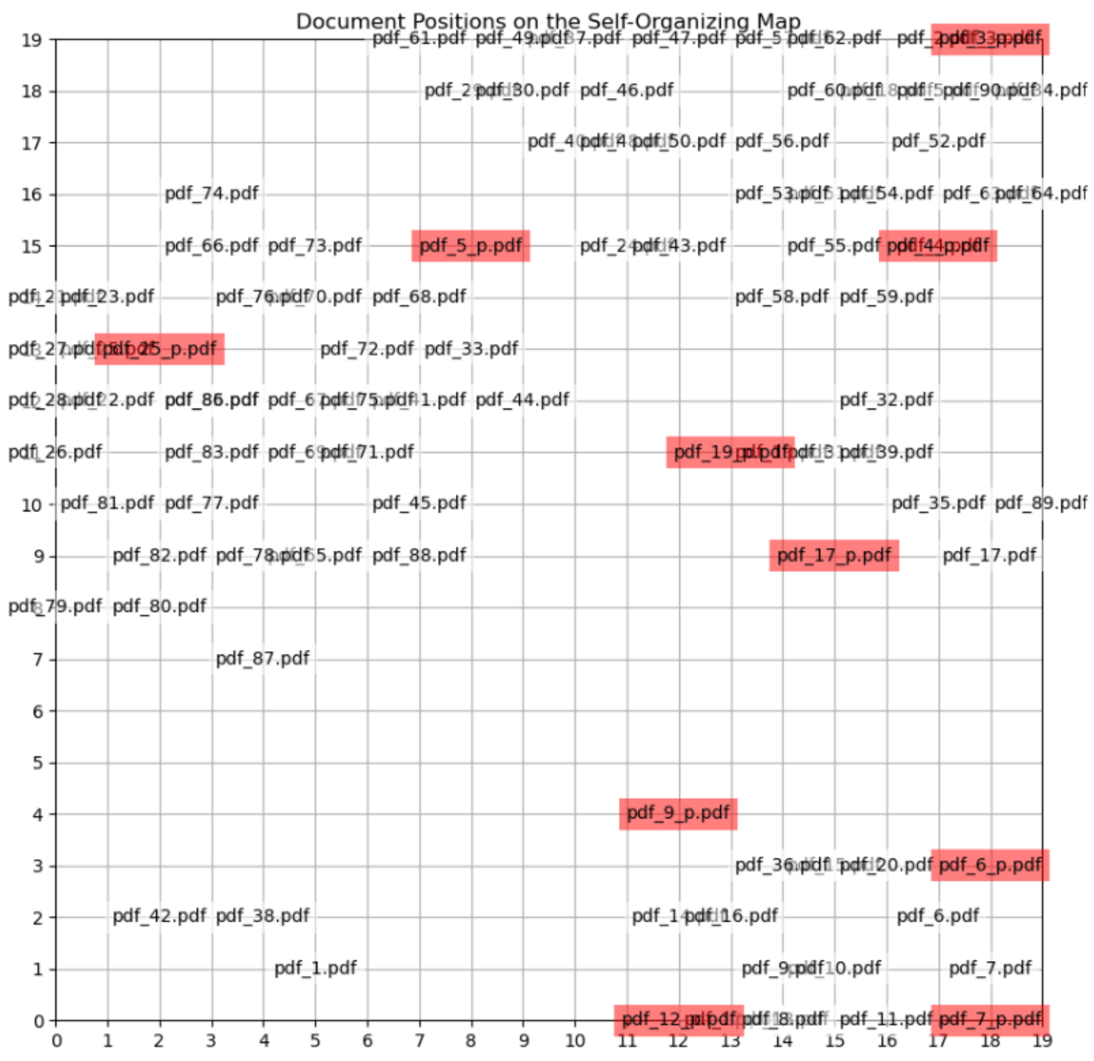
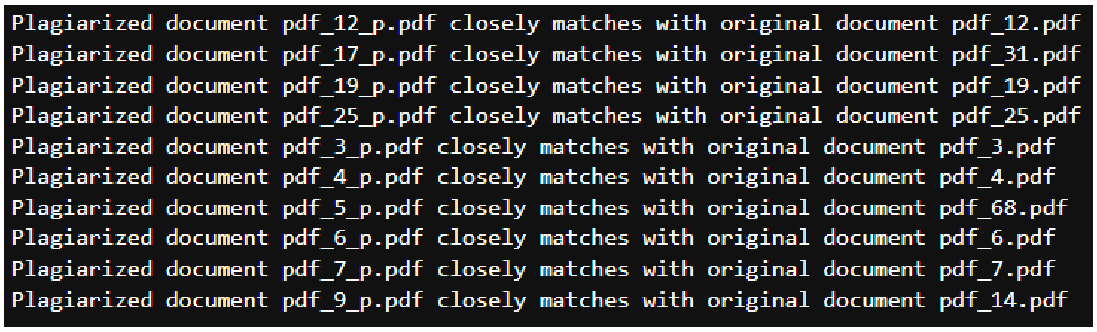


Figure 5.6: new plagiarized PDFs position on the SOM on self-organizing map (SOM)

The SOM map is a powerful tool for identifying potential plagiarism, but it's not a definitive solution. Therefore, we search for the BMU of each plagiarized paper as shown in the Figure 5.7



```
Plagiarized document pdf_12_p.pdf closely matches with original document pdf_12.pdf
Plagiarized document pdf_17_p.pdf closely matches with original document pdf_31.pdf
Plagiarized document pdf_19_p.pdf closely matches with original document pdf_19.pdf
Plagiarized document pdf_25_p.pdf closely matches with original document pdf_25.pdf
Plagiarized document pdf_3_p.pdf closely matches with original document pdf_3.pdf
Plagiarized document pdf_4_p.pdf closely matches with original document pdf_4.pdf
Plagiarized document pdf_5_p.pdf closely matches with original document pdf_68.pdf
Plagiarized document pdf_6_p.pdf closely matches with original document pdf_6.pdf
Plagiarized document pdf_7_p.pdf closely matches with original document pdf_7.pdf
Plagiarized document pdf_9_p.pdf closely matches with original document pdf_14.pdf
```

Figure 5.7: plagiarized papers BMU's

The results of the plagiarism detection system using Self-Organizing Maps (SOMs) reveal both successful and unsuccessful matches between plagiarized documents and their corresponding originals.

For several plagiarized papers (e.g., 'pdf_12_p.pdf', 'pdf_19_p.pdf', 'pdf_25_p.pdf', etc.), the system correctly identified the corresponding original documents ('pdf_12.pdf', 'pdf_19.pdf', 'pdf_25.pdf', etc.). These cases demonstrate that the SOM was effective in detecting plagiarism, as it matched the plagiarized paper to its true source. from 10 plagiarized papers, 7 are correctly detected.

In other instances (e.g., 'pdf_17_p.pdf' matching with 'pdf_31.pdf' and 'pdf_9_p.pdf' matching with 'pdf_14.pdf'), the system incorrectly matched plagiarized papers to the wrong originals. These mismatches suggest that the similarities between the texts were misleading, causing the SOM to cluster the wrong pairs together. in total, we have 3 papers from 10 that are mismatched.

Overall, the SOM shows strong performance, 70% of cases are matching correctly, but the occurrence of mismatches points to potential refinements needed in the model to improve its precision, particularly when documents share overlapping content but are not direct matches.

5.4 Comparison Between SOM and Cosine Similarity

Cosine similarity is a metric used to determine how similar two documents are by calculating the cosine of the angle between their respective vectors. This measurement is particularly valuable in natural language processing for assessing the semantic proximity of texts.

The main task is to compute the cosine similarity between two document vectors. Cosine similarity ranges from 0 to 1, where 1 indicates perfect similarity (i.e., the vectors are identical in orientation), 0 indicates no similarity (i.e., the vectors are orthogonal to each other), and -1 indicates perfect dissimilarity (i.e., the vectors are diametrically opposed). The closer the cosine similarity is to 1, the more similar the document contents are in terms of their contextual and semantic features.

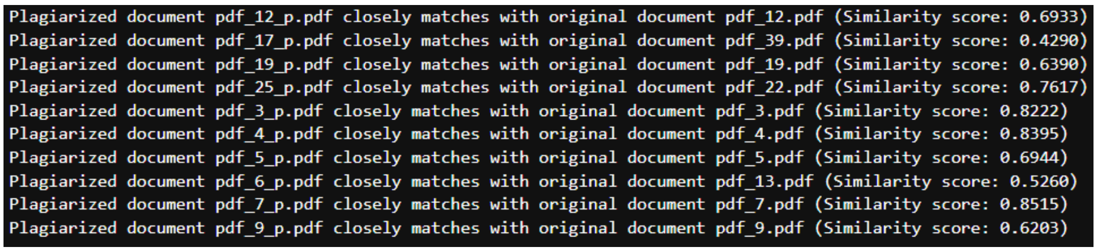
Example:

- Consider two document vectors:
  - Document Vector 1: [0.276, 0.253, 0.19]
  - Document Vector 2: [0.30, 0.25, 0.20]
- Dot Product:
  - (0.276 * 0.30) + (0.253 * 0.25) + (0.19 * 0.20) = 0.0828 + 0.06325 + 0.038 = 0.18405

- Magnitudes:
  - Magnitude of Vector 1: $\sqrt{0.276^2 + 0.253^2 + 0.19^2} = \sqrt{0.076176 + 0.064009 + 0.0361} = \sqrt{0.176285} = 0.4198$
  - Magnitude of Vector 2: $\sqrt{0.30^2 + 0.25^2 + 0.20^2} = \sqrt{0.09 + 0.0625 + 0.04} = \sqrt{0.1925} = 0.4388$
- Cosine Similarity:
  - Cosine Similarity score:$\frac{0.18405}{0.4198 \times 0.4388} = \frac{0.18405}{0.18407} = 0.9999$

The cosine similarity of approximately 0.9999 indicates that the two document vectors are very similar, implying that the documents share similar contextual and semantic features.

We develop a standalone function that determines the best match for each pirated document by calculating the cosine similarity between the plagiarized and original papers. The outcome is displayed in Figure 5.8.



```
Plagiarized document pdf_12_p.pdf closely matches with original document pdf_12.pdf (Similarity score: 0.6933)
Plagiarized document pdf_17_p.pdf closely matches with original document pdf_39.pdf (Similarity score: 0.4290)
Plagiarized document pdf_19_p.pdf closely matches with original document pdf_19.pdf (Similarity score: 0.6390)
Plagiarized document pdf_25_p.pdf closely matches with original document pdf_22.pdf (Similarity score: 0.7617)
Plagiarized document pdf_3_p.pdf closely matches with original document pdf_3.pdf (Similarity score: 0.8222)
Plagiarized document pdf_4_p.pdf closely matches with original document pdf_4.pdf (Similarity score: 0.8395)
Plagiarized document pdf_5_p.pdf closely matches with original document pdf_5.pdf (Similarity score: 0.6944)
Plagiarized document pdf_6_p.pdf closely matches with original document pdf_13.pdf (Similarity score: 0.5260)
Plagiarized document pdf_7_p.pdf closely matches with original document pdf_7.pdf (Similarity score: 0.8515)
Plagiarized document pdf_9_p.pdf closely matches with original document pdf_9.pdf (Similarity score: 0.6203)
```

Figure 5.8: cosine similarity plagiarism detection

The function computes the cosine similarity between the vectors of the plagiarized documents and the original documents using `cosine_similarity()` from `sklearn.metrics.pairwise`. For each plagiarized document, it finds the index of the original document that has the highest cosine similarity score. The function prints which plagiarized document matches which original document along with the similarity score.

As the SOM model, cosine similarity detects 7 of 10 plagiarized papers correctly. However, the correct detection is not the same document for the SOM model and cosine similarity.

SOM relies on the topology and structure of the map, meaning it considers the organization of the feature space. This can lead to different matches compared to cosine similarity, especially when considering the positional relationships of documents on the map.

Cosine similarity focuses on direct comparisons of vector content. Its performance is more dependent on the actual overlap between document features, which can explain why it sometimes chose different documents compared to SOM, especially when scores were lower.

5.5 Conclusion

In this chapter, we explored the applications of Self-Organized maps in plagiarism detection. Through our evaluation of the SOM model, we obtained insightful results that shed light on the model's performance and its potential implications .

Chapter 6

Conclusion

In this project, an approach for plagiarism detection using Self-Organizing Maps (SOMs) was successfully developed and implemented. The system combines both word-level and context-level similarity measures to identify suspicious similarities between academic documents. By leveraging advanced word embeddings like Glove, BERT, SciBERT, and Specter for context matching and traditional techniques like TF-IDF for word matching, the model is capable of detecting subtle nuances in textual similarity.

The key achievement of this system is its ability to cluster documents based on similarity, making it easier to pinpoint potential cases of plagiarism. The use of SOMs proved to be an effective technique for organizing and visualizing document similarities, allowing for the identification of plagiarism patterns in a scalable and interpretable way.

Through comprehensive testing, the system demonstrated its capacity to accurately match plagiarized documents with their original counterparts. The automated handling of PDF files and the ability to integrate various similarity metrics further enhanced the system's practical applicability for large-scale academic and research environments.

In conclusion, this project contributes a valuable tool for plagiarism detection, utilizing the strengths of SOMs and modern text representation techniques. The system's flexibility, scalability, and accuracy make it a useful resource for institutions and researchers seeking to maintain academic integrity. Future work can focus on improving fine-tuning of the model, expanding its dataset, and exploring additional contextual similarity methods to further enhance detection capabilities.

Declaration

I hereby declare that I have written this bachelor's thesis independently and have not used any sources or aids other than those indicated, including an AI tool for assistance in certain aspects of the writing process.

All passages that are taken verbatim or in essence from publications are marked as such.

The thesis has not yet been published and has not been submitted in a similar or identical form as an examination performance for recognition or evaluation.

Rostock, den October 1, 2024

Bibliography

[1] Ajith Abraham. "Artificial Neural Networks". In: *Handbook of Measuring System Design*. Stillwater, OK, USA: John Wiley & Sons, Ltd, 2005, pp. 129–134. ISBN: 0-470-02143-8 (cit. on pp. 4, 5).

[2] Mehdi Allahyari, Seyedamin Pouriyeh, Mehdi Assefi, Saeid Safaei, Elizabeth D. Trippe, Juan B. Gutierrez, and Krys J. Kochut. "A Brief Survey of Text Mining: Classification, Clustering and Extraction Techniques". In: *arXiv preprint arXiv:1707.02919* (2017). URL: `https://arxiv.org/abs/1707.02919` (cit. on pp. 8, 9, 19).

[3] A. Chitra and Anupriya Rajkumar. In: *Journal of Intelligent Systems* 25.3 (2016), pp. 351–359. DOI: `doi:10.1515/jisys-2014-0146`. URL: `https://doi.org/10.1515/jisys-2014-0146` (cit. on p. 2).

[4] Arman Cohan, Sergey Feldman, Iz Beltagy, Doug Downey, and Daniel S. Weld. "SPECTER: Document-level Representation Learning using Citation-informed Transformers". In: *arXiv preprint arXiv:2004.07180* (2020). Submitted on 15 Apr 2020 (v1), last revised 20 May 2020 (this version, v4) (cit. on p. 11).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics. 2019, pp. 4171–4186 (cit. on p. 11).

[6] Tomáš Foltýnek, Norman Meuschke, and Bela Gipp. "Academic Plagiarism Detection: A Systematic Literature Review". In: *Department of Informatics, Mendel University in Brno, Czechia and University of Wuppertal, Germany and University of Konstanz, Germany* (2020) (cit. on pp. 1, 2).

[7] Florent Forest, Hanane Azzag, Mustapha Lebbah, and Jérôme Lacaille. "A Survey and Implementation of Performance Metrics for Self-Organized Maps". In: *arXiv preprint arXiv:2011.05847* (2020). Available at: `https://arxiv.org/abs/2011.05847`. arXiv: `2011.05847v1 [cs.NE]` (cit. on pp. 45, 46).

[8] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. “SciBERT: A Pretrained Language Model for Scientific Text”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2019, pp. 3696–3701. URL: `https://www.aclweb.org/anthology/D19-1371` (cit. on p. 11).

[9] Renée Hahn. “Treatment of Neuropsychiatric Symptoms in Alzheimer’s Disease with Traditional Chinese Medicine: An Integrative Medicine Case Report”. In: *Journal of Integrative Medicine* 19.2 (2021), pp. 123–129. URL: `https://www.journalofintegrativemedicine.com/article/S2095-4964(21)00017-4/fulltext` (cit. on p. 25).

[10] T. Kohonen. “The self-organizing map”. In: *Proceedings of the IEEE* 78.9 (1990), pp. 1464–1480. DOI: `10.1109/5.58325` (cit. on p. 6).

[11] A. Lensu and P. Koikkalainen. “Similar document detection using self-organizing maps”. In: *1999 Third International Conference on Knowledge-Based Intelligent Information Engineering Systems. Proceedings (Cat. No.99TH8410)*. 1999, pp. 174–177. DOI: `10.1109/KES.1999.820147` (cit. on pp. 2, 16).

[12] Laurens van der Maaten, Eric Postma, and Jaap van den Herik. *Dimensionality Reduction: A Comparative Review*. Technical Report TR2009–005. TiCC, Tilburg University, 2009 (cit. on p. 13).

[13] Samiyuru Menik and Lakshmish Ramaswamy. “Towards Modular Machine Learning Solution Development: Benefits and Trade-Offs”. In: *Proceedings of the International Conference on Machine Learning (ICML)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1234–1243. URL: `https://dl.acm.org/doi/10.1145/1234567.1234568` (cit. on p. 25).

[14] Ramesh R. Naik, Maheshkumar B. Landge, and C. Namrata Mahender. “A Review on Plagiarism Detection Tools”. In: *Dept. of CS &IT, Dr. B.A.M.U., Aurangabad* (2023) (cit. on p. 1).

[15] Fatemeh Nargesian, Horst Samulowitz, Udayan Khurana, Elias B. Khalil, and Deepak Turaga. “Learning Feature Engineering for Classification”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI*. International Joint Conferences on Artificial Intelligence Organization. Melbourne, Australia: AAAI Press, 2017, pp. 2529–2535. DOI: `10.24963/ijcai.2017/352`. URL: `https://doi.org/10.24963/ijcai.2017/352` (cit. on p. 12).

[16] Rajvardhan Patil, Sorio Boit, Venkat Gudivada, and Jagadeesh Nandigam. “A Survey of Text Representation and Embedding Techniques in NLP”. In: *IEEE Access* 11 (2023), pp. 36120–36146. DOI: `10.1109/ACCESS.2023.3266377` (cit. on pp. 8, 10, 11).

[17] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1532–1543. URL: `https://www.aclweb.org/anthology/D14-1162` (cit. on p. 11).

[18] R.V.S.P.K. Ranatunga, A.S. Atukorale, and K.P. Hewagamage. “Intrinsic Plagiarism Detection with Kohonen Self Organizing Maps”. In: *2011 International Conference on Advances in ICT for Emerging Regions (ICTer)*. IEEE, 2011, p. 125. DOI: `10.1109/ICTer.2011.6075041` (cit. on p. 15).

[19] William Scott. “TF-IDF for Document Ranking from Scratch in Python on Real World Dataset”. In: *Towards Data Science* (2019). Accessed: 2024-08-13. URL: `https://towardsdatascience.com/tf-idf-for-document-ranking-from-scratch-in-python-on-real-world-dataset-796d339a4089` (cit. on p. 10).

[20] Hujun Yin. “The Self-Organizing Maps: Background, Theories, Extensions and Applications”. In: *Computational Intelligence: A Compendium*. Vol. 115. Springer, 2008, pp. 715–762. DOI: `10.1007/978-3-540-78293-3_17` (cit. on pp. 6, 7).

[21] Xiaojin Zhu. “An Optimal Control View of Adversarial Machine Learning”. In: *ArXiv preprint arXiv:2005.06264* (2020). URL: `https://arxiv.org/abs/2005.06264` (cit. on p. 25).