Trabajo Fin de Grado
Grado en Ingeniería Aeroespacial

Automated paper clustering for conferences: A natural language processing approach

Autor: Ana Sánchez Periñán

Tutor: Antonio Luque Estepa

Dpto. Ingeniería Aeroespacial y Mecánica de Fluidos
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado

Grado en Ingeniería Aeroespacial

Automated paper clustering for conferences: A natural language processing approach

Autor:

Ana Sánchez Periñán

Tutor:

Antonio Luque Estepa

Profesor Titular

Dpto. Ingeniería Aeroespacial y Mecánica de Fluidos

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Grado: Automated paper clustering for conferences: A natural language processing approach

Autor: Ana Sánchez Periñán
Tutor: Antonio Luque Estepa

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A Anabel: "Te mereces la paz de haber terminado". A mi madre por rezar por mí a diario. A Chali :) A mi padre y a mis hermanos por confiar en que lo conseguiría. A Ali y María por ser lo único bueno (bueno es poco) que saco de este infierno. A Wuolah y al Deep Learning. A Antonio por darme la oportunidad de hacer esto y por su gran paciencia y ayuda. A los Carrus por acompañarme a la sala Brake. Gracias a todos vosotros no estoy en el paro.

Resumen Global

Título: Automated paper clustering for conferences: A natural language processing approach

Resumen del trabajo:

Este proyecto tiene como objetivo utilizar técnicas de procesamiento de lenguaje natural y aprendizaje automático para agrupar documentos de una conferencia por similitud temática. Se explorarán diferentes algoritmos de clustering y se aplicará un enfoque basado en la representación vectorial de los textos utilizando la frecuencia de términos y el esquema TF-IDF. Además, se discutirán los resultados obtenidos y las posibles limitaciones y futuras líneas de investigación.

Palabras clave:

Asignación de ponentes, NLP, Aprendizaje automático, Similitud temática, IECON.

Conclusiones:

El proyecto mejora la coherencia temática en las salas de conferencias de IECON mediante la aplicación de técnicas de agrupamiento. Se destacan avances significativos, como una exploración perspicaz de los hiperparámetros, la exitosa implementación de K-means y un fortalecimiento general que aporta considerablemente a la robustez del sistema.

Resumen

Este proyecto presenta un sistema automatizado diseñado para mejorar la asignación de ponentes en las conferencias IECON, la Conferencia Internacional de Electrónica Industrial. Con el objetivo de elevar tanto la experiencia de los asistentes como la eficiencia para los organizadores, el sistema utiliza avanzadas técnicas de procesamiento del lenguaje natural y aprendizaje automático. Estas técnicas permiten agrupar a los ponentes según la similitud temática de sus investigaciones, fomentando así una mayor coherencia y fluidez en las sesiones conferenciales de este destacado evento internacional. Esto, a su vez, contribuye a una gestión más eficiente de los proyectos IECON en términos de asignación temática de ponentes.

La metodología del sistema se basa en la extracción de características relevantes de las tesis de los ponentes, las cuales son utilizadas para entrenar un modelo de aprendizaje automático encargado de llevar a cabo la agrupación. La evaluación del modelo se ha realizado mediante comparaciones con métodos tradicionales de asignación de ponentes, asegurando la eficacia y superioridad del enfoque propuesto, específicamente en el contexto de las conferencias IECON.

Abstract

This project introduces an automated system designed to enhance speaker allocation in IECON conferences, the International Conference on Industrial Electronics. With the aim of improving both attendee experience and efficiency for organizers, the system employs advanced natural language processing and machine learning techniques. These methods facilitate the grouping of speakers based on the thematic similarity of their research, fostering greater coherence and fluidity in the conference sessions of this prominent international event. This, in turn, contributes to more efficient management of IECON projects in terms of speaker thematic allocation.

The system's methodology relies on extracting relevant features from the papers of the speakers, which are used to train a machine learning model responsible for the grouping process. The model's effectiveness has been evaluated through comparisons with traditional speaker allocation methods, ensuring the efficiency and superiority of the proposed approach, specifically within the context of IECON conferences.

List of Figures

List of Tables

Contents

1 Introduction

In this chapter, we provide a concise overview of the objectives and content outlined in this document. Initially, Sections 1.1 and 1.2 delve into the motivation and goals of this undertaking. Subsequently, Section 1.3 delineates the document's organization and structure.

1.1 Context and Motivation

Conference organizers face a significant challenge in manually assigning speakers to different sessions. This process involves manually grouping speakers who discuss related topics in adjacent time slots into the same session. The thematic content of speakers' presentations is closely linked to their respective research papers.

This project addresses the manual nature of this allocation process by proposing an automated and more efficient solution. The goal is to cluster speakers based on the similarity of their research papers. By doing so, the thematic coherence within each session can be significantly improved, ensuring a more seamless and meaningful flow of topics throughout the conference.

This leads to the central problem tackled by this project: the automated grouping of texts according to their thematic similarity. Currently, the manual effort invested in this task is substantial, and this research aims to provide a machine learning-based solution leveraging Natural Language Processing (NLP) techniques.

The rationale behind this study is evident: to enhance and automate the speaker allocation process during conferences, with a specific focus on IECON. IECON stands out as the premier annual conference hosted by the IEEE Industrial Electronics Society (IES), addressing contemporary industry themes spanning electronics, controls, manufacturing, communications, and computational intelligence. The conference serves as a global platform for scientists and practicing engineers to share their latest research findings and ideas in the field of Industrial Electronics, emphasizing potential contributions to sustainable development and environmental preservation[1]. By constructing a robust machine learning model, this research aims to optimize conference organization, resulting in significant time and resource savings. The ultimate goal is to create a more engaging and cohesive conference experience for all attendees.

1.2 Objectives and Approach

The fundamental goals of this project center on the automation of thematic clustering for conference speakers at IECON events. These objectives can be outlined as follows:

- **Automated Text Clustering System**: Develop a robust and reliable methodology for automated text clustering specifically tailored to the unique characteristics of IECON conferences. This entails creating a system capable of effectively grouping speakers based on thematic similarities in their presented content.

- **Integration of Natural Language Processing (NLP)**: Utilize NLP techniques to ensure an accurate representation of thematic content within the automated clustering system. This integration extends throughout the entire pipeline, encompassing text preprocessing, vectorization, and similarity analysis.

- **Machine Learning**: Complement NLP with machine learning techniques to enhance the clustering model. While NLP focuses on the language-specific nuances of thematic analysis, machine learning introduces a broader framework, enabling the model to learn and adapt from the data. Specifically, machine learning will streamline the optimization of clustering algorithms, allowing the system to autonomously identify and adapt to evolving patterns in the conference speaker data.

The approach to achieving these objectives involves a synergistic collaboration between NLP and machine learning. NLP will navigate the complexities of language understanding, ensuring a sophisticated thematic analysis, while machine learning will provide adaptive learning capabilities crucial for dynamic and effective thematic clustering. This holistic approach aims to establish a comprehensive system that seamlessly integrates language-specific intricacies and adaptive learning for precise and automated thematic clustering in the context of IECON conferences.

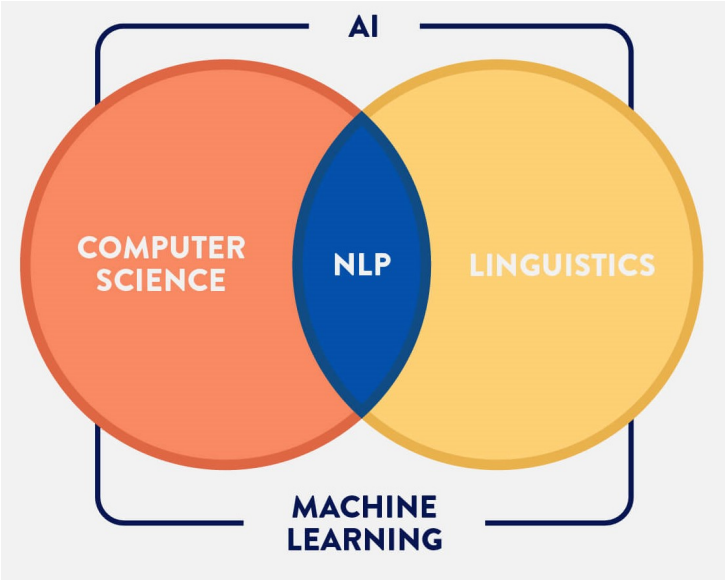

Figure 1.1 Natural Language Processing and Machine Learning [2].

1.3 Document Structure

This section serves as a roadmap for the document, outlining the content and methodologies discussed in each chapter.

- **Chapter 1: Introduction**, Provides an introduction to the document, setting the context for the research and outlining the objectives and approach.
- **Chapter 2: State of Art**, Explores the current landscape of text analysis methodologies, providing insights into the latest developments and approaches.
- **Chapter 3: Methodology**, Details the steps taken in the research process, covering data collection and preparation, papers preprocessing, feature extraction, and text representation.
- **Chapter 4: Results**, Presents the outcomes of the research, analyzing hyperparameter configurations, identifying the optimal configuration, and providing insights into clustering results.
- **Chapter 5: Conclusions**, Summarizes key findings and contributions.
- **Appendix A: User Manual**, Provides a user manual, introducing the functionality, dependencies, setup, and detailed processes for text processing, TF-IDF vectorization, and clustering.

2 State of Art

This chapter provides a comprehensive exploration of text analysis methodologies, spanning manual and computational approaches. It begins with basic tools like Venn diagrams, progressing to delve into the intricacies of Natural Language Processing (NLP). The critical stages of the NLP data pipeline, including text preprocessing and various text vectorization methods, are highlighted. The significance of text similarity in NLP and its broad applications are discussed. The chapter then transitions to machine learning, covering system classifications, training and prediction processes, hyperparameter tuning, and model evaluation. Additionally, clustering as a machine learning model is introduced, detailing its definition, objectives, and diverse algorithms. This meticulous examination unveils the complex nature of text analysis and machine learning systems, establishing a strong foundation for the project exploration.

2.1 Text Analysis Methodologies

In the expansive domain of text analysis, the quest for understanding and organizing textual information involves a spectrum of methodologies. Here, we explore various approaches categorized into manual and computational methods:

- **Manual Methods** :
  - **Venn Diagram**: An intuitive visual tool aiding in the comparison and contrast of objects, texts, events, or concepts, elucidating differences and similarities simultaneously. Venn diagrams are widely used in epidemiology to understand the overlap between different diseases. For instance, in the context of comorbidities, a Venn diagram can visually represent the common symptoms or risk factors shared among multiple diseases. [3].

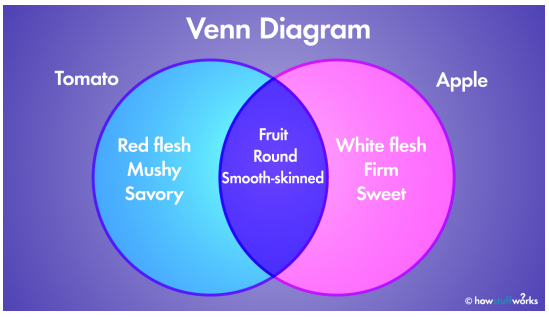

Figure 2.1 Venn Diagram [4].

  - **Taxonomies**: Hierarchical classifications grouping related concepts, instrumental in text analytics for categorizing customer mentions. Manual taxonomy creation or auto-categorization against existing taxonomies is facilitated through text analytics. Taxonomies are extensively used in biology to classify and organize living organisms. The Linnaean taxonomy, for example, categorizes species into hierarchical groups based on shared characteristics, facilitating systematic study and identification.[5].

- **Computational Methods - Natural Language Processing (NLP)**:
  - **Rule-based Approach**: Anchored in linguistic rules and patterns, this method provides a systematic analysis of text. Rule-based NLP approaches find application in chatbots for customer support. By defining rules for common queries, a chatbot can understand and respond appropriately without the need for extensive machine learning.
  - **Machine Learning Approach**: Harnessing statistical analysis, this technique empowers text processing with efficiency and scalability. Machine learning in NLP is prominently used in sentiment analysis for social media monitoring. By training models on labeled datasets, businesses can automatically analyze social media posts to gauge public sentiment about their products or services.
  - **Neural Network Approach** : Leveraging various artificial, recurrent, and convolutional neural network algorithms, this approach stands as a sophisticated avenue for text analysis. Neural network approaches in NLP, especially using models like BERT [6] or GPT [7], are employed in machine translation services. These models can understand context and nuances in language, leading to more accurate translations. [8].

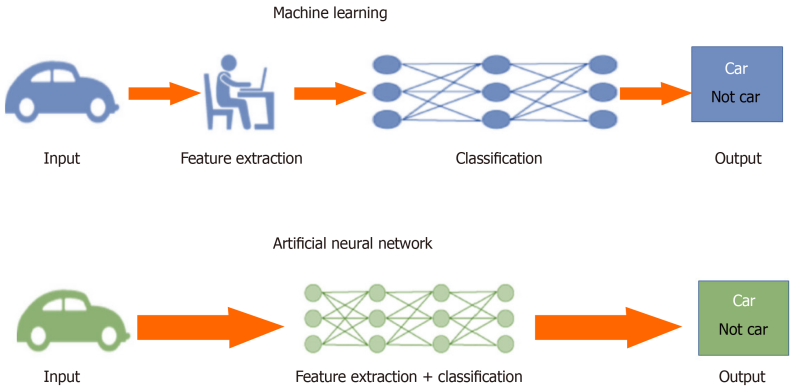

Figure 2.2 Machine Learning vs. Neural Network [9].

Within this exploration, we acknowledge that natural language processing (NLP) represents a computational facet of text analysis, coexisting with manual methodologies in the multifaceted landscape of understanding and interpreting textual information. In the exploration of text analysis methodologies for speaker grouping, the choice between manual and computational methods is crucial. In this context, the preference is for Machine Learning (ML) due to its automation capabilities, providing efficiency in the speaker clustering task. While manual methods have their merits, the dynamic nature of speaker grouping, especially in large datasets, benefits from ML's scalability and automation. Notably, a balanced approach within ML is favored, considering both rule-based and neural network methods. The decision is influenced by the potential computational costs associated with exclusive reliance on neural networks like

BERT or GPT, ensuring a pragmatic balance between computational feasibility and analytical robustness. The goal is to optimize the clustering process, making it resource-efficient while extracting meaningful insights from speaker textual information.

2.2 Natural Language Processing

Natural Language Processing (NLP) stands as a dynamic branch of artificial intelligence that revolves around instructing machines to comprehend, produce, and manipulate human language, including data structured in a human-like manner. This field is bifurcated into two key domains: natural language understanding (NLU), concentrated on deciphering the intended meaning within text, and natural language generation (NLG), focused on machine-driven text creation.

NLP has seamlessly woven itself into the fabric of daily existence, finding applications across diverse sectors such as retail, medicine, and social media. Noteworthy examples include conversational agents like Amazon's Alexa and Apple's Siri, leveraging NLP to interpret user queries and furnish responses. Similarly, GPT-4 represents a recent milestone, showcasing the ability to generate nuanced prose across an extensive spectrum of subjects. Moreover, major tech players like Google deploy NLP to enhance search results, while social platforms like Facebook utilize it for hate speech detection and filtration.

Despite the advancing sophistication within NLP, persisting challenges include the potential for bias and lapses in coherence within existing systems. Nevertheless, the continual evolution of NLP holds paramount significance for society, offering machine learning engineers myriad opportunities to integrate it into increasingly pivotal roles.[10]

The NLP pipeline involves the following stages:

- **Text Preprocessing**: Initiating with the cleaning and transformation of raw text data, this phase ensures the conversion into a format conducive to machine learning analysis. Techniques encompass noise removal, lexicon normalization, lemmatization, stemming, and object standardization.
- **Text to Features**: Transitioning from textual data to numerical vectors, this stage facilitates the utilization of machine learning models. Diverse feature extraction techniques include Syntactical Parsing, Dependency Grammar, Part of Speech Tagging, Entity Parsing, Phrase Detection, Named Entity Recognition, Topic Modeling, N-Grams, Statistical features, TF-IDF, Frequency/Density Features, Readability Features, and Word Embeddings.
- **Important NLP Tasks**: This step involves the selection of an appropriate machine learning model and its training on vectorized text data. Crucial NLP tasks encompass Text Classification, Text Matching, Levenshtein Distance, Phonetic Matching, Flexible String Matching, Coreference Resolution, Clustering and other problem-solving domains.

2.2.1 Text Preprocessing

Text preprocessing is a vital step involving the refinement of textual data before its integration into models. Textual information tends to be intricate due to the presence of diverse elements like emotions, punctuation, and case variations. In human language, conveying the same idea can take various forms, posing a challenge for machines that inherently comprehend numerical values rather than words.

The NLP preprocessing involves several steps, such as:

- **Lowercasing**: Ensuring uniformity by converting all words to lowercase.

- **Tokenization**: Breaking paragraphs into smaller units like sentences or words.
- **Punctuation Removal**: Eliminating punctuation marks from the text.
- **Stopword Removal**: Excluding common words such as "the," "is," and "and" that contribute little meaning.
- **Stemming**: Reducing words to their root form.
- **Lemmatization**: Reducing words to their base form. [11]

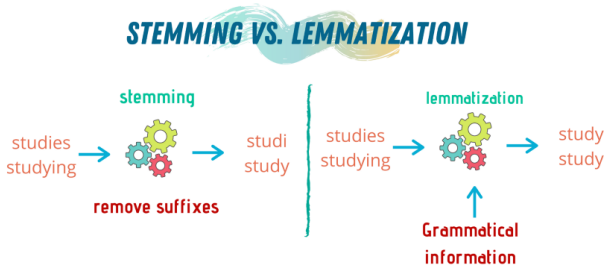

Figure 2.3 Stemming vs. Lemmatization [12].

2.2.2 Text Vectorization Techniques

Word vectorization techniques can be broadly categorized into two types: frequency-based and prediction-based.

Frequency-based Techniques

These methods represent words as vectors by considering their frequency of occurrence in a text corpus. While they are straightforward to implement and computationally efficient, capturing some semantic information, they don't delve as deeply into semantics as prediction-based techniques do [13]. Notable examples include:

- **Bag-of-Words (BoW)**: This method involves tokenizing input text, creating a vocabulary of distinct words, and constructing a sparse matrix based on the frequency of vocabulary words. Each matrix row forms a sentence vector with a length equal to the size of the vocabulary. BoW is used in document classification for organizing large sets of legal documents. By representing each document as a bag of its words, legal professionals can quickly identify relevant documents based on shared keywords.

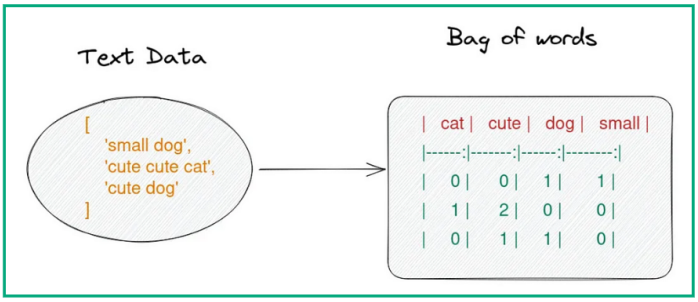

Figure 2.4 Bag of Words [14].

- **Term Frequency-Inverse Document Frequency (TF-IDF)**: Going beyond BoW, this technique calculates each word's frequency within a document and scales it by the inverse frequency of the word across all documents. The resulting matrix indicates the significance of each word in relation to each document.TF-IDF is widely used in information retrieval and document similarity. Search engines utilize TF-IDF to rank documents based on the importance of terms within them, improving the relevance of search results. [15].

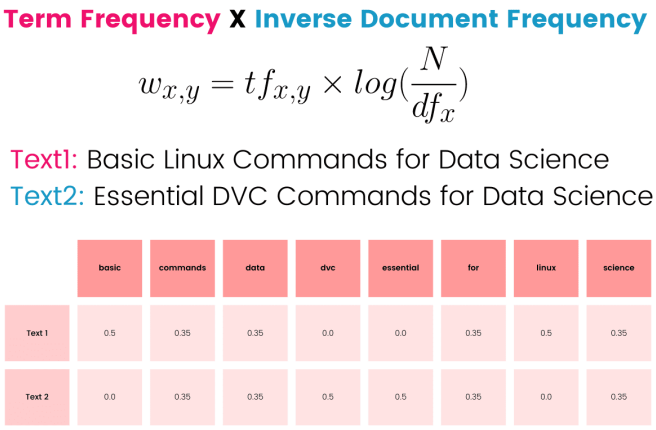

Figure 2.5 Term Frequency-Inverse Document Frequency [16].

- **Topic Models**: Including LSA, pLSA, and LDA, reveal latent variables governing document semantics. These techniques involve matrix decomposition and probability theory, capturing topics' word distributions to understand the underlying structure of documents and corpora. Topic models are applied in social media analysis for identifying prevalent themes in discussions. Analyzing topics in tweets or forum posts helps understand public opinion and trending subjects. [17]

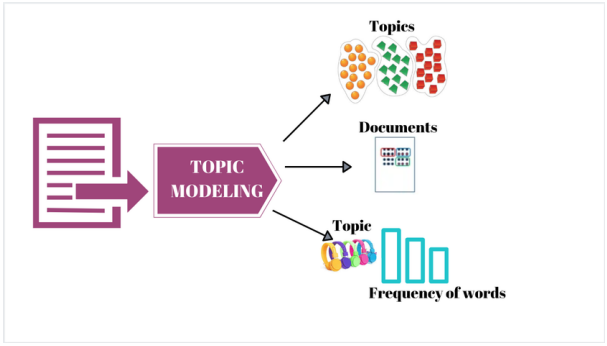

Figure 2.6 Topic Modeling [18].

Prediction-based Techniques

On the other hand, prediction-based techniques represent words as vectors based on their predictive relationships with other words in the corpus. While more complex and computationally expensive than frequency-based methods, prediction-based techniques excel at capturing richer semantic information [13]. Some examples include:

- **Word Embeddings**: This method represents words as condensed vectors in a high-dimensional space, positioning semantically similar words closer together. Word embeddings are acquired through neural networks, including models like GloVe and Word2Vec. Word embeddings, such as those produced by Word2Vec, are applied in recommendation systems for e-commerce. By understanding semantic relationships between words, these systems can suggest products based on user preferences.

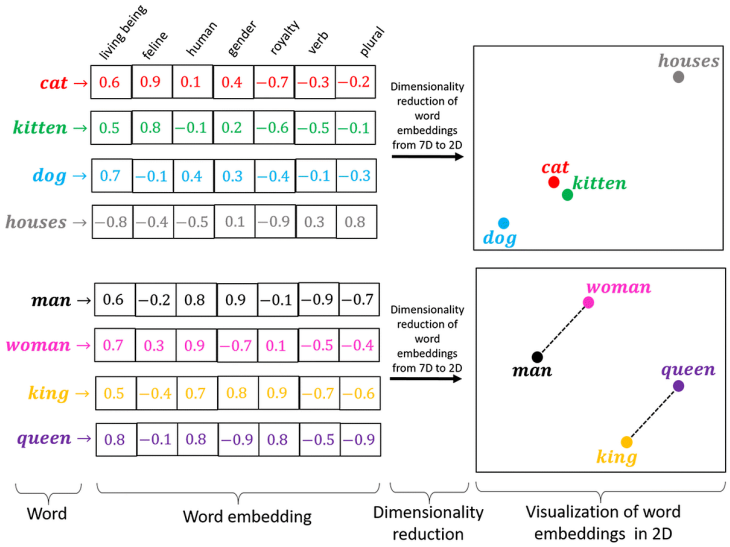

Figure 2.7 Word Embedding [19].

- **Paragraph Vectors**: This approach depicts entire paragraphs as compact vectors in a high-dimensional space, where semantically similar paragraphs are in proximity. Learning paragraph vectors can be achieved through neural networks, such as Doc2Vec. Paragraph vectors are employed in content recommendation systems, where entire articles or documents are represented as vectors. This helps recommend articles with similar content to users based on their preferences. [15].

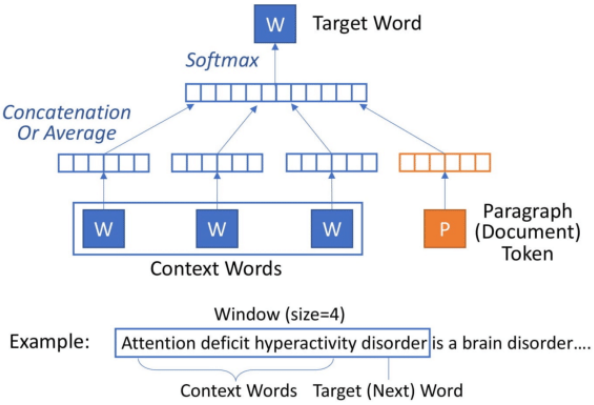

Figure 2.8 An illustration of the Paragraph Vector model with distributed memory (PV-DM) [20].

- **Transformers**: These models, which have achieved state-of-the-art performance in various NLP tasks, capture context and meaning by examining relationships in sequential data. They utilize an

evolving set of mathematical techniques known as attention or self-attention. Prominent transformer-based models include BERT, GPT, T5, RoBERTa, and ALBERT. Transformers are extensively used in chatbots for natural language understanding. Models like BERT and GPT enable chatbots to comprehend context, leading to more contextually relevant and coherent responses. [21].

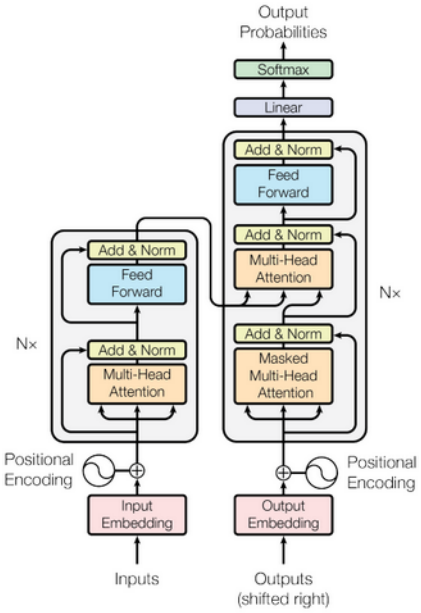

Figure 2.9 Transfomers architecture [22].

In summary, frequency-based techniques rely on the occurrence frequency of words, providing simplicity and efficiency, while prediction-based techniques focus on predictive relationships, offering more in-depth semantic understanding at the cost of increased complexity and computational resources. For text vectorization in the field of electrical and electronic engineering, frequency-based techniques are preferred over prediction-based ones due to the clarity and precision of scientific language in this domain. Prediction-based methods are deemed unnecessary for this task due to their complexity in capturing semantic nuances, the lack of specialized word dictionaries, and the high computational cost of pre-training. Bag-of-Words (BoW) is not chosen despite its simplicity and efficiency because it fails to account for word significance. Topic models, such as LSA, pLSA, and LDA, are also excluded due to their potential limitations in handling small datasets and their inability to capture the intricate topics in scientific texts. Among frequency-based techniques, TF-IDF emerges as the most suitable choice due to its ability to assign weights to words based on their importance, making it particularly well-suited for the nuanced analysis of scientific texts in electrical and electronic engineering.

2.2.3 NLP Task: Text Similarity

Text similarity, a fundamental aspect of Natural Language Processing (NLP), holds immense practical value by discerning shared meaning among textual content. This versatile tool finds widespread application in various scenarios, ranging from information retrieval to machine translation and beyond.

In information retrieval, text similarity becomes instrumental, allowing search engines to effectively rank results based on their relevance to user queries. Within NLP, it serves multifaceted purposes such as identifying synonyms, generating text analogous to a given sample, and refining machine translation

accuracy. Its adaptability extends to crucial tasks like plagiarism detection, document classification, language translation, sentiment analysis, and summarization.

Diverse algorithms contribute to the measurement of text similarity, each serving unique purposes:

- **Cosine Similarity**: Quantifying similarity based on the angle between word vectors, often employed with TF-IDF vectors, where values range from -1 to 1, indicating dissimilarity to identity. Cosine similarity is employed in plagiarism detection systems for academic and journalistic content. By comparing the cosine similarity between documents, these systems can identify potential instances of content duplication.
- **Levenshtein Distance**: Measuring the minimum edits required to transform one string into another, applicable in spell-checking. By calculating the minimum number of edits needed to transform one word into another, spell-checkers can suggest corrections for misspelled words.
- **Jaccard Index**: Gauging set similarity by comparing the intersection to the union of sets, proving beneficial for sparse or high-dimensional data. The Jaccard Index is applied in DNA sequence analysis to measure the similarity between two sets of genetic material. It helps identify common genetic elements across different species.
- **Euclidean Distance**: Calculating distance between vectors, finding application in clustering and anomaly detection.
- **Hamming Distance**: Evaluating differences between equal-length strings, with applications in error-correcting codes and cryptography. It ensures the accuracy of transmitted data by identifying and correcting errors in binary sequences.
- **Word Embeddings**: Distributed word representations capturing semantic relationships, surpassing traditional methods like one-hot encoding.
- **Pre-trained Language Models**: Models like BERT, RoBERTa, and USE learn high-quality text representations, with performance enhancements achieved through fine-tuning on specific tasks.

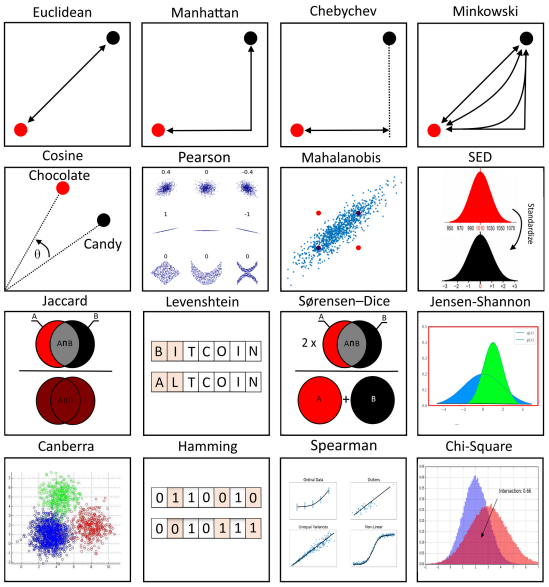

Figure 2.10 Various similarity metrics [23].

These methodologies play pivotal roles in fortifying search engines, refining machine translation systems, and advancing the overall comprehension of textual data. The broad applicability of text similarity algorithms across diverse domains underscores their significance in the expansive realm of NLP. [24]. Different techniques handle detail, clarity, and processing load differently. A robust choice is employing Cosine Similarity with TF-IDF, which excels at identifying thematic connections in scientific fields by assigning greater weight to important words. However, Euclidean and Hamming Distances are not recommended as they struggle with capturing nuanced meanings, being better suited for straightforward yes/no data. In summary, choosing Cosine Similarity with TF-IDF provides a balanced approach, effectively capturing research themes without unnecessary complexity. This aligns perfectly with the objective of streamlining conferences, such as IECON, by creating a more coherent flow of topics.

2.3 Machine Learning

Machine Learning is the discipline that involves programming computers to acquire the ability to learn from data. To illustrate, consider a spam filter—a Machine Learning program capable of learning to identify spam based on examples of spam and non-spam emails. The collection of examples used for learning is the training set, with each instance termed a training instance. In this scenario, task T is to flag spam in new emails, experience E is the training data, and performance measure P, such as accuracy, evaluates the system's ability to correctly classify emails. A computer program learns from experience E in a task T if its performance, measured by P, improves with that experience.

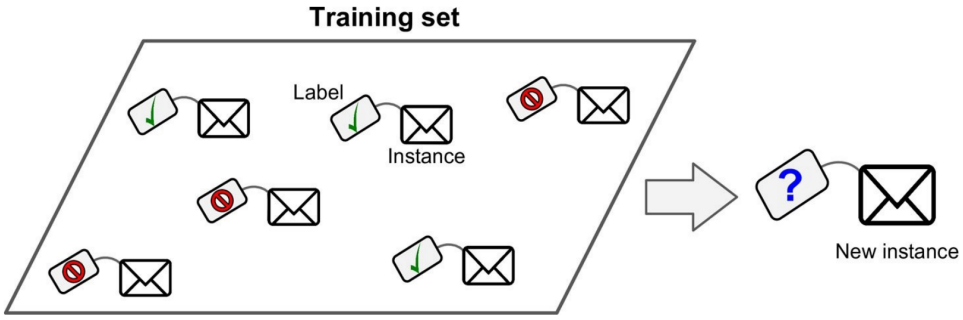

Figure 2.11 A labeled training set for supervised learning (e.g., spam classification).

Notably, merely accumulating vast amounts of data, such as downloading Wikipedia, does not enhance a computer's performance in tasks. For true Machine Learning, the system must improve its task performance based on the provided experience.

2.3.1 Classification of Machine Learning Systems

Machine Learning systems showcase a diverse range of characteristics, necessitating classification based on various critical criteria:

- **Supervision**:
  - **Supervised Learning**: Trained with human supervision using labeled data. Supervised learning is extensively used in healthcare for predicting patient outcomes. For instance, predicting the likelihood of disease recurrence after treatment based on historical patient data.

  - **Unsupervised Learning**: Operates independently of labeled data, autonomously identifying patterns and structures. Unsupervised learning is applied in market basket analysis for retail. It helps identify patterns and associations among products purchased together, informing strategies for product placement and promotions.

  - **Semi-Supervised Learning**: Utilizes a blend of labeled and unlabeled data for training. Semi-supervised learning is used in image recognition tasks, where a limited set of labeled images is augmented with a larger set of unlabeled images. This approach enhances model performance without requiring extensive labeling efforts.

  - **Reinforcement Learning**: Learns through dynamic interaction with an environment, gaining feedback through rewards or penalties. Reinforcement learning is employed in autonomous vehicle navigation. Agents learn to make decisions (e.g., steering, acceleration) based on feedback from the environment, enabling vehicles to navigate complex scenarios.

- **Learning Style**:

  - **Online Learning**: Incremental learning in real-time as new data arrives. Online learning is applied in financial fraud detection. Models can adapt in real-time to emerging patterns of fraudulent transactions, enhancing the system's ability to identify and prevent fraud.

  - **Batch Learning**: Conducts training on the entire dataset offline, without incremental updates. Batch learning is used in credit scoring. Models trained on historical batches of credit data help assess the creditworthiness of individuals applying for loans.

- **Approach to Learning**:

  - **Instance-Based Learning**: Compares new data points to existing known data points. Instance-based learning, such as k-nearest neighbors, is utilized in recommendation systems for e-commerce. Similarity between user preferences and historical data helps recommend products.

  - **Model-Based Learning**: Identifies patterns in training data, constructing predictive models. Model-based learning is applied in predictive maintenance for machinery. Models learn patterns of machinery degradation from sensor data, facilitating early detection of potential issues.

These criteria are not mutually exclusive; they can be interwoven. For instance, a cutting-edge spam filter might embody an online, model-based, supervised learning system. This system continually learns using a deep neural network model, trained with examples of both spam and non-spam (ham). This multifaceted approach underscores the adaptability and versatility inherent in the domain of Machine Learning systems. [25]

2.3.2 Training and Prediction Processes

In the realm of machine learning, model training denotes the procedure wherein a machine learning algorithm autonomously discerns patterns from data. These patterns are acquired through statistical learning, either by observing signals that indicate correct or incorrect answers (supervised learning) or by uncovering inherent data patterns without explicit knowledge of correct responses (unsupervised learning).

The culmination of the training process yields a computer program, commonly referred to as the model, endowed with the capability to make decisions and predictions on unfamiliar data. This means the model can generalize its acquired knowledge to new data it has not encountered during the training phase. [26]

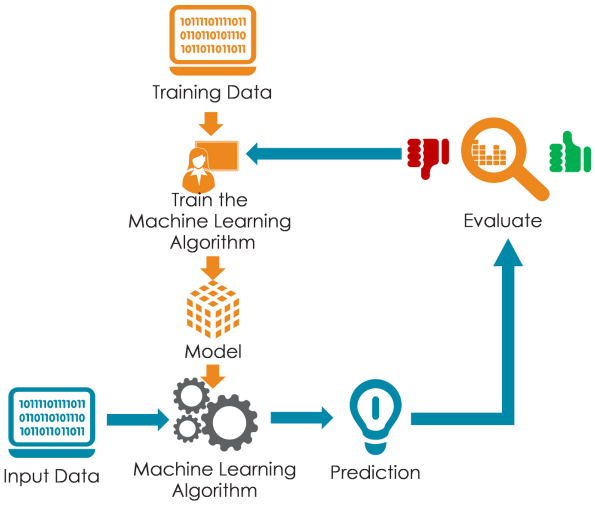

Figure 2.12 Training and validation in Machine Learning [27].

Model parameters are those values that the model calculates based on the provided data, such as the weights in a deep neural network. On the other hand, model hyperparameters are not directly estimated from the data by the model itself. Instead, they are utilized in the process of estimating model parameters. A classic example is the learning rate in deep neural networks.

Hyperparameter Tuning

Hyperparameter tuning is the process of selecting the optimal hyperparameters for a machine learning model. Hyperparameters are parameters that cannot be estimated from the data and must be set manually. They include learning rate, regularization strength, number of hidden layers, and many others. The objective in hyperparameter tuning is to discover the hyperparameter configuration that maximizes the model's performance. Hyperparameter tuning is crucial in image recognition tasks. Optimizing hyperparameters enhances the accuracy and efficiency of deep learning models for image classification.

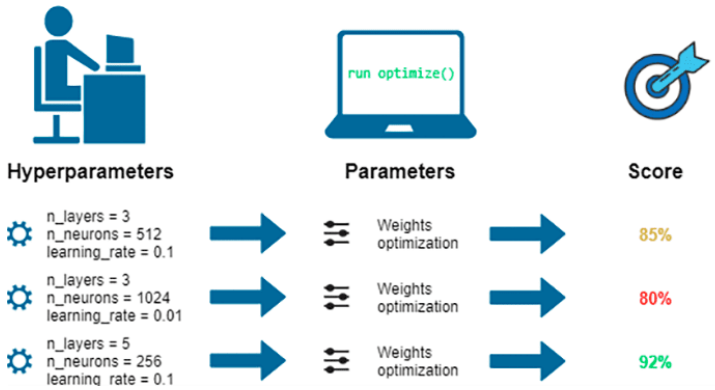

Figure 2.13 Hyperparameter tuning [28].

There's two ways to set hyperparameters:

- **Manual tuning** involves experimenting with different sets of hyperparameters manually. This technique requires a robust experiment tracker that can track a variety of variables from images, logs to system metrics.

- **Automated tuning** involves running multiple trials in a single training process. Every trial involves running your training application with specific values assigned to the chosen hyperparameters, all of which are defined within the limits you have set. This process once finished will give you the set of hyperparameter values that are best suited for the model to give optimal results. [29]

2.3.3 Model Evaluation

Model evaluation is a pivotal phase in developing machine learning models, offering insights into performance and predictive power. Various metrics, such as Accuracy, Precision, Recall, and others, contribute to this assessment. Cross Validation and Holdout represent common techniques for model evaluation.

- **Cross Validation** involves reserving a part of the dataset for testing during the training phase. K Fold Cross Validation, a prevalent type, divides the dataset into k subsets or folds, using one fold for testing and the remaining for training in each iteration. This ensures each data point serves both as a test subject and a training subject. K Fold Cross Validation is applied in financial modeling to assess the performance of predictive models for stock price movements. It helps ensure the model's robustness across different market conditions.

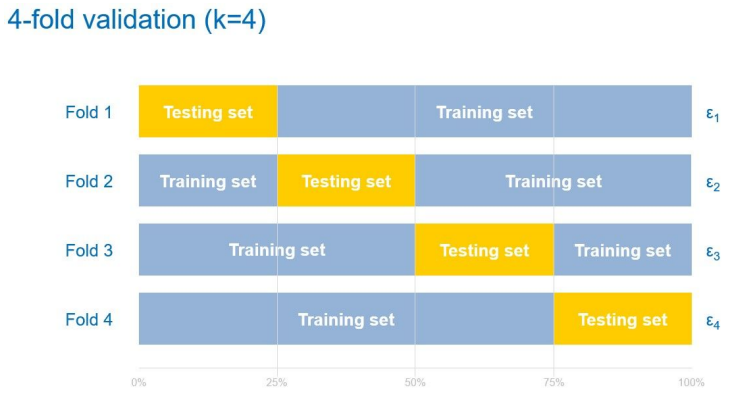

Figure 2.14 Cross Validation [30].

- **Holdout**, a simpler approach, entails dividing the dataset into training and test sets, typically in ratios like 70:30 or 80:20. A large portion of the data trains the model, while a smaller segment evaluates its performance. Holdout is used in A/B testing for website optimization. A portion of users (holdout group) is exposed to a new website version, and their interactions are compared with the control group to assess the effectiveness of changes.[31]

2.4 Clustering: a Machine Learning Model

2.4.1 Definition and Objectives

Clustering, an unsupervised machine learning technique, entails the grouping together of similar data points, eliminating the need for labeled data. Through clustering algorithms, patterns within data are discerned, and the data is systematically organized into clusters based on their shared similarities. Various clustering algorithms cater to different data characteristics, contributing to the versatility of this approach. [32]

2.4.2 Types of Clustering Algorithms

- **Centroid-based** clustering. This method arranges data into non-hierarchical clusters. Among these algorithms, k-means stands out as one of the most widely utilized centroid-based clustering techniques. Centroid-based clustering is used in customer segmentation for targeted marketing. By grouping customers based on purchasing behavior, businesses can tailor marketing strategies to specific segments.
- **Density-based** clustering presents another algorithmic approach, forming clusters by connecting areas of high example density. Notably, this algorithm accommodates arbitrary-shaped distributions, as long as dense areas can be interconnected. Density-based clustering is utilized in anomaly detection for network security. Unusual patterns in network traffic density can indicate potential security threats.
- **Distribution-based** clustering functions on the idea that data consists of various distributions, with examples including Gaussian distributions. This assumption guides the algorithm in grouping data points based on their distributional characteristics. Distribution-based clustering is applied in customer segmentation for retail. It helps identify groups of customers with similar purchasing behavior, enabling personalized marketing strategies.
- **Hierarchical** clustering, which constructs a tree-like structure of clusters. This method is particularly effective for handling hierarchical data, such as taxonomies, as it encapsulates the inherent hierarchical relationships within the data. Hierarchical clustering is employed in ecological studies to classify vegetation based on hierarchical relationships. It helps understand the structure and composition of ecosystems. [33]

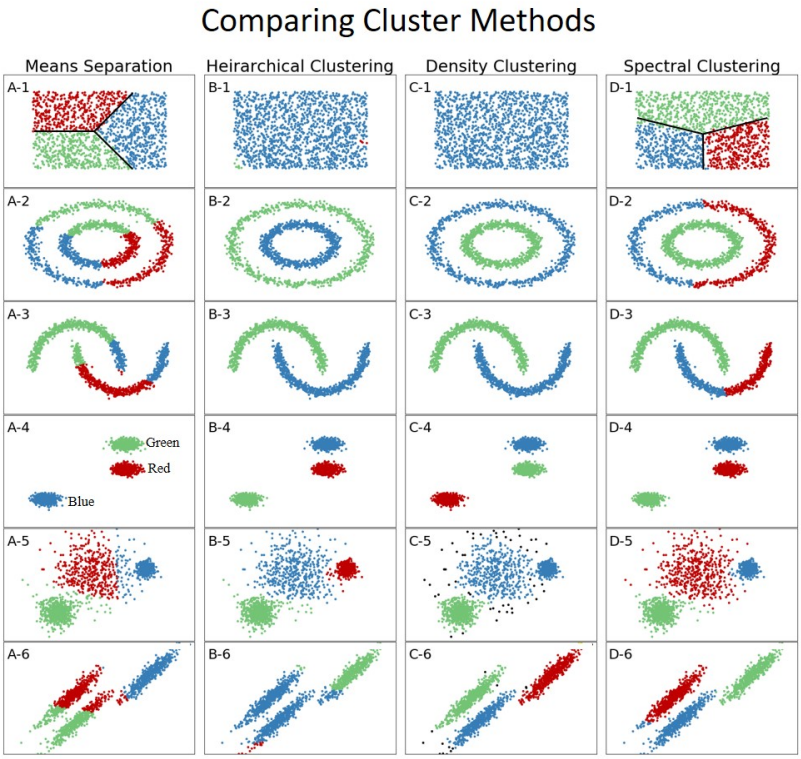

Figure 2.15 Clustering Methods [34].

In addressing the manual challenges associated with speaker allocation in conferences, this research proposes an automated solution centered on the application of centroid-based clustering, with a specific emphasis on the widely utilized k-means algorithm. The choice of centroid-based clustering aligns with the objective of grouping speakers based on the thematic similarity of their research papers. Unlike hierarchical or density-based clustering, centroid-based clustering is particularly well-suited for our scenario, as it facilitates non-hierarchical grouping, promoting a seamless and meaningful thematic flow within each conference session.

2.4.3 Clustering Evaluation

When evaluating the outcomes of clustering, the objective is to assess how well the generated clusters align with expectations. Various metrics are commonly employed for this purpose:

- **Rand Index (RI):** This metric quantifies the similarity between two clusterings by considering true positive (matching pairs) and true negative (non-matching pairs) assignments. An RI value approaching 1 indicates a close match between the clustering outcome and known assignments. For instance, in image segmentation, RI can be applied to compare the algorithm's partitioning with manually annotated ground truth.
- **Purity:** Cluster quality is evaluated by comparing clusters to external labels (known assignments). Purity measures the effectiveness of the majority class dominating each cluster, with higher purity values indicating superior clustering results. In an email clustering system, purity could ensure that each cluster predominantly contains emails from a specific topic.
- **Sum of Square Distance (SSD):** This metric evaluates the compactness of clusters by computing the sum of squared distances between data points and their respective cluster centroids. Smaller SSD values indicate more tightly knit clusters. For example, in finance, SSD can be employed for clustering stocks based on historical price movements to assess the tightness of the resulting clusters.
- **Average Silhouette Coefficient:** The Average Silhouette Coefficient measures the separation between clusters, taking into account both intra-cluster cohesion and inter-cluster separation. A higher silhouette score signifies more well-defined clusters. For instance, in social media user segmentation, this metric can be used to evaluate how well-separated user clusters are, informing tailored content recommendations. The use of the Average Silhouette Coefficient is justified as it provides a comprehensive assessment of both cluster cohesion and separation, offering a balanced perspective on clustering performance [35].

Among the clustering evaluation metrics at hand, the preference for this project leans towards the Average Silhouette Coefficient. Its efficacy lies in its capacity to harmonize two vital elements: cohesive clusters (cohesion) and clear separation between them. This well-balanced perspective offers a more precise depiction of clustering performance.

In summary, this chapter emphasizes the necessity to push the boundaries of text analysis and machine learning, laying the foundation for the forthcoming chapters. These chapters will delve into the developed methodologies and their applications, aiming to overcome the challenges associated with the manual allocation of speakers in conferences, with a specific focus on the context of IECON.

3 Methodology

In this chapter, we explore the fundamental methodology chosen for the clustering of speakers at IECON conferences. The intricacies of text analysis procedures are meticulously explained to address the challenges inherent in speaker clustering tasks. Each section serves as a guiding principle, elucidating essential elements such as data acquisition, preparation, feature extraction, and the carefully orchestrated steps involved in optimization and clustering.

The following diagram illustrates the process:

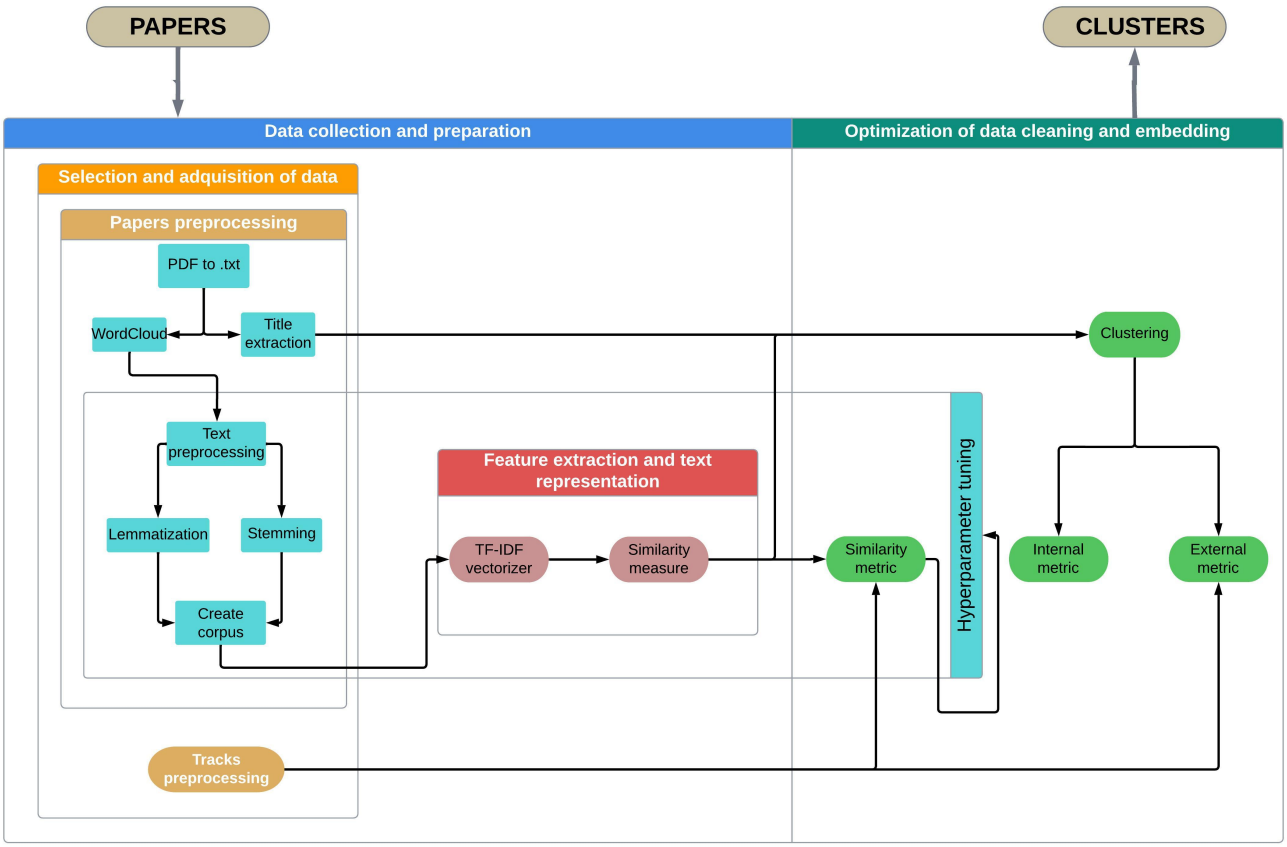

Figure 3.1 Methodology diagram.

3.1 Implementation and Tools

The proposed methodology is implemented using the Python programming language, capitalizing on its extensive ecosystem of libraries and tools designed for natural language processing (NLP) and machine learning. Python provides a versatile environment conducive to data analysis, preprocessing, and model development, thereby offering the flexibility required to achieve the objectives outlined in this project.

In the implementation of our methodology, various tools and libraries play pivotal roles. Below, we highlight the key components, including libraries and a unique interactive tool:

- **NLTK (Natural Language Toolkit)**: NLTK is a Python library that offers user-friendly interfaces to a plethora of corpora and lexical resources, including WordNet. It encompasses a suite of text processing libraries designed for tasks such as classification, tokenization, stemming, tagging, parsing, and semantic reasoning. NLTK stands as a prominent platform for constructing Python programs that handle human language data, catering to a diverse audience, including linguists, engineers, students, educators, researchers, and industry professionals. Compatible with Windows, Mac OS X, and Linux, NLTK is freely available as an open-source project driven by a collaborative community effort [36].

- **Scikit-learn**: Scikit-learn is a Python library that provides simple and efficient tools for predictive data analysis. It offers several algorithms for classification, regression, clustering, dimensionality reduction, model selection, and preprocessing. The library is open source and can be used commercially under the BSD license. [37]

- **NumPy**: NumPy is a fundamental Python library for numerical computations. It introduces a powerful N-dimensional array object, along with tools for working with these arrays. NumPy facilitates mathematical operations, array manipulation, and integration with other data science and machine learning libraries, making it essential for scientific computing tasks in Python. [38]

- **Matplotlib**: Matplotlib is a comprehensive Python library for creating static, animated, and interactive visualizations. With a versatile set of plotting tools, it enables the generation of high-quality charts and graphs for effective data representation in fields such as data analysis, scientific research, and machine learning. [39]

- **pandas**: pandas stands as a robust Python library designed for the manipulation and analysis of data. It introduces two key data structures, Series and DataFrame, making it easy to handle and analyze structured data. pandas simplifies tasks like data cleaning, exploration, and transformation, serving as a crucial tool for data scientists and analysts. [40] [41]

- **Seaborn**: Seaborn is a statistical data visualization library for Python, based on Matplotlib. It furnishes a high-level interface for crafting visually appealing and informative statistical graphics. With concise syntax, Seaborn simplifies the generation of aesthetically pleasing plots, facilitating the exploration and presentation of complex datasets in a visually appealing manner.

- **Jupyter Notebook**: Jupyter is an open-source interactive web-based tool for creating and sharing live code, equations, visualizations, and narrative text. Supporting multiple programming languages, Jupyter notebooks empower users to develop, document, and execute code interactively, fostering collaborative and reproducible data science workflows in a flexible and user-friendly environment. [42]

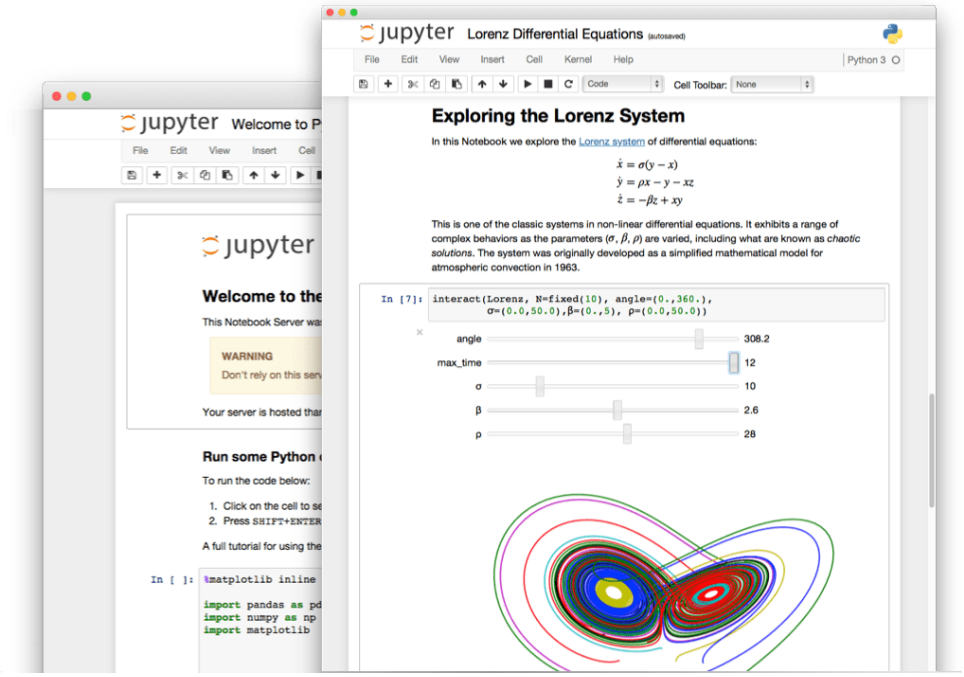

Figure 3.2 Example of Jupyter Notebook view [43].

3.2 Data Collection and Preparation

3.2.1 Selection and Acquisition of Data

The initial dataset for this research, comprising PDF papers from the IECON conference held in 2021, was provided by the project supervisor. These papers, representing contributions from various conference speakers, served as the training set for the machine learning model. Subsequently, for the evaluation phase, papers from the IECON conference in 2022 were incorporated. Accompanying the dataset was an Excel file detailing the 'track' or thematic category associated with each paper, a classification made by the respective speakers. Rigorous curation was conducted to exclude papers not listed in the track Excel file, ensuring a refined and relevant dataset for subsequent analysis and model development.

Tracks Preprocessing

In this section, we outline the process of preprocessing the Excel file containing track information to manage track details within our project. The objective is to extract unique track names and create lists of paper names associated with each track. Our implementation employs several functions to enhance organization and modularity.

The `table` function takes the folder path where the Excel file containing track information is located as input. It reads the Excel file and converts it into a DataFrame using the Pandas library. The function then extracts unique track names.

The `save_tracks_paper_list` function orchestrates the entire process. It iterates through each track, filters, and extracts paper names, saving them in designated folders.

This approach ensures a systematic preprocessing of track information, resulting in organized lists of paper names for subsequent analysis or use within the project. It's worth noting that in these lists,

papers are named according to their positions (0 to n) in the similarity matrix, facilitating subsequent calculations.

TrackList Management System

[ ]:
```
import os
import pandas as pd
import pickle
from string import digits


# Input: folder where the tracks Excel is located
# Output: list of unique track names and Excel converted to a DataFrame␣
 ↪(table)
def table(folder_excel):
    os.chdir(folder_excel)
    slots = pd.read_excel('classification.xlsx')  # Convert Excel to␣
 ↪DataFrame
    names = []

    # Include each track in the list only once
    for i in slots.index:
        track = slots['Name'][i]
        if track not in names:
            names.append(track)  # Unique names of the tracks
    return names, slots

# Function for saving tracks paper list
# Input: names of the tracks (names), Excel converted to DataFrame (slots),
# folder containing the list of names of all papers (folder_list),
# folder where to save the tracks (folder_track)
def save_tracks_paper_list(names, slots, folder_list, folder_track):
    k = -1
    papers = []  # List of paper names
    for name in names:
        track_list = []
        for i in range(slots.shape[0]):
            for j in range(slots.shape[1]):
                if not slots.iloc[i, j]:  # If the cell is not empty
                    k += 1  # Each time this loop is entered, a paper is␣
 ↪added to the list,
                                # therefore, it occupies position k in the list
                    paper = slots.iloc[i, j]
                    paper = str(paper) + '.pdf'
                    papers.append(paper)
                    track_list.append(k)  # List with all the papers of the␣
 ↪same track
```

```
        save_list(name, track_list, folder_track)  # Save the list in a file␣
↪inside folder_track
    # Save the list of papers in the folder folder_list
    save_list('papers_list', papers, folder_list)

# Utility function for saving a list to a file
# Input: file name (file_name), list to be saved (lista), folder to save the␣
↪file (folder)
def save_list(file_name, lista, folder):
    os.chdir(folder)  # Redirect to the folder
    # Write the list to a file named 'file_name'
    with open(file_name, "wb") as file:  # "wb" because pickle reads and␣
↪writes in binary
        pickle.dump(lista, file)
        file.close()
```

Papers Preprocessing

To initiate the preprocessing of the papers, the first step involved converting the PDF documents into plain text (txt) format. This conversion was executed using *pdfminer.six* [44], an open-source tool available on GitHub. This transformation facilitated subsequent text analysis and feature extraction procedures essential for the development of the machine learning model. The *problem_files* variable stores the names of files that caused exceptions during processing. This can be useful for debugging and managing issues that may arise.

PDF to Text Conversion Code with Error Handling

[ ]:
```
import os
from pdfminer.high_level import extract_text  # Ensure pdfminer is installed

# Function to write texts extracted from papers into text files
def write_texts(papers, folder_papers, folder_texts):
    problem_files = []  # List to store names of problematic files
    mkdir(folder_texts)  # Create folder to store text files

    for paper in papers:
        try:
            os.chdir(folder_papers)  # Switch to the directory where PDF␣
↪documents are located
            text = extract_text(paper)  # Use pdfminer to extract text from␣
↪the PDF
            file_name = paper + '.txt'

            os.chdir(folder_texts)  # Switch to the directory where text␣
↪files will be stored
            with open(file_name, 'w', encoding='utf-8') as file:
                file.write(text)
                file.close()
        except Exception as e:
```

```
            print(f"Error processing file {paper}: {str(e)}")
            problem_files.append(paper)  # Add problematic file to the list␣
↪of files with issues

    return problem_files  # Return the list of problematic files at the end␣
↪of the process
```

To obtain the titles of the papers, we utilize the function `papers_names`. This function extracts the titles from the text files within the specified folder, ensuring that the list of extracted papers maintains the same order as the placement of their corresponding texts in the corpus.

Paper Title Extraction

```
[ ]: # Function to extract paper titles from text files in a specified folder.
def papers_names(folder_texts):

    files_names = [] # Initialize an empty list to store the titles of the␣
↪papers
    files = os.listdir(folder_texts) # List all files in the specified folder

    # Iterate through each file in the folder
    for file_name in files:
        with open(os.path.join(folder_texts, file_name), 'r',␣
↪encoding='utf-8') as file:
            text = file.read()
            title_end_index = text.find("\n\n") # Find the index of the␣
↪first occurrence of "\n\n"
                                                 # which indicates the end of␣
↪the title
            title = text[:title_end_index].replace("\n", "").strip() #␣
↪Extract the title from the
            # remove any newline characters, and strip any leading or␣
↪trailing whitespace

            # Add the extracted title to the list of paper titles
            files_names.append(title)

    return files_names
```

In preparing the documents for the similarity task, a series of preprocessing steps were applied to clean and transform the text data. It is noteworthy that the chosen algorithm for text transformation incorporates certain preprocessing steps, rendering their application redundant at this juncture.

Before delving into the preprocessing steps, it is crucial to highlight the meticulous selection of preprocessing techniques, a decision guided by an in-depth analysis of the training dataset. This involved a systematic approach, where random texts were chosen, and their representations were visualized through word clouds [45]. A word cloud provides a visual depiction of the most frequent words in a corpus, with the size of each word indicating its frequency. In this context, it served as a diagnostic tool to identify noise and irrelevant information present in the texts.

Script for WordCloud Creation from Text File

```
[ ]: import os
from wordcloud import WordCloud
import matplotlib.pyplot as plt

# Change to the directory where text files are stored
os.chdir(folder_texts)

# List all files in the text files directory
files = os.listdir(folder_texts)

# Select the i-th file from the list
file_name = files[i]

# Open the file in read mode
with open(file_name, 'r', encoding='utf-8') as file:
    # Read the content of the file and store it in the 'text' variable
    text = file.read()

# Perform text processing using custom functions
text = prep_text(text)
text = lemat(text)

# Create a WordCloud based on the processed text
wordcloud = WordCloud().generate(text)

# Display the WordCloud in a graphical representation
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")  # Turn off the axes for a cleaner presentation
plt.show()  # Show the graphical representation of the WordCloud
```

Word clouds served to analyze the unique characteristics of the dataset, particularly within the context of IECON conferences. Scientific papers commonly feature numerous mathematical formulas incorporating Greek alphabets and numbers. Additionally, they often include tables and figures, which, when extracting text from PDFs, introduce noise due to persistent identifiers associated with these elements. To illustrate, examples of word clouds derived from the dataset are presented herein:

Figure 3.3 Example of the WordCloud of a text before and after preprocessing.

These visualizations not only offer a glimpse into the prevalent themes within the texts but also underscore the significance of noise reduction for more accurate analysis.

By examining these word clouds, it became evident that certain preprocessing steps were imperative to enhance the quality of the text data. The identified noise needed to be systematically removed to ensure the semantic purity of the content. This iterative process of analysis and refinement exemplifies the data-driven approach undertaken to tailor the preprocessing steps specifically to the characteristics of the IECON dataset. With this comprehensive understanding, we can now proceed to explore the intricacies of the preprocessing pipeline.

The preprocessing steps encompassed the following:

- **(CID:XX) Identifier Removal**: Instances of (CID:XX) identifiers, commonly present in tables and figures, were eliminated to ensure the removal of extraneous information that does not contribute to the semantic content of the text.
- **Figure Reference Removal**: Similarly, references to figures in captions, indicated by the term 'fig,' were also expunged from the text.

Text Preprocessing Function for Cleaning and Filtering Text Data

```
[ ]: import string
from string import digits

# Text preprocessing: removes (cid:nº), 'fig' and applies abc
# The lines of the function are in this order to be able to remove␣
 ↪'cid'
#even though transformers remove numbers, punctuation, and uppercase␣
 ↪letters

def prep_text(text):
    text = text.lower()  # Convert all text to lowercase
    text = text.translate(str.maketrans('', '', digits))  # Remove␣
 ↪numbers
    text = text.replace('(cid:)', '')  # 'CID' is an identifier
    text = text.translate(str.maketrans('', '', string.punctuation))  #␣
 ↪Remove punctuation
    text = ' '.join(word for word in text.split() if word != 'fig' and␣
 ↪abc(word) == True)
        # Apply the abc custom function to each word
    return text
```

- **Non-Alphabetic Token Filtering**: To concentrate on relevant and meaningful words, tokens containing non-alphabetic characters were discarded. This step aided in eliminating numbers, punctuation marks, and special characters that do not contribute to the semantic content of the text.

Clean Text by Removing Non-alphabetic Characters Function

```
[ ]: # Text preprocessing function
# Returns False for words with characters that are not letters from the␣
 ↪alphabet a-z
```

```
def abc(w):
    x = True
    for char in w:
        if ord(char) not in range(97, 123):
            x = False
    return x
```

- **Root Text Formation**: The text underwent tokenization into individual word units, and these tokens were then reassembled to form a cohesive text unit. This process involved:
  - **Lemmatization**: To perform lemmatization, the algorithm utilizes the *WordNetLemmatizer* class [46] from the NLTK library, considering the part of speech (noun, adjective, etc.) for accurate transformation to reduce each word to its common form.

NLTK-based POS Tagging and Lemmatization Implementation

```
[ ]: import nltk

# Download all NLTK resources only if it's the first time importing␣
 ↪NLTK in this environment, comment this line otherwise

#nltk.download('all')

from nltk import pos_tag
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer

# Function that maps NLTK POS tags to WordNet POS tags
def pos_tagger(nltk_tag):
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

# Function for lemmatizing text (reducing words to their base form)
def lemat(text):
    lema = WordNetLemmatizer()
    tokens = nltk.word_tokenize(text)  # Tokenize the text (split␣
 ↪all words and add them to a list in order)
    tagged_tokens = pos_tag(tokens)  # List of tuples (tuple =␣
 ↪(token, tag)), tag: type of word (noun, adjective, etc.)
    words_lema = []
```

```
    for token, tag in tagged_tokens:
        words_lema.append(lema.lemmatize(token, pos_tagger(tag)))
 ↪# Lemmatize each token indicating its POS tag
    text = ' '.join(word for word in words_lema)
    return text
```

– **Stemming**: The *PorterStemmer* class [47] was utilized to further reduce words to their root form.

Word-Level Text Stemming using NLTK's PorterStemmer

```
[ ]: from nltk.stem import PorterStemmer

# Stemming (reducing to stem)
def stem(text):
    porter = PorterStemmer()
    text = ' '.join(porter.stem(word) for word in text.split())  #
 ↪Apply stemming to each word
    return text
```

- **Text Corpus Construction**: In the final stages of preprocessing, the construction of the text corpus occurred. This involved employing a unified custom function (`txt_corpus`) that processed each text file in the designated folder. The preprocessing pipeline encompassed all the aforementioned steps, excluding lemmatization or stemming, as the choice between these will be determined later during the model training, allowing flexibility in text transformation methods. The outcome was a vector named *corpus* containing the preprocessed text ready for subsequent analysis and clustering within the context of IECON conferences.

Create Corpus from Files with Custom Preprocessing and Choice of Normalization

```
[ ]: # Function to preprocess and create a corpus using either stemming or
 ↪lemmatization
def txt_corpus(folder_texts, root_function):
    os.chdir(folder_texts)
    corpus = []
    files = os.listdir(folder_texts)  # List of files in the specified
 ↪folder
    for file_name in files:
        with open(file_name, 'r', encoding='utf-8') as file:
            text = file.read()
            file.close()
        text = prep_text(text)  # Apply the prep_text function
        text = root_function(text)  # Apply either stemming or
 ↪lemmatization
        corpus.append(text)  # Add the preprocessed paper to the corpus
 ↪vector
    return corpus
```

3.2.2 Feature Extraction and Text Representation

In Natural Language Processing (NLP), a pivotal step in preparing textual data for machine learning involves feature extraction, where raw text undergoes transformation into a numerical format. To achieve this, the Term Frequency-Inverse Document Frequency (TF-IDF) vectorization technique is widely employed, with implementation facilitated through the scikit-learn library.

TF-IDF Vectorizer

TF-IDF is a numerical statistic reflecting a word's importance in a document relative to a collection of documents (corpus). The *TfidfVectorizer* function in scikit-learn computes a matrix representing TF-IDF scores for each term in the corpus. It's noteworthy that during the tokenization process, the vectorizer makes preprocessing steps like removing plurals and one letter words. Scikit-learn, a powerful Python machine learning library, provides an efficient and user-friendly implementation of the TF-IDF vectorization process.

TF-IDF Generation Function with Adjustable Parameters

```
[ ]: from sklearn.feature_extraction.text import TfidfVectorizer

# Convert the corpus into a TF-IDF matrix
# The values in this matrix represent the importance of each word in each␣
 ↪text with respect to the corpus

def tfidf(corpus, parameters):
    transformer = TfidfVectorizer(**parameters)
    matrix_tfidf = transformer.fit_transform(corpus)
    return matrix_tfidf
```

Chosen Features and Parameters

Careful selection of various parameters in the TF-IDF vectorizer is crucial for obtaining meaningful representations of textual data. The chosen features and parameters for this task are as follows:

a. **Stopwords**: The 'english' stopword list is applied to eliminate common words contributing little to the text's overall meaning, thereby focusing on content-rich terms.

b. **mindf (Minimum Document Frequency)**: Experimentation with different minimum document frequency thresholds (mindf) excludes infrequently appearing terms, eliminating those with low significance.

c. **maxdf (Maximum Document Frequency)**: Setting a maximum document frequency threshold excludes terms appearing too frequently in the corpus, which may not be discriminative. This aids in filtering out terms common across various documents.

d. **ngram**: Experimentation with the ngramrange parameter is done for both unigrams (individual words) and bigrams (pairs of consecutive words) to observe the impact of considering single words and word pairs in the vectorization process.

e. **sublinear**: Setting the *sublineartf* parameter to true applies a sublinear transformation to the term frequency matrix. This diminishes the importance of very frequent terms and amplifies the importance of rarer terms, achieving a more balanced representation.

f. **maxfeatures**: The maximum number of features to be extracted is set to different values, recognizing that this may vary based on corpus size and computational resources. This parameter controls the dimensionality of the feature space.

The selection of these features and parameters aims to obtain a concise yet informative representation of textual data, considering factors like term frequency, document frequency, and the balance between common and rare terms. The removal of plurals and other grammatical forms during tokenization enhances the quality of representation.

Similarity measure

Cosine similarity serves as a metric for quantifying the similarity between two vectors, representing the cosine of the angle formed between them. In the realm of document similarity, this measure is specifically applied by comparing the TF-IDF vectors of two documents. The formula for calculating cosine similarity is expressed as follows:

$$Cosine\ similarity = \frac{\text{Dot product of the two vectors}}{\text{Length of the first vector} \times \text{Length of the second vector}}$$

The dot product of two vectors is obtained by summing the products of their corresponding elements, whereas the length of a vector is the square root of the sum of the squares of its elements. For practical implementation, the cosine similarity is computed through the *cosine_similarity* function found in the *sklearn.metrics.pairwise* module.

Cosine Similarity Matrix Calculation and Plotting

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics.pairwise import cosine_similarity

def plot_cosine_similarity(matrix_tfidf):
    # Calculate cosine similarity matrix
    matrix_sim = cosine_similarity(matrix_tfidf)

    # Plot the similarity matrix
    f, ax = plt.subplots(figsize=(50, 50))  # Determine the dimensions of␣
 ↪the figure
    sns.heatmap(matrix_sim, annot=True, linewidths=.5, fmt='.2f',␣
 ↪cmap="YlGnBu")
    plt.show()

    # Return the cosine similarity matrix
    return matrix_sim
```

In the generated heatmap with *seaborn.heatmap* [48], cells with darker colors indicate higher cosine similarity between the two documents associated with that cell.

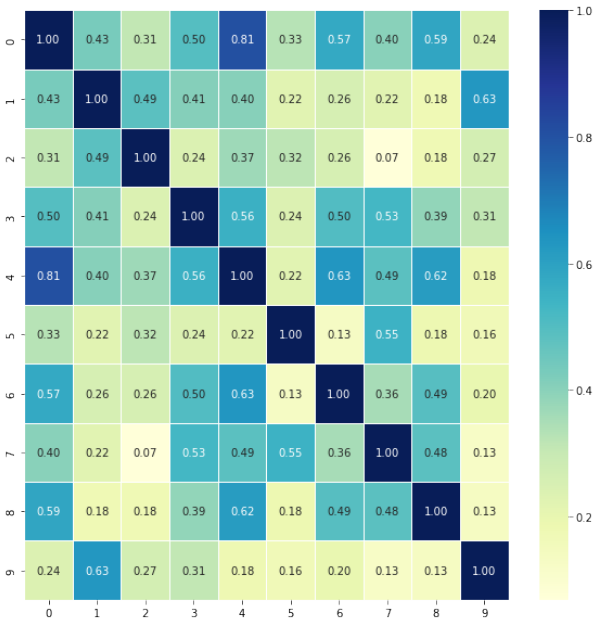

Figure 3.4 Cosine similarity heatmap example.

3.3 Optimization of Data Cleaning and Embedding

This section details the comprehensive strategy employed to optimize data cleaning and embedding for achieving optimal similarity scoring. The process begins with the meticulous selection of initial hyperparameters, informed by insights derived from word cloud visualizations and a thorough analysis of the training dataset.

Certain critical preprocessing steps, including the identification of *(CID:XX)* identifiers, figures, and non-alphabetic tokens, are intentionally excluded from hyperparameter tuning due to their fundamental importance, remaining consistent throughout the process. The initial hyperparameter set includes the choice between *lemmatization* and *stemming*, recognizing their distinct impacts on semantic content.

Subsequently, a systematic exploration of hyperparameter ranges is conducted, considering computational constraints. Parameters associated with TF-IDF embedding, such as *maxdf*, *mindf*, *ngram*, *maxfeatures*, and *sublineartf*, are defined based on a combination of empirical observations and practical considerations.

The calculation of similarity scores for all possible hyperparameter combinations, utilizing the cosine similarity metric, is executed with the goal of pinpointing the combination that maximizes similarity.

Hyperparameter Tuning Code for Text Analysis using TF-IDF and Cosine Similarity

```
[ ]: import numpy as np
from itertools import product
from sklearn.metrics.pairwise import cosine_similarity
import time

# Preprocessing parameters
root_functions = [stem, lemat]
```

```
# TfidfVectorizer parameters
max_df_values = np.arange(0.7, 1.01, 0.1)  # Ignore terms that appear in␣
 ↪more than 'max_df' documents: [0.8:1] (80%)
min_df_values = np.arange(0.0, 0.4, 0.1)  # Ignore terms that appear in less␣
 ↪than 'min_df' documents: [0:0.2] (20%)
ngram_ranges = [(1, 1), (1, 2)]  # Token: word / word and bi-gram␣
 ↪(consecutive word pairs)
max_features_values = [20, 50, 200, 500]  # Take the top 'max_features'␣
 ↪terms that are most frequent in the corpus
sublinear_tf_values = [True, False]  # Apply logarithm to the frequency of␣
 ↪each token in the corpus

# Hyperparameter combinations
combinations = list(product(root_functions, max_df_values, min_df_values,␣
 ↪max_features_values, ngram_ranges,
                             sublinear_tf_values))

# Calculate the score for each combination of parameters
# Input: Parameter combination
# Output: Score and hyperparameters
def tuning(combination, folder_texts):
    # Preprocessing
    parameters = {'root_function': combination[0]}
    corpus = txt_corpus(folder_texts, combination[0])

    # Tfidf
    parameters_tfidf = {'max_df': combination[1],
                        'min_df': combination[2],
                        'stop_words': 'english',
                        'max_features': combination[3],
                        'sublinear_tf': combination[5],
                        'ngram_range': combination[4]}
    matrix_tfidf = tfidf(corpus, parameters_tfidf)
    parameters.update(parameters_tfidf)

    # Cosine similarity
    matrix_sim = cosine_similarity(matrix_tfidf)
    score = metric(matrix_sim)

    return score, parameters

# Iterate through all parameter combinations and calculate their score
start_time = time.time()
i = 0
scores = []
hyperparameters = []
```

```
while True:
    combination = combinations[i]
    score, parameters = tuning(combination, r'C:
 ↪\Users\ana_s\OneDrive\Escritorio\tfg\archivos\texts_extract')
    hyperparameters.append(parameters)  # List with parameters for each␣
 ↪iteration
    scores.append(score)  # List with the score for each iteration

    i += 1
    if i == len(combinations):
        break
elapsed_time = time.time() - start_time
max_score = max(scores)  # Choose the maximum score
hyperparameters_max = hyperparameters[scores.index(max_score)]  # Choose the␣
 ↪parameters that achieve the maximum score
```

3.3.1 Similarity Metric

Effectively navigating this intricate parameter space requires a customized similarity metric. This metric quantifies clustering effectiveness by assessing both intra-cluster and inter-cluster similarities. It computes the median similarity of texts within the same track (intra-cluster) and subtracts the median similarity of texts from different tracks (inter-cluster). Maximizing this metric guides the crucial task of hyperparameter tuning.

We use the middle-point (median) instead of the average (mean) because it's less affected by outliers. This makes the evaluation of intra-cluster and inter-cluster similarities more reliable, especially when tuning settings, where precise assessment without getting thrown off by unusual values is critical for good clustering.

Track Similarity Metric for Evaluating TF-IDF Representations

[ ]:
```
import os
import pickle
import itertools
import statistics

def load_track(folder_tracks, i, name):
    """
    Load a track from a file.

    Args:
    - folder_tracks (str): Path to the folder containing track files.
    - i (int): Index of the track.
    - name (str): Name of the track file.

    Returns:
    - Loaded track.
    """
```

```
    os.chdir(folder_tracks)
    with open(name, "rb") as file:
        return pickle.load(file)

def combinations_same(folder_tracks):
    """
    Generate all combinations of pairs of papers within the same track.

    Args:
    - folder_tracks (str): Path to the folder containing track files.

    Returns:
    - List of all combinations of pairs of papers within the same track.
    """
    comb_total = []
    os.chdir(folder_tracks)
    names_tracks = os.listdir(folder_tracks)
    for i, name in enumerate(names_tracks):
        track = load_track(folder_tracks, i, name)
        comb = list(itertools.combinations(track, 2))
        comb_total.extend(comb)
    return comb_total

pairs_tracks = combinations_same(r'C:
 ↪\Users\ana_s\OneDrive\Escritorio\tfg\archivos\tracks')

def combinations_diff(folder_tracks, papers):
    """
    Generate all combinations of pairs of papers from different tracks.

    Args:
    - folder_tracks (str): Path to the folder containing track files.
    - papers (list): List of all paper names in the corpus.

    Returns:
    - List of all combinations of pairs of papers from different tracks.
    """
    comb_total = []
    os.chdir(folder_tracks)
    names_tracks = os.listdir(folder_tracks)
    for i, name in enumerate(names_tracks):
        track = load_track(folder_tracks, i, name)
        for j in track:
            for paper in papers:
                k = papers.index(paper)
                if k not in track:
                    comb_total.append((j, k))
```

```
    return comb_total

papers = [...]  # List of all paper names in the corpus
pairs_diff_track = combinations_diff(r'C:
 ↪\Users\ana_s\OneDrive\Escritorio\tfg\archivos\tracks', papers)

def same_track(matrix_sim):
    """
    Calculate similarity between papers from the same track.

    Args:
    - matrix_sim: Similarity matrix.

    Returns:
    - List of similarities between papers from the same track.
    """
    sim_track = []
    for pair in pairs_tracks:
        i, j = pair
        sim_track.append(matrix_sim[i][j])
    return sim_track

def diff_track(matrix_sim):
    """
    Calculate similarity between papers from different tracks.

    Args:
    - matrix_sim: Similarity matrix.

    Returns:
    - List of similarities between papers from different tracks.
    """
    sim_no_track = []
    for pair in pairs_diff_track:
        i, j = pair
        sim_no_track.append(matrix_sim[i][j])
    return sim_no_track

def metric(matrix_sim):
    """
    Calculate the metric based on similarity between papers
    from the same and different tracks.

    Args:
    - matrix_sim: Similarity matrix.

    Returns:
```

```
    - Metric score.
    """
    # Same track
    sim_track = same_track(matrix_sim)
    median_max = statistics.median(sim_track)

    # Different track
    sim_no_track = diff_track(matrix_sim)
    median_min = statistics.median(sim_no_track)

    score = median_max - median_min
    return score
```

Designed with a clear objective in mind — texts within the same track should exhibit higher similarity compared to texts from different tracks — the metric provides a robust measure of the model's proficiency in capturing meaningful clusters within the data.

3.3.2 Algorithm for Clustering and Model Development

Utilizing the K-means algorithm, data points are iteratively grouped based on Euclidean distance, optimizing cluster assignments. Renowned for its efficiency, simplicity, and scalability, K-means is particularly suited for datasets featuring a moderate to large number of instances.

The algorithm operates on a cosine similarity matrix, which encapsulates transformed textual data, and is implemented using the `KMeans` function within the Python `sklearn.cluster` module.

Leveraging the capabilities of the scikit-learn library, the algorithm produces visual representations via dynamic charts, where distinct groups are distinguished by different colors. Evaluations of groupings are conducted utilizing the silhouette metric, facilitating the recording of text assignments for subsequent analysis.

To address the constraint of maximum elements per cluster, the script adopts a strategy wherein the top elements closest to the centroid are selected if a cluster surpasses the threshold. Furthermore, unassigned elements are allocated to the nearest cluster within the limit.

The titles of papers linked with each cluster are saved in a CSV script, facilitating additional analysis and interpretation.

The comprehensive function not only provides the visualization of clusters, the silhouette score, and the dictionary of clusters with paper titles, but also ensures an integrated approach that fosters efficient clustering and offers insights into the composition of each cluster.

Clustering with K-Means: Visualization and Evaluation

```
[ ]: import matplotlib.pyplot as plt
     from sklearn.cluster import KMeans
     from sklearn.metrics import silhouette_score
     import numpy as np
     import csv

     def clustering(matrix_sim, n_clusters, papers_titles,␣
      ↪max_elements_per_cluster, folder_clusters):
```

```
    num_papers = len(papers_titles)

    if n_clusters >= num_papers:
        n_clusters = num_papers - 1
        print("Warning: Number of clusters is high. Adjusting to maximum␣
↪possible value:", n_clusters)

    if max_elements_per_cluster < (num_papers/ n_clusters):
        max_elements_per_cluster = int(num_papers/ n_clusters) + 1
        print("Warning: Maximum elements per cluster is low. Adjusting to␣
↪minimum possible value:", max_elements_per_cluster)

    # Execute clustering
    kmeans = KMeans(n_clusters=n_clusters).fit(matrix_sim)

    # Get labels and centroids
    labels = kmeans.labels_
    centroids = kmeans.cluster_centers_

    # Dictionary to store indices selected by cluster
    selected_indices_by_cluster = {}

    for i in range(n_clusters):
        # Find indices of points assigned to cluster i
        cluster_indices = np.where(labels == i)[0]

        # Check if there are enough points in the cluster to select
        num_points_in_cluster = len(cluster_indices)

        if num_points_in_cluster > max_elements_per_cluster:
            # Calculate distance from each point to cluster centroid
            distances = np.linalg.norm(matrix_sim[cluster_indices] -␣
↪centroids[i], axis=1)

            # Sort indices of points based on distance to centroid
            sorted_indices = cluster_indices[np.argsort(distances)]

            # Select the first max_elements_per_cluster points and store them
            selected_indices = sorted_indices[:max_elements_per_cluster]

            # Store selected indices in dictionary
            selected_indices_by_cluster[i] = selected_indices
        else:
            # If there are fewer points than max_elements_per_cluster,␣
↪select all available points
            selected_indices_by_cluster[i] = cluster_indices
```

```
    # Assign remaining indices to nearest clusters
    unassigned_indices = [idx for idx in range(len(matrix_sim)) if idx not␣
↪in np.concatenate(list(selected_indices_by_cluster.values()))]

    for idx in unassigned_indices:
        distances = [np.linalg.norm(matrix_sim[idx] - centroids[j]) for j in␣
↪range(n_clusters)]
        # Find the closest cluster that has not reached its capacity limit␣
↪yet
        closest_cluster = np.argmin(distances)
        # Look for the next available cluster if the closest cluster is full
        for j in range(1, n_clusters):
            next_cluster = (closest_cluster + j) % n_clusters
            if len(selected_indices_by_cluster[next_cluster]) <␣
↪max_elements_per_cluster:
                selected_indices_by_cluster[next_cluster] = np.
↪append(selected_indices_by_cluster[next_cluster], idx)
                break

    # Transform the dictionary to a labels array for next steps
    values_keys = [(value, key) for key, values in␣
↪selected_indices_by_cluster.items() for value in values]
    labels = [key for value, key in values_keys]
    values_keys.sort()
    labels = [key for value, key in values_keys]

    # Visualize the results
    plt.figure(figsize=(5, 5))  # Set the figure size
    clusters_visual = plt.scatter(matrix_sim[:, 0], matrix_sim[:, 1],␣
↪c=labels, s=50, cmap='rainbow')
    plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,␣
↪1], marker='x', s=200, c=range(len(kmeans.cluster_centers_)),␣
↪cmap='rainbow')
    plt.colorbar(clusters_visual)
    plt.show()

    # Silhouette score to evaluate clustering
    silhouette = silhouette_score(matrix_sim, labels)
    print('Silhouette score =', silhouette)

    # Generating unique CSV file name based on number of clusters and max␣
↪elements per cluster
    csv_name = f'clusters_papers_{n_clusters}_{max_elements_per_cluster}.csv'

    # Writing cluster assignments to CSV
    os.chdir(folder_clusters)
```

```
    with open(csv_name, 'w', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(['Paper Title', 'Cluster'])
        for i, cluster in enumerate(labels):
            writer.writerow([papers_titles[i], cluster])

    # Return the plot, silhouette score, and the papers clustering␣
↪dictionary
    return clusters_visual, silhouette
```

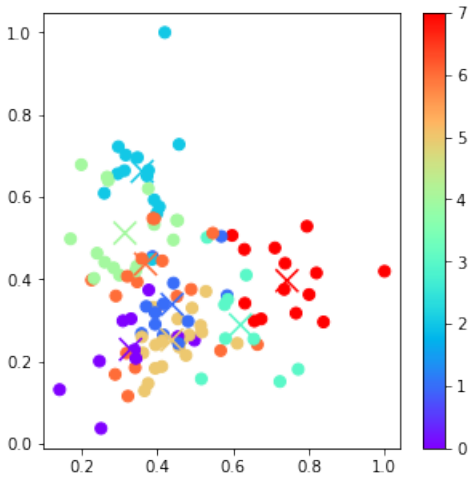

Figure 3.5 Example of representation of clustering.

3.3.3 Evaluation

Internal Metrics

- The **silhouette index**, our chosen metric, gauges cluster well-definition within the dataset. A higher score signifies more distinct clusters, quantifying algorithm performance.
- The **elbow method** guides optimal cluster determination, plotting distortion against cluster count to identify a distinct "elbow" point where the distortion rate sharply decreases.

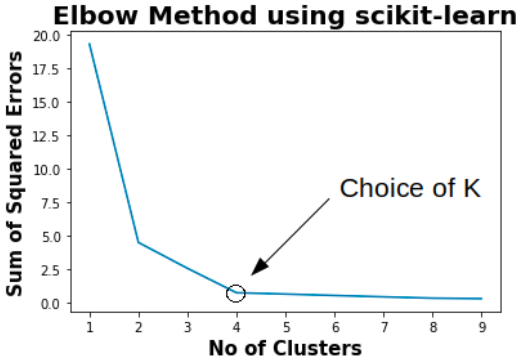

Figure 3.6 Elbow method in clustering [49].

Code for WCSS-based K-Means Evaluation

```
[ ]: #elbow_method uses KMeans on similarity matrix, plots WCSS to find optimal␣
      ↪clusters for given data.

def elbow_method(matrix_sim, n_clusters):
    wcss = []
    cl_num = n_clusters + 1
    for i in range(1, cl_num):
        kmeans = KMeans(i)
        kmeans.fit(matrix_sim)
        wcss_iter = kmeans.inertia_
        wcss.append(wcss_iter)

    number_clusters = range(1, cl_num)
    plt.plot(number_clusters, wcss)
    plt.title('The Elbow Method')
    plt.xlabel('Number of clusters')
    plt.ylabel('Within-cluster Sum of Squares')
    plt.show()
```

External Metric: Comparisons with Original Tracks

In this external evaluation, we assess the performance of our clustering methodology by conducting a comparative analysis against the original tracks provided in the dataset. The primary goal is to understand how well our clustering results align with the predefined groupings (tracks).

The evaluation metric employed in this analysis is designed to assign a score to each comparison. Specifically, a score of 1 is assigned when a predicted group perfectly matches a corresponding original track, indicating a correct prediction. On the other hand, a score of 0 is assigned when there is no match, signifying an incorrect prediction.

The comparative analysis provides valuable insights into the accuracy of our clustering algorithm in reproducing the predefined tracks. A higher cumulative score across all comparisons suggests a closer alignment between our clustered groups and the ground truth, demonstrating the effectiveness of our methodology in capturing the inherent patterns within the dataset.

This external metric serves as a crucial measure of the clustering model's performance, offering a clear assessment of its ability to reproduce the original groupings and highlighting areas for potential improvement.

Implementation of Metric for Cluster Performance (Predicted vs. True)

```
[ ]: def calculate_similarity_scores(predicted_groups, true_groups):
        # List to store final similarity scores.
        final_scores = []

        # Iterate through each predicted group.
        for predicted_group in predicted_groups:
            total_score = 0  # Initialize total score for the predicted group.

            # Iterate through each element in the predicted group.
```

```
        for predicted_element in predicted_group:
            matches = 0  # Initialize count of matching elements.
            len_predicted = len(predicted_group)  # Length of predicted␣
↪group.

            # Iterate through each true group.
            for true_group in true_groups:
                if predicted_element in true_group:
                    len_true = len(true_group)  # Length of true group.

                    # Iterate through each element in the true group.
                    for true_element in true_group:
                        if true_element in predicted_group and true_element !
↪= predicted_element:
                            matches += 1  # Increment matches count.

            # Calculate similarity score for the current predicted element.
            element_score = matches / (len_true - 1) if len_true > 1 else 0
            total_score += element_score  # Add element score to total score.

        # Calculate final similarity score for the predicted group.
        final_score = total_score / len_predicted if len_predicted > 0 else 0
        final_scores.append(final_score)  # Add final score to the list.

    # Print final scores and average score.
    print('Final Scores:', final_scores)
    print('Average Score:', statistics.mean(final_scores))

calculate_similarity_scores(labels, tracks)
```

4 Results

4.1 Analysis of Hyperparameter Configurations

This chapter thoroughly examines various combinations of hyperparameters and their influence on the results of our research. Through careful exploration, significant patterns have been revealed, offering valuable insights into the intricacies of textual relationships within the 2021 IECON conference papers dataset, which serves as our training dataset.

4.1.1 Chosen Hyperparameter Ranges

- **Preprocessing Parameters**
  - Root Function: stemming, lemmatization
- **TfidfVectorizer Parameters**
  - Max Document Frequency (max_df): 0.7, 0.8, 0.9, 1.0
  - Min Document Frequency (min_df): 0.0, 0.1, 0.2, 0.3
  - N-gram Range: (1, 1), (1, 2)
  - Max Features: 20, 50, 200, 500
  - Sublinear Term Frequency (sublinear_tf): True, False

These chosen hyperparameter ranges encompass critical aspects of our data preprocessing and vectorization strategy. The root function selection (stemming or lemmatization) and the settings for TfidfVectorizer parameters play a pivotal role in shaping the representation of textual data. The exploration of these ranges aims to identify the configurations that lead to optimal similarity scoring and meaningful text relationships within the dataset.

4.1.2 The Optimal Configuration

The combination of hyperparameters that has achieved the highest score is:

```
{'root_function': stem,
 'max_df': 0.8,
 'min_df': 0.0,
```

```
 'max_features': 50,
 'sublinear_tf': False,
 'ngram_range': (1, 1)}
```

To comprehend trends and patterns, each hyperparameter value is set to observe its variation across different combinations. Representation of these trends involves extracting the top five maximum values and the mean for each hyperparameter, enabling a comprehensive comparison. The goal of this approach is to identify the optimal hyperparameter combination not only at a specific point but in a general sense, taking into account overall trends and variations.

Highlight Top-Performing Hyperparameter Combinations in Matplotlib

[ ]:
```
def scores_plot(scores, combinations):
    import matplotlib.pyplot as plt

    # Create a dictionary to store scores corresponding to each␣
↪hyperparameter value
    scores_dict = {}

    # List to store combination information corresponding to each score
    combination_info = []

    # Fill the dictionary and combination information
    for combination, score in zip(combinations, scores):
        combination_info.append((combination, score))
        for hyperparam_value in combination:
            # Convert to a string if it is a boolean
            if isinstance(hyperparam_value, bool):
                hyperparam_value = str(hyperparam_value)
            if hyperparam_value not in scores_dict:
                scores_dict[hyperparam_value] = [score]
            else:
                scores_dict[hyperparam_value].append(score)

    # Create subplots
    fig, axs = plt.subplots(nrows=len(scores_dict), figsize=(5, 4 *␣
↪len(scores_dict)))

    # Iterate over the dictionary and plot on different subplots
    for i, (hyperparam_value, hyperparam_scores) in enumerate(scores_dict.
↪items()):
        axs[i].plot(hyperparam_scores, label=f'{hyperparam_value}')
        axs[i].legend()

        # Find the top 5 values
        max_indices = sorted(range(len(hyperparam_scores)), key=lambda j:␣
↪hyperparam_scores[j], reverse=True)[:5]

        # Highlight the top 5 values with a marker or label
```

```
        axs[i].scatter(max_indices, [hyperparam_scores[idx] for idx in␣
↪max_indices], color='red', marker='o', label='Maximums')

        # Draw a horizontal line at the mean
        mean_score = sum(hyperparam_scores) / len(hyperparam_scores)
        axs[i].axhline(y=mean_score, color='green', linestyle='--',␣
↪label='Mean')
        axs[i].legend()

    # Adjust layout
    plt.tight_layout()

    # Set manual limits to ensure consistency in x and y axes
    for ax in axs:
        ax.set_ylim(-0.051, 0.02)

    plt.show()

    return scores_dict, combination_info
```

- **Root Function**: Our investigation reveals a consistent trend where lemmatization tends to outperform stemming in terms of intra-cluster similarity. Despite the winning combination featuring `stem` as the root function, caution is advised, as this choice may be coincidental due to variability in the maxima. It is essential to prioritize the overall trend over individual instances.

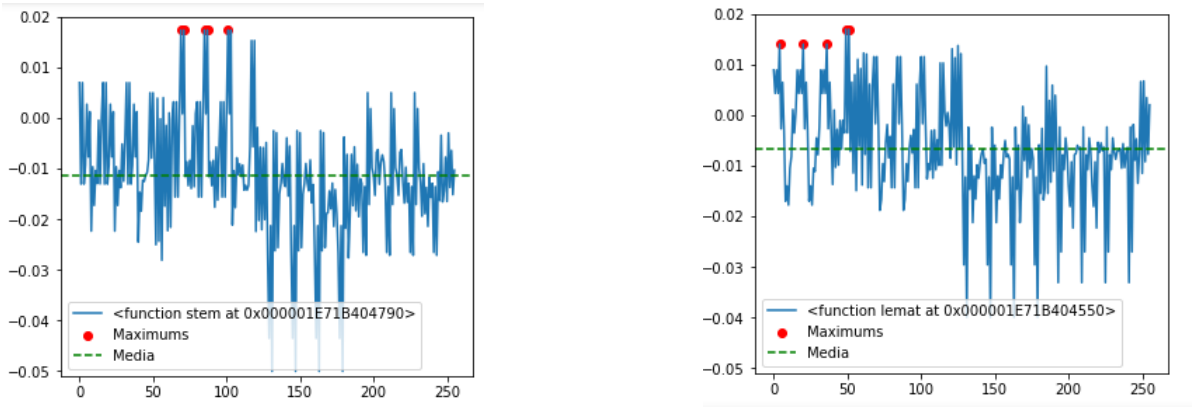

Figure 4.1 Scores obtained for every combination of hyperparameters: `stem` vs. `lemat`.

- **MaxDF**: An interesting observation emerges with MaxDF, showcasing optimal performance with values of 0.7 and 0.8.

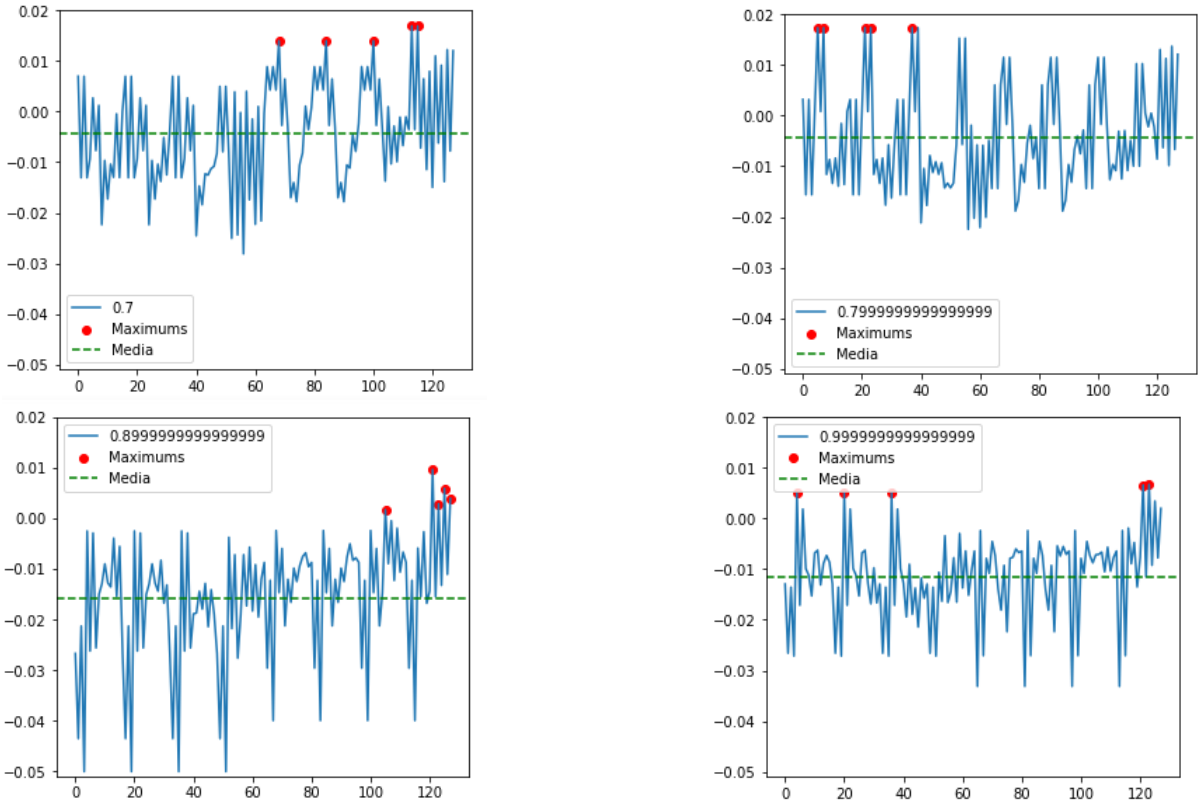

Figure 4.2 Scores obtained for every combination of hyperparameters: range of MaxDf values.

- **MinDF**: The analysis of MinDF indicates that it attains comparable results across the considered values. The observed similarity in outcomes among different MinDF settings highlights the robustness of the model's performance, suggesting that variations in this parameter do not significantly impact the overall results.

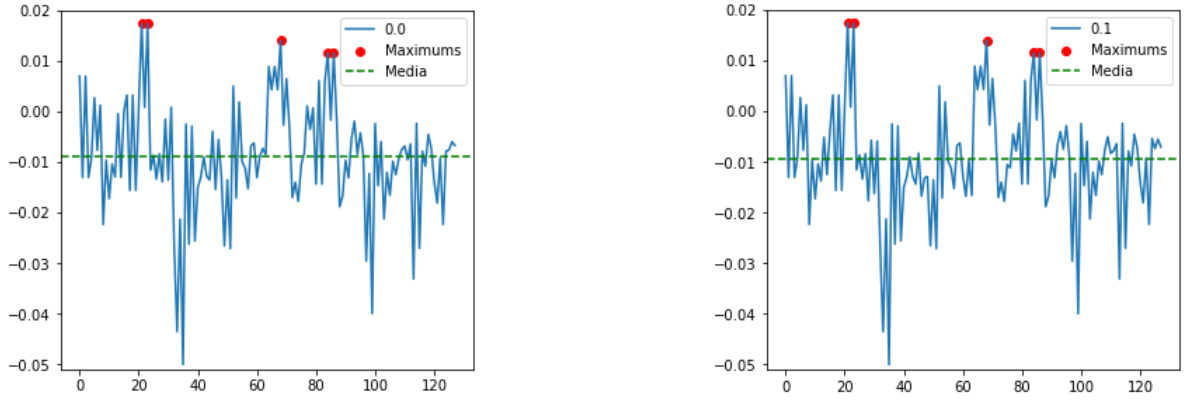

Figure 4.3 Scores obtained for every combination of hyperparameters: range of `mindf` values.

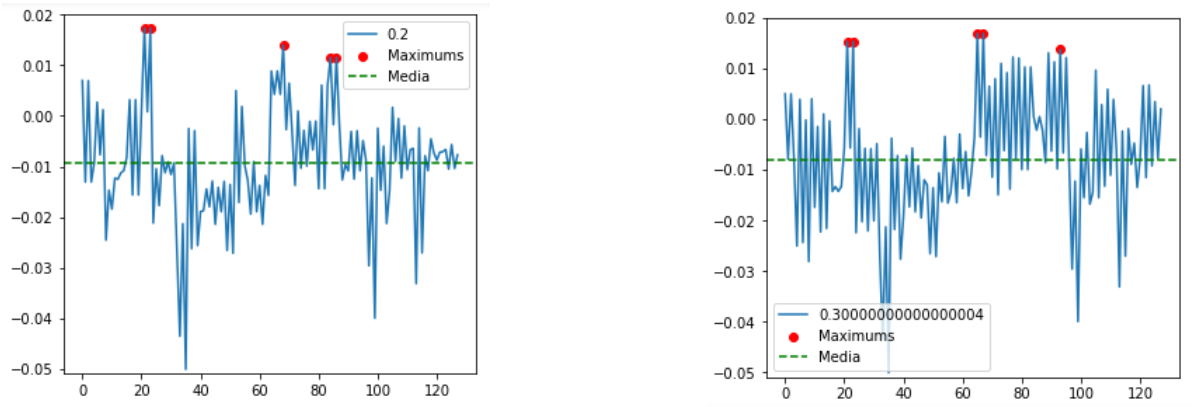

Figure 4.3 Scores obtained for every combination of hyperparameters: range of `mindf` values (continued).

- **Sublinear TF**: The mean similarity scores for True and False are similar, but False consistently appears in most of the maximum values. However, it's worth noting that the maximum scores for True are close, and False has lower minimum values.

Figure 4.4 Scores obtained for every combination of hyperparameters: `sublinearTF` `True` vs. `False`.

- **Ngram Range**: A preference for (1, 1) over (1, 2) in the *ngram_range* parameter is evident from our analysis. This choice aligns with the tendency of (1, 1) to yield superior results, providing clarity on the optimal configuration for this aspect.

Figure 4.5 Scores obtained for every combination of hyperparameters: ngram `(1,1)` vs. `(1,2)`.

- **Max Features**: In the exploration of different values for the *Max Features* parameter, it is evident that a setting of 50 exhibits superior performance. Not only does it achieve a higher mean similarity score, but it also consistently secures higher maximum values.

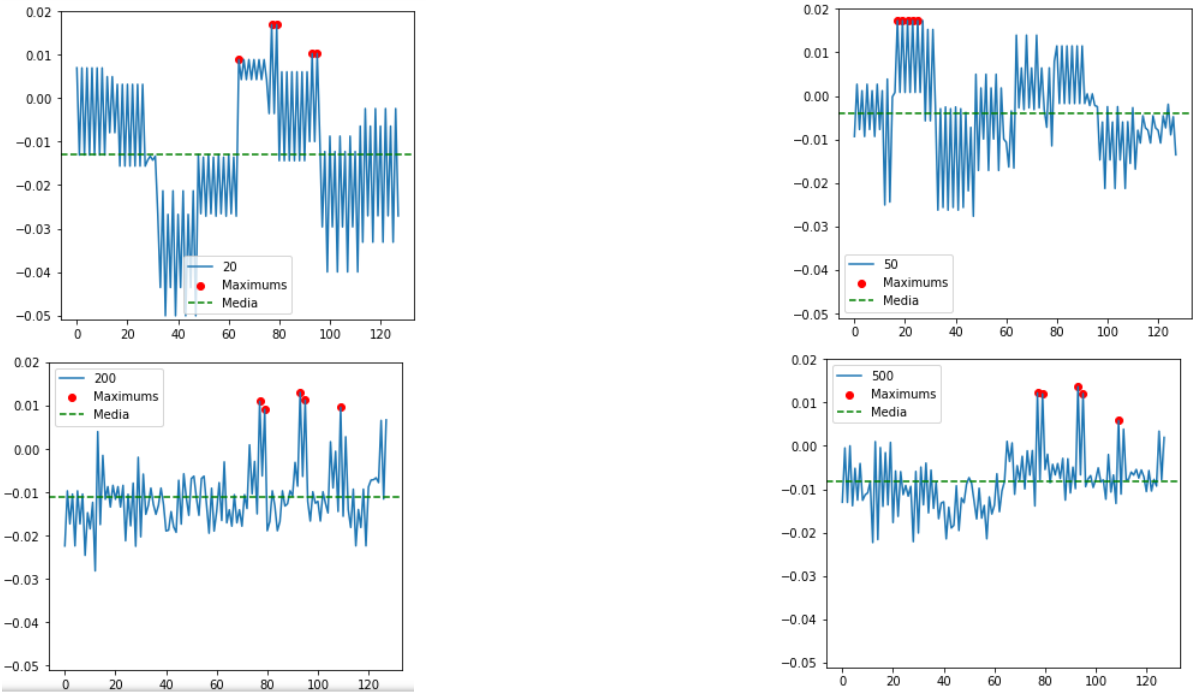

Figure 4.6 Scores obtained for every combination of hyperparameters: range of MinDf values.

Maximum Scores

Here is presented a compilation of the most prominent hyperparameter combinations, along with their respective scores, obtained during the model tuning process.

Table 4.1 Top Scoring Hyperparameter Combinations.

Score	Root Function	MaxDF	MinDF	Max Features	Ngram Range	Sublinear TF
0.0173	Stem	0.8	0.0	50	(1, 1)	False
0.0173	Stem	0.8	0.0	50	(1, 2)	False
0.0173	Stem	0.8	0.1	50	(1, 1)	False
0.0173	Stem	0.8	0.1	50	(1, 2)	False
0.0173	Stem	0.8	0.2	50	(1, 1)	False
0.0173	Stem	0.8	0.2	50	(1, 2)	False
0.0169	Lemmat	0.7	0.3	20	(1, 1)	False
0.0169	Lemmat	0.7	0.3	20	(1, 2)	False
0.0153	Stem	0.8	0.3	50	(1, 1)	False
0.0153	Stem	0.8	0.3	50	(1, 2)	False
0.0140	Lemmat	0.7	0.0	50	(1, 1)	True
0.0140	Lemmat	0.7	0.1	50	(1, 1)	True

Score	Root Function	MaxDF	MinDF	Max Features	Ngram Range	Sublinear TF
0.0140	Lemmat	0.7	0.2	50	(1, 1)	True
0.0137	Lemmat	0.8	0.3	500	(1, 1)	False
0.0130	Lemmat	0.8	0.3	200	(1, 1)	False

After careful examination, a specific configuration was chosen:

```
{'root_function': lemat,
 'max_df': 0.7,
 'min_df': 0.2,
 'max_features': 50,
 'sublinear_tf': True,
 'ngram_range': (1, 1)}
```

This decision stemmed from a comprehensive analysis indicating consistent strong performance for this combination with lemmatization, *Max Features: 50*, and *Ngram Range: (1, 1)*. While the individual score of 0.0140 may not be the pinnacle, this configuration was favoured due to its excellence in crucial areas. The choice of 0.2 aligns with the established pattern and ensures a balance between capturing meaningful terms and avoiding overly restrictive feature selection. The adopted approach prioritizes a more balanced optimization, valuing consistency and robustness over singular peak scores.

4.2 Clustering Results

In this section, we present the outcomes of the clustering process applied to the IECON 2022 papers using the test dataset.

4.2.1 Text Preprocessing and Transformation

To initiate clustering, we apply the chosen hyperparameters to preprocess and transform the text data. Texts undergo tokenization, stemming, and stop word removal to enhance their suitability for clustering. The resulting preprocessed texts are then transformed into a cosine similarity matrix, serving as the foundational representation for subsequent K-Means clustering.

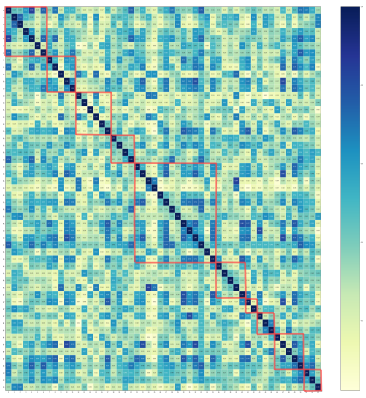

Figure 4.7 Cosine similarity heatmap of the test dataset..

After generating the cosine similarity heatmap, tracks from the original dataset are highlighted in red. Notably, papers within the same track demonstrate a high degree of similarity, validating the clustering approach. However, the heatmap also reveals instances where the algorithm has identified substantial similarity between papers not originally in the same track. This suggests that the algorithm has captured additional relationships beyond the predefined tracks, opening avenues for further exploration and analysis.

Here, two papers within the same track and two papers outside the same track are showcased through word clouds. The similarity score between the first pair is 0.84, while between the second pair, it is 0.76. As evident, they share key words, leading to higher similarity, even though the presenters assigned them to different tracks.

Figure 4.8 Word Clouds - Same Track.

Figure 4.9 Word Clouds - Different Tracks.

4.2.2 K-Means Clustering

In this subsection, we delve into the application of K-Means clustering to the cosine similarity matrix derived from the IECON 2022 test dataset. The number of clusters and de maximum number of papers per cluster are initially set to align with the predefined tracks within the dataset, providing a logical basis for grouping.

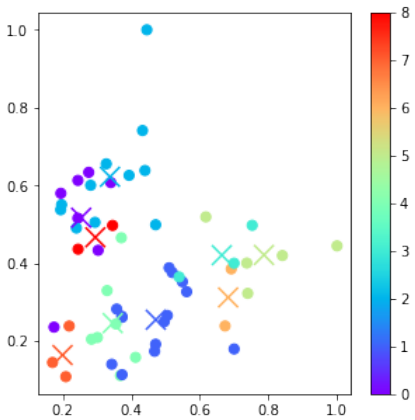

Figure 4.10 2022 IECON papers clustering (*n_clusters*=9, *max_elements_per_cluster*=15).

4.2.3 Evaluation Metrics

Internal Evaluation: Elbow Method and Silhouette Score

The Silhouette Score is 0.19415587908239387.

The elbow method acts as a heuristic to identify the most suitable number of clusters in a dataset. This approach entails plotting the total within-cluster sum of squared errors (SSE) against various values of K, representing the number of clusters. SSE gauges how effectively each data point fits within its assigned cluster. The optimal cluster count is pinpointed at the juncture where the SSE experiences a notable upturn.

In this scenario, although the SSE initially diminishes with an increasing number of clusters, it begins to plateau after reaching 5 clusters. This implies that, according to the elbow method, the optimal number of clusters for this dataset is 5.

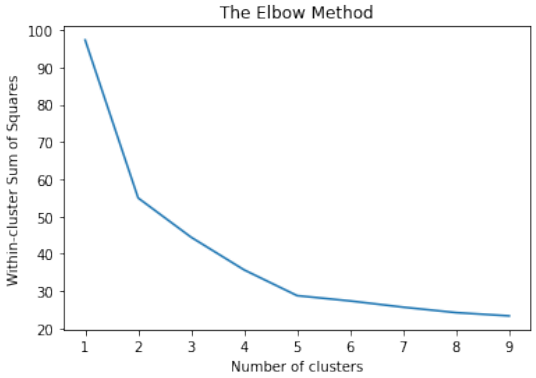

Figure 4.11 Elbow method assessment on the test dataset for clustering..

External Evaluation: Comparisons with Original Tracks

Externally, we conduct a comparative analysis against the original tracks provided in the dataset. The evaluation metric assigns a score of 1 for correctly predicted groups and 0 otherwise. This assessment aims to reveal the extent to which our clustering methodology aligns with the predefined tracks.

In this case the average score is 0.13878925307496734.

Exploration of Different Cluster Numbers

In this exploration, our objective is to assess the impact of utilizing 5 clusters, a number identified as optimal through the elbow method. This investigation allows us to discern potential improvements in the clustering results. The upper limit for the number of papers per cluster is set generously high to observe the outcomes of the KMeans clustering process without necessitating reassignment of elements.

Below, we present the visual representation of the clusters, offering insights into their structure and composition.

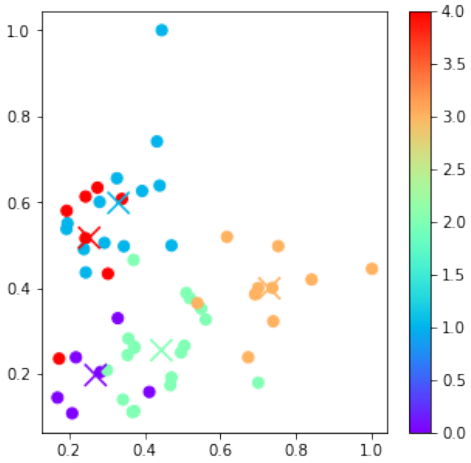

Figure 4.12 2022 IECON papers clustering (*n_clusters*=5, *max_elements_per_cluster*=20).

The silhouette index for the 5-cluster configuration is 0.3203293461460062. Additionally, the comparative score with the original track is 0.22271701843130415.

In the following table, we compare the metrics for 9 and 5 clusters:

Number of Clusters	Silhouette Index	Comparative Score
9	0.17	0.12
5	0.32	0.22

Table 4.2 Comparison of metrics for 9 and 5 clusters.

Exploring Various Cluster Sizes

We will now lower the maximum number of elements per cluster, considering that the largest cluster previously contained 18 papers, in order to observe how the results are affected:

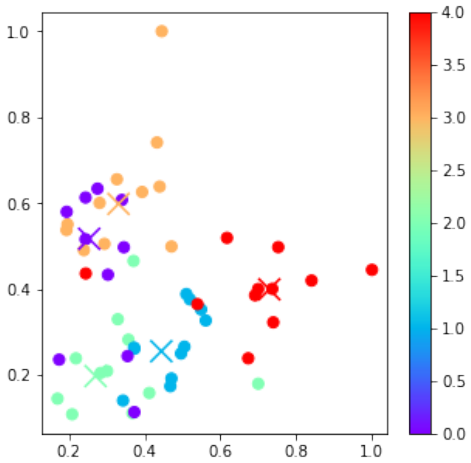

Figure 4.13 2022 IECON papers clustering (*n_clusters*=5, *max_elements_per_cluster*=11).

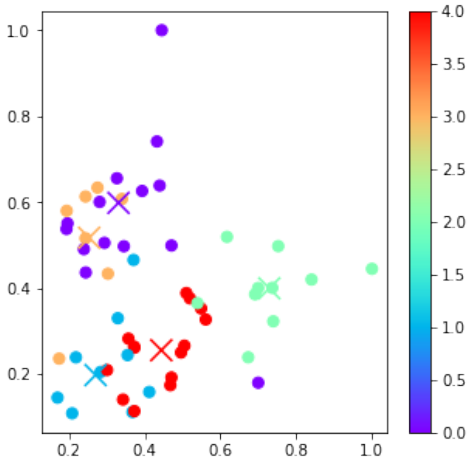

Figure 4.14 2022 IECON papers clustering (*n_clusters*=5, *max_elements_per_cluster*=11).

Max Size of Cluster	Silhouette Index	Comparative Score
11	0.17	0.22
14	0.28	0.21

Table 4.3 Comparison of metrics for 11 and 14 maximum papers per cluster.

As we can see, the silhouette index decreases when restricting the cluster size; however, the comparison with the tracks remains largely unchanged.

Example of Paper Titles

In this demonstration, we'll utilize a reduced collection of test papers to showcase the titles grouped within clusters. We've configured the demonstration to feature four clusters, each containing a maximum of six papers.

- "Normalised hybrid flux weakening strategy for automotive asymmetrical dual three-phase IPMSMs".
- "Reward Shaping-based Double Deep Q-networks for Unmanned Surface Vessel Navigation and Obstacle Avoidance".
- "The state feedback control for a class of singular Markovian jump systems subject to input saturation and time delay*".
- "Synthesis of Decentralized Variable Gain Robust Controllers with Guaranteed L2 Gain Performance via Piecewise Lyapunov Functions for a Class of Uncertain Large-Scale Interconnected Systems".
- "Direct Torque Control in Series-End Winding PMSM Drives".
- "Urban road users detection and velocity estimation from top-view fish-eye imagery under low light conditions".
- "Comparative Study on Collision Avoidance Methods in Path Planning for Warehouse Robots Using MPC".
- "Coordinated Charging Strategy of Cascaded H-bridge With Bidirectional DC-DC Converter for Supercapacitor Energy Storage Applications".

- “A Dense Multilevel 24-sided Polygonal Voltage Space Vector Structure for IM Drive with Open-end Winding Configuration”.
- “A Quadratic High Step-up Interleaved Converter with Coupled Inductor”.
- “A Novel Voltage Balancing Method of Cascaded H-bridge Multilevel Converter With Supercapacitors Energy Storage System for Capacitor Voltage Ripple Reduction”.
- “Research on the Stochasticity Control Strategy of Wind Farm Incorporating System Contingencies”.
- “Robust control and energy management in a hybrid DC microgrid using second-order SMC”.
- “A DCX-LLC Resonant Converter”.
- “A System for Identification of Lamps Based on Artificial Intelligence”.
- “Improving Disturbance-Rejection Performance Using Combination of Sliding-Mode Control and Equivalent-Input-Disturbance Approach”.
- “Synchronous reluctance motor flux linkages saturation modeling based on stationary identification and neural networks”.
- “A novel rotor position estimation method of permanent magnet synchronous motor based on DC compensation and cascade filter”.
- “Direct-axis Dead-time Effect Compensation Strategy Based on Adaptive Linear Neuron Method for PMSM Drives”.
- “Improved MPPT Algorithm for Differential Power Processing PV Converters”.
- “DPP Converters with Reduced Sensors”.
- “High Order Fast Terminal Sliding-Mode Control of Permanent Magnet Synchronous Motor”.

Table 4.4 Titles of Papers Within Clusters.

Cluster	Titles of Papers
0	**1.** "Normalised hybrid flux weakening strategy for automotive asymmetrical dual three-phase IPMSMs". **2.** "Direct Torque Control in Series-End Winding PMSM Drives". **3.** "Synchronous reluctance motor flux linkage saturation modeling based on stationary identification and neural networks". **4.** "A novel rotor position estimation method of permanent magnet synchronous motor based on DC compensation and cascade filter". **5.** "Direct-axis Dead-time Effect Compensation Strategy Based on Adaptive Linear Neuron Method for PMSM Drives". **6.** "High Order Fast Terminal Sliding-Mode Control of Permanent Magnet Synchronous Motor".

Continued on the next page.

Table 4.4 – Continued from previous page

Cluster	Titles of Papers
1	**1.** "Reward Shaping-based Double Deep Q-networks for Unmanned Surface Vessel Navigation and Obstacle Avoidance". **2.** "Urban road users detection and velocity estimation from top-view fish-eye imagery under low light conditions". **3.** "Comparative Study on Collision Avoidance Methods in Path Planning for Warehouse Robots Using MPC". **4.** "Research on the Stochasticity Control Strategy of Wind Farm Incorporating System Contingencies". **5.** "A System for Identification of Lamps Based on Artificial Intelligence".
3	**1.** "The state feedback control for a class of singular Markovian jump systems subject to input saturation and time delay*". **2.** "Synthesis of Decentralized Variable Gain Robust Controllers with Guaranteed L2 Gain Performance via Piecewise Lyapunov Functions for a Class of Uncertain Large-Scale Interconnected Systems". **3.** "Improving Disturbance-Rejection Performance Using Combination of Sliding-Mode Control and Equivalent-Input-Disturbance Approach". **4.** "Improved MPPT Algorithm for Differential Power Processing PV Converters". **5.** "DPP Converters with Reduced Sensors".
2	**1.** "Coordinated Charging Strategy of Cascaded H-bridge With Bidirectional DC-DC Converter for Supercapacitor Energy Storage Applications". **2.** "A Dense Multilevel 24-sided Polygonal Voltage Space Vector Structure for IM Drive with Open-end Winding Configuration". **3.** "A Quadratic High Step-up Interleaved Converter with Coupled Inductor". **4.** "A Novel Voltage Balancing Method of Cascaded H-bridge Multilevel Converter With Supercapacitors Energy Storage System for Capacitor Voltage Ripple Reduction". **5.** "Robust control and energy management in a hybrid DC microgrid using second-order SMC". **6.** "A DCX-LLC Resonant Converter".

5 Conclusions and Future Work

5.1 Key Findings and Achievements

This project centered around the application of clustering methodologies to enrich the thematic coherence of conference rooms, specifically within the context of the IECON event. The aim was to optimize speaker arrangements, fostering a more engaging and cohesive conference experience through meticulous analyses of text relationships and hyperparameter tuning.

The obtained results unveiled both triumphs and avenues for improvement. Highlighting the key findings and achievements:

1. **Insights from Hyperparameter Exploration:** The analysis provided valuable insights into the impact of various settings on text relationships. Although achieving the highest similarity score posed challenges, the exploration significantly advanced our understanding, laying the groundwork for future improvements.

2. **Promising Validation of K-means Clustering:** The application of K-means clustering exhibited promise, substantiated by heatmap and word cloud analyses. The positive outcomes signify a notable stride forward, with potential applications in enhancing thematic coherence.

3. **Enhanced Robustness through Analysis:** Integrating validation metrics (e.g., Elbow Method, Silhouette Score) with robustness and sensitivity analysis bolstered the project's robustness. Understanding the model's stability under different conditions represents a significant achievement.

In conclusion, this study marks a notable advancement in clustering methodologies for conference paper analysis. The achieved insights and accomplishments pave the way for future research, emphasizing the importance of building upon current successes, continuous refinement, and leveraging domain knowledge for even more impactful results.

5.2 Future Directions and Opportunities

While this research has established a foundation, exciting opportunities exist for further advancement:

1. **Advanced Text Representation:** Integrating advanced techniques like word embeddings or contextual embeddings (e.g., transformer models) can capture more nuanced paper relationships.

2. **Domain-Specific Knowledge:** Incorporating conference-specific themes, topics, and author relationships can guide the clustering process, potentially leading to more relevant groupings.
3. **Dynamic Clustering:** Implementing dynamic approaches can adapt to evolving conference structures and themes over time, ensuring flexibility and ongoing effectiveness.
4. **Ensemble Methods:** Combining multiple clustering algorithms or representations through ensemble methods can enhance robustness and generalizability across various conference settings.
5. **Continuous Refinement:** Ongoing hyperparameter tuning and exploration of additional parameters can optimize clustering performance for different conferences and contexts.
6. **User Feedback Integration:** User feedback loops, collecting input from participants and organizers, can continuously refine the model and ensure practical effectiveness and thematic coherence.
7. **Transformer-Based Pretraining:** Pretraining transformer models on relevant conference paper corpora can impart domain-specific knowledge, leading to more accurate and context-aware clustering.
8. **Dynamic Hyperparameter Adjustment:** Developing mechanisms for adjusting hyperparameters during clustering, leveraging real-time feedback or adaptive strategies, can enhance model adaptability to evolving data.
9. **Speaker Preference Incorporation:** Extending the algorithm to account for speaker preferences and collaboration histories can further improve cluster quality by leveraging rich contextual information.
10. **Diverse Evaluation Metrics:** Expanding the evaluation metrics beyond the Silhouette index to incorporate additional measures of cluster quality, potentially including metrics that consider semantic coherence and context-awareness, can provide a more nuanced understanding of clustering effectiveness specifically for conference settings.

This research lays the groundwork for further development and application of advanced clustering methodologies in the dynamic and nuanced context of conference organization. The potential to leverage transformer models, incorporate domain knowledge, and integrate user feedback paves the way for significant advancements in conference room grouping, ultimately aiming to foster a more engaging and thematically coherent experience for participants.

Appendix A
User Manual

A.1 Introduction

This guide assists you in utilizing a Python code to streamline and enhance the process of allocating speakers for conferences, with a particular emphasis on IECON (IEEE Industrial Electronics Conference).

A.2 Functionality and Dependencies

The code makes use of various Python libraries and modules, such as `NLTK`, `os`, `numpy`, and `scikit-learn`.

Furthermore, the proper execution of the code relies on custom functions detailed in preceding sections. These functions include:

- `write_texts`: A function designed to write texts to files.
- `papers_names`: This function extracts paper titles from text files within a specified folder.
- `txt_corpus`: A function responsible for creating a corpus from text files.
- `tfidf`: A function performing TF-IDF vectorization on a set of documents.
- `plot_cosine_similarity`: A function used to visualize cosine similarity between documents.
- `elbow_method`: A function for determining the optimal number of clusters using the elbow method.
- `clustering`: A function for executing document clustering.

In order to ensure the correct operation of the code, these custom functions must be available.

A.3 Setup

Users are prompted to input paths to the paper folder and the location for extracted texts. The '`r`' before paths ensures they are treated as raw strings.

User Input for Specifying Papers Folder and Text Extraction Destination Folder

```
[ ]: # User inputs
folder_papers = input(r"Please provide the path to the papers folder: ")
folder_texts = input(r"Please provide the path to save the extracted texts:␣
 ↪")
folder_clusters = input(r"Please provide the path to save the clusters of␣
 ↪paper titles:")
```

A.4 Text Processing

The code retrieves files from the specified folder, extracts text using the `write_texts` function, and handles problematic PDFs.

Raw String Handling for User-Input Paths and PDF Text Extraction

```
[ ]: #The letter 'r' is added to the beginning of the paths entered by the user,␣
 ↪ensuring that they are treated as raw strings
folder_papers = r"" + folder_papers
folder_texts = r"" + folder_texts

# Extract texts from papers and write them to text files
# Notify about problematic PDFs
papers = os.listdir(folder_papers)
problems = write_texts(papers, folder_papers, folder_texts)
if problems:
    print("Problematic files:")
    for problem in problems:
        print(problem)

num_pdfs = len(papers)
num_texts = len(papers) - len(problems)
num_problems = len(problems)
print(f"The number of PDFs is {num_pdfs}, the number of extracted texts is␣
 ↪{num_texts}, and the number of problematic PDFs is {num_problems}.")
```

A.5 TF-IDF Vectorization and Similarity

A corpus is created using lemmatization (`lemat`), and the TF-IDF vectorizer transforms it into a numerical matrix (`matrix_tfidf`). Cosine similarity is then calculated based on this matrix.

A heatmap of the cosine similarity matrix is generated using `seaborn` and displayed with `matplotlib`.

Text Corpus Creation and TF-IDF Transformation

```
[ ]: # Create a list of the papers titles
papers_titles = papers_names(folder_texts)

# Create a corpus from the extracted texts
```

```
corpus = txt_corpus(folder_texts, lemat)

# Transform the corpus using TF-IDF vectorizer
parameters = {'max_df' : 0.7, 'min_df' : 0.2, 'stop_words' : 'english',␣
 ↪'max_features' : 50, 'sublinear_tf' : True}
matrix_tfidf = tfidf(corpus, parameters)

# Calculate and plot cosine similarity matrix
matrix_sim = plot_cosine_similarity(matrix_tfidf)
```

A.6 Clustering

Users input the desired number of clusters for the KMeans algorithm.

The code applies the elbow method to determine the optimal number of clusters, helping users identify a suitable cluster number based on a plot of within-cluster sum of squares.

A loop allows users to experiment with different cluster numbers and the maximum number of papers per cluster. The code executes the KMeans algorithm, visualizes clustering results, calculates silhouette scores and save the clusters in a script.

This iterative process enables users to explore and evaluate different cluster configurations until they are satisfied with the outcome.

Clustering Analysis and User Interaction

[ ]:
```
# Input for the number of clusters
n_clusters = int(input("Please enter the desired number of clusters: "))
```

[ ]:
```
#elbow_method uses KMeans on similarity matrix, plots WCSS to find optimal␣
 ↪clusters for given data.
elbow_method(matrix_sim, n_clusters)
```

[ ]:
```
mkdir(folder_clusters)  # Create folder to store cluster files

while True:
    # Input for the number of clusters
    n_clusters = int(input("Please enter the desired number of clusters: "))

    # Input for the maximum number of elements per cluster
    max_elements_per_cluster = int(input("Please enter the maximum number of␣
 ↪elements per cluster: "))

    clusters_visual, silhouette = clustering(matrix_sim, n_clusters,␣
 ↪papers_titles, max_elements_per_cluster, folder_clusters)

    # Ask the user if they want to try another number of clusters and␣
 ↪maximum number of elements.
```

```
    another_try = input("Do you want to try another number of clusters and␣
↪maximum number of elements? (y/n): ").lower()

    if another_try != 'y':
        break
```

Bibliography

[1] IECON 2023. Iecon 2023. *https://www.iecon2023.org/*, 2023.

[2] Aniket Patil Vashi. Natural language processing: Advance techniques. *https://medium.com/analytics-vidhya/natural-language-processing-advance-techniques-in-depth-analysis-b67bca5db432*, 2021.

[3] Amanda Morin. How to use a venn diagram. *https://www.verywellfamily.com/what-is-a-venn-diagram-620971*, 2020.

[4] J. Shields and S. Bos. The venn diagram: How circles illustrate relationships. *https://science.howstuffworks.com/math-concepts/venn-diagram.htm*, 2023.

[5] Ascribe. Taxonomies in text analytics. *https://goascribe.com/blog/taxonomies-text-analytics/*, 2023.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Harry Guinness. What is gpt? everything you need to know about gpt-3 and gpt-4. *https://zapier.com/blog/what-is-gpt/*, 2023.

[8] Pawar Kumar Gunjan. Rule based approach in nlp. *https://www.geeksforgeeks.org/rule-based-approach-in-nlp/*, 2023.

[9] WB Mao, JY Lyu, DK Vaishnani, YM Lyu, W Gong, XL Xue, YP Shentu, and J Ma. Application of artificial neural networks in detection and diagnosis of gastrointestinal and liver tumors. *World J Clin Cases*, 2020.

[10] Andrew Ng and Karthik Raman. A complete guide to natural language processing. *https://www.deeplearning.ai/resources/natural-language-processing/*, 2023.

[11] Raghav Agrawal. Must known techniques for text preprocessing in nlp. *https://www.analyticsvidhya.com/blog/2021/06/must-known-techniques-for-text-preprocessing-in-nlp/*, 2022.

[12] Sara A. Metwalli. 6 nlp techniques every data scientist should know. *Towards Data Science*, 2021. URL *https://towardsdatascience.com/6-nlp-techniques-every-data-scientist-should-know-7cdea012e5c3*.

[13] Chirag Goyal. Part 5: Step by step guide to master nlp – word embedding and text vectorization. *https://www.analyticsvidhya.com/blog/2021/06/part-5-step-by-step-guide-to-master-nlp-text-vectorization-approaches/*, 2021.

[14] Aysel Laydin. 4 bag of words model in nlp. *Medium*, 2023. URL *https://ayselaydin.medium.com/4-bag-of-words-model-in-nlp-434cb38cdd1b*.

[15] techsmartfuture. Nlp vectorization techniques: A guide to text representation in nlp. *https://www.techsmartfuture.com/nlp-vectorization-techniques/*, 2023.

[16] Abid Ali Awan. Convert text documents to a tf-idf matrix with tfidfvectorizer. *KDnuggets*, 2022. URL *https://www.kdnuggets.com/2022/09/convert-text-documents-tfidf-matrix-tfidfvectorizer.html*.

[17] Joyce Xu. Topic modeling with lsa, plsa, lda lda2vec. *https://medium.com/nanonets/topic-modeling-with-lsa-psla-lda-and-lda2vec-555ff65b0b05*, 2018.

[18] Shivika K Bisen. How to perform topic modeling using mallet. *Analytics Vidhya*, 2020. URL *https://medium.com/analytics-vidhya/how-to-perform-topic-modeling-using-mallet-abc43916560f*.

[19] datascience904. Word embeddings. *https://datascience904.wordpress.com/2019/08/23/word-embeddings/*, 2019.

[20] Xiaonan Ji, Han-Wei Shen, Alan Ritter, Raghu Machiraju, and Po-Yin Yen. Visual exploration of neural document embedding in information retrieval: Semantics and feature selection. *IEEE Transactions on Visualization and Computer Graphics*, 25:2181–2192, 2019. URL *https://api.semanticscholar.org/CorpusID:84185382*.

[21] Rick Merritt. What is a transformer model? *https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/*, 2022.

[22] Peter Martigny. Nlp breakfast 2: The rise of language models. *Feedly Blog*, 2019. URL *https://feedly.com/engineering/posts/nlp-breakfast-2-the-rise-of-language-models*.

[23] Mahmoud Harmouch. 17 types of similarity and dissimilarity measures used in data science. *Towards Data Science*, 2021. URL *https://towardsdatascience.com/17-types-of-similarity-and-dissimilarity-measures-used-in-data-science-3eb914d2681*.

[24] Neri Van Otten. Top 7 ways to implement document text similarity in python: Nltk, scikit-learn, bert, roberta, fasttext and pytorch. *https://spotintelligence.com/2022/12/19/text-similarity-python/*, 2022.

[25] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn TensorFlow*. O'Reilly Media,Inc., 2017.

[26] Kavita Ganesan. What is machine learning model training? *https://www.opinosis-analytics.com/ai-glossary/machine-learning-model-training*, 2023.

[27] Allan Gonçalves. Machine learning: Is it a real deal? *Medium*, 2017. URL *https://medium.com/@allanvgoncalves/machine-learning-is-it-a-real-deal-7c258020788d*.

[28] Akashdeep Gupta. Integrating machine learning with devops and docker. *Analytics Vidhya*, 2020. URL *https://medium.com/analytics-vidhya/integrating-machine-learning-with-devops-and-docker-2812125083d0*.

[29] Jakub Czakon. Hyperparameter tuning in python: a complete guide. *https://neptune.ai/blog/hyperparameter-tuning-in-python-complete-guide*, 2023.

[30] The MathWorks Inc. Cross-validation. *https://es.mathworks.com/discovery/cross-validation.html*, n.d.

[31] GeeksforGeeks. Machine learning model evaluation. *https://www.geeksforgeeks.org/machine-learning-model-evaluation/*, 2023.

[32] Google AI. What is clustering? *https://developers.google.com/machine-learning/clustering/overview*, 2022.

[33] Google AI. Clustering algorithms. *https://developers.google.com/machine-learning/clustering/clustering-algorithms*, 2022.

[34] Clustering methods. *https://subscription.packtpub.com/book/data/9781789800265/5/ch05lvl1sec30/clustering-methods*.

[35] Computing4All. Evaluation of clustering algorithms: Measure the quality of a clustering outcome. *https://computing4all.com/courses/introductory-data-science/*, 2024.

[36] NLTK Community. Natural language toolkit. *https://www.nltk.org/*, 2024.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[38] NumPy Developers. Numpy developers. *https://numpy.org/doc/stable/index.html#*, 2024.

[39] Matplotlib Development Team. Matplotlib. *https://matplotlib.org/stable/index.html*, 2024.

[40] The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL *https://doi.org/10.5281/zenodo.3509134*.

[41] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.

[42] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[43] Project Jupyter. Project jupyter | home. *https://jupyter.org/*.

[44] pdfminer. pdfminer.six. *https://github.com/pdfminer/pdfminer.six*, 2022.

[45] Andreas Mueller. word_cloud. *https://github.com/amueller/word_cloud*, 2022.

[46] NLTK Community. Source code for nltk.stem.wordnet. *https://www.nltk.org/_modules/nltk/stem/wordnet.html*, 2023.

[47] NLTK Community. Source code for nltk.stem.porter. *https://www.nltk.org/_modules/nltk/stem/porter.html*, 2023.

[48] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6 (60):3021, 2021. doi: 10.21105/joss.03021. URL *https://doi.org/10.21105/joss.03021*.

[49] Dinesh Chandrasekaran. Optimal number of cluster identification with k-means algorithm using elbow method. *Blog by Dinesh Chandrasekaran*, 2019. URL *https://dchandra.com/machine%20learning/2018/12/16/K-means-Clustering-Algorithm-using-scikit-learn.html*.