Classification of Documents Using Graph-Based Features and KNN

Session: 2021 – 2025

Project Supervisor:

Mr. Waqas Ali

Submitted By:

Subhan Anjum	2021-CS-13
Hammad Younas	2021-CS-20

Department of Computer Science

University of Engineering and Technology Lahore Pakistan

Contents

List of Figures

1 Introduction

Effective document classification is essential for organizing and deriving valuable insights from massive amounts of textual data in the modern era of information overload. The emergence of graph theory in conjunction with machine learning methodologies has enabled novel approaches to document classification, providing opportunities for improved precision and efficacy. Inspired by landmark studies, this project explores the field of document categorization through a graph-based model, utilizing the capabilities of maximal common subgraphs (MCS) and the K-Nearest Neighbors (KNN) method.

2 Objective

This project's main goal is to use the concepts of graph theory and machine learning to create a reliable system for categorizing articles into predetermined subjects. The method attempts to identify naturally occurring links between terms inside documents by representing each document as a directed graph and extracting important aspects through common subgraph identification. The KNN approach is then used to accurately categorize topics by using the similarity measurements extracted from graph structures to classify test articles.

3 Background

In natural language processing (NLP) and information retrieval, document categorization is a basic job having applications in content recommendation, sentiment analysis, and topic modeling, among other disciplines. Textual data presents inherent contextual nuances and semantic linkages that are difficult for traditional vector-based algorithms to capture. Graph-based methods, on the other hand, encode both the syntactic and semantic relationships between terms, providing a more sophisticated depiction of document content.[1]

This research aims to improve the state-of-the-art by investigating the potential of maximal common subgraphs (MCS) as discriminative features for classification, building upon seminal papers in graph-based document classification. The study attempts to improve classification accuracy and reveal the underlying structure of textual data by using MCS to collect shared material across texts within the same topic.[2]

4 Data Description

The collection of materials used in this project was taken from a number of sectors and included subjects including travel, fashion and beauty, and diseases and symptoms. 45 documents in total were gathered, fifteen documents for each topic. Because each document is roughly 500 words long, there is a large corpus available for the classification model to be trained and tested.

4.1 Data Collection

The process of collecting data began with the utilization of Google Instant Data Scraper to capture website links. This allowed for the extraction of pertinent textual content from many sources. Afterwards, web scraping methods were utilized, utilizing BeautifulSoup (bs4) and requests libraries, in order to retrieve the textual content from the websites linked to the gathered links. The extracted papers were then divided into 36 training documents and 9 testing documents, respectively, for the training and testing sets.

```
import pandas as pd
import requests
from bs4 import BeautifulSoup
from docx import Document

def extract_text_from_url(url):
    try:
        response = requests.get(url)
        if response.status_code == 200:
            soup = BeautifulSoup(response.content, 'html.parser')
            # Extract all paragraphs from the webpage
            paragraphs = soup.find_all('p')
            # Combine text from paragraphs up to 500 words
            word_count = 0
            extracted_text = []
            for p in paragraphs:
                if word_count >= 500:
                    break
                text = p.get_text().strip()
                words = text.split()
                word_count += len(words)
                if word_count <= 500:
                    extracted_text.append(text)
            return ' '.join(extracted_text)
        else:
            print(f"Failed to retrieve content from URL: {url}")
            return None
    except Exception as e:
        print(f"Error occurred while extracting text: {str(e)}")
        return None

def main(input_file):
    df = pd.read_excel(input_file, header=None)  # Assuming links are in the first column
    for i, row in df.iterrows():
        link = row[0]
        extracted_text = extract_text_from_url(link)
        if extracted_text:
            # Create a new document
            doc = Document()
            doc.add_heading(f"Document {i+1}", 0)
            doc.add_paragraph(extracted_text)
            doc_file_name = f"Document {i+1}.docx"
            doc.save(doc_file_name)
            print(f"Text extracted from {link} and saved to {doc_file_name}")

if __name__ == "__main__":
    input_file = 'FashionBeautyData.xlsx'
    main(input_file)
```

4.2 Data Processing

The training data was preprocessed in a number of ways to improve its suitability for categorization before the model was trained:

1. **Tokenization:**
   ach document was tokenized, breaking down the text into individual tokens (words) to facilitate further processing.

2. **Stop Word Removal:**
   Common stop words, such as articles, conjunctions, and prepositions, were removed from the tokenized text to eliminate noise and reduce dimensionality.

```
import os
import docx
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import string
import nltk
from textblob import TextBlob

# Ensure NLTK stopwords are downloaded


# Function to remove stopwords and punctuation from text
def remove_stopwords_and_punctuation(text):
    stop_words = set(stopwords.words('english'))
    word_tokens = word_tokenize(text)
    filtered_text = [word for word in word_tokens if word.lower() not in stop_words and word not in
    string.punctuation]
    return ' '.join(filtered_text)

# Function to correct spellings using TextBlob
def correct_spelling(text):
    blob = TextBlob(text)
    corrected_text = str(blob.correct())
    return corrected_text

# Function to process .docx files in a directory and save processed files into a new folder
def process_files(input_folder, output_folder):
    # Create the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Process each .docx file in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith('.docx'):
            input_path = os.path.join(input_folder, filename)
            output_path = os.path.join(output_folder, filename)
            # Open the .docx file
            doc = docx.Document(input_path)

            # Extract text from the document
            text = ""
            for paragraph in doc.paragraphs:
                text += paragraph.text + "\n"

            # Remove stopwords and punctuation from the text
            processed_text = remove_stopwords_and_punctuation(text)

            # Correct spellings
            corrected_text = correct_spelling(processed_text)

            # Create a new document
            new_doc = docx.Document()
            new_doc.add_paragraph(corrected_text)

            # Save the new document
            new_doc.save(output_path)

# Specify input and output folders
input_folder = 'FashionBeauty'  # Relative path from the current working directory
output_folder = 'CFashionBeauty'  # Relative path from the current working directory

# Get the absolute paths
current_directory = os.path.dirname(os.path.abspath(__file__))
input_folder = os.path.join(current_directory, input_folder)
output_folder = os.path.join(current_directory, output_folder)

# Process files
process_files(input_folder, output_folder)
```

3. **Stemming:**
   The Porter Stemmer algorithm was applied to the tokenized text to reduce inflected words to their root forms, thereby standardizing variations and improving the effectiveness of feature extraction

```
import os
import docx
import re
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# Function to remove punctuation marks from text
def remove_punctuation(text):
    # Define the pattern to match punctuation marks
    punctuation_pattern = r'[^\w\s]'

    # Remove punctuation marks using regex
    text_without_punctuation = re.sub(punctuation_pattern, '', text)

    return text_without_punctuation
```

```
def preprocess_text(text):
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if word.lower() not in stop_words]
    porter = PorterStemmer()
    stemmed_tokens = [porter.stem(word) for word in filtered_tokens]
    return stemmed_tokens

# Function to process .docx files in a directory and save processed files into a new folder
def process_files(input_folder, output_folder):
    # Create the output folder if it doesn't exist
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Process each .docx file in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith('.docx') and not filename.startswith('~$'):  # Ignore temporary files
            input_path = os.path.join(input_folder, filename)
            output_path = os.path.join(output_folder, filename)

            # Open the .docx file
            try:
                doc = docx.Document(input_path)

                # Extract text from the document
                text = ""
                for paragraph in doc.paragraphs:
                    text += paragraph.text + "\n"

                # Remove punctuation marks
                processed_text = remove_punctuation(text)

                # Create a new document
                new_doc = docx.Document()
                new_doc.add_paragraph(processed_text)

                # Save the new document
                new_doc.save(output_path)
            except Exception as e:
                print(f"Error processing file '{filename}': {e}")


# Specify input and output folders
input_folder = 'CFashionBeauty'   # Relative path from the current working directory
output_folder = 'ProcessedCFashionBeauty'  # Relative path from the current working directory

# Get the absolute paths
current_directory = os.path.dirname(os.path.abspath(__file__))
input_folder = os.path.join(current_directory, input_folder)
output_folder = os.path.join(current_directory, output_folder)

# Process files
process_files(input_folder, output_folder)
```

4.3 Data Format

CSV (Comma-Separated Values) files were used to hold the preprocessed data, making management and manipulation simple. Two columns were present in every CSV file:

- **Content:**
  This column contained the processed text content of the documents after tokenization, stop word removal, and stemming.
- **Type:**
  The 'Type' column specified the topic category of each document, providing the ground truth labels required for training and evaluating the classification model.

```
import os
import csv
from docx import Document

# Function to read the content of a .docx file


def read_docx_content(file_path):
    doc = Document(file_path)
    content = ''
    for paragraph in doc.paragraphs:
        content += paragraph.text + '\n'
    return content.strip()

# Function to iterate over files in a folder, read their content, and write to CSV


def create_csv_from_folder(folder_path, output_file):
    # Open the CSV file in write mode
    with open(output_file, 'w', newline='', encoding='utf-8') as csvfile:
        # Define the headers for the CSV file
        fieldnames = ['Content', 'Type']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

        # Write the headers to the CSV file
        writer.writeheader()

        # Iterate over files in the folder
        for filename in os.listdir(folder_path):
            # Check if the file is a .docx file
            if filename.endswith('.docx'):
                # Read the content of the .docx file
                content = read_docx_content(
                    os.path.join(folder_path, filename))
                # Write the content and type to the CSV file
                writer.writerow({'Content': content, 'Type': 'Travel'})


# Specify the input folder containing the .docx files
input_folder = 'TestTravel'
```

```
# Specify the output CSV file
output_csv = 'output2.csv'

# Call the function to create the CSV file from the folder
create_csv_from_folder(input_folder, output_csv)

print(f"CSV file '{output_csv}' created successfully.")
```

We made sure that the input data was suitable for the creation of a reliable and accurate document classification system by carefully selecting and preparing the dataset. This laid the groundwork for the following phases of graph building, feature extraction, and classification.

5 Graph Construction

Within the project, each document's text content was used to create directed graphs. The relationships between terms, or words, inside the documents are shown by these graphs. These graphs were created using a number of steps:

1. **Tokenization:**
   Every document's text was tokenized into distinct words or phrases. By dissecting the text into its component elements, this phase facilitates analysis and processing of the material.

2. **Graph Representation:**
   Every document was shown as a directed graph, with nodes standing for distinct terms and edges for relationships between terms indicated by the terms' placement in the text. A directed edge would connect ”word1” and ”word2” in the graph, for instance, if ”word1” comes in the document before ”word2”.

3. **Visualization:**
   The directed graphs that were created were stored in files with the extension **.graphml**. Software like as Gephi is frequently used to visualize and analyze graphs in this format. Using **Gephi**, it is simple to see and explore the document structures by saving the graphs in the.graphml format.

4. **CSV Storage:**
   The directed graphs were additionally saved in the CSV (Comma-Separated Values) format. A single graph is represented by each CSV file, which has columns for the source node, target node, and edge weight. Further processing and analysis using tools like Python's pandas library or other data analysis applications is made possible by storing the graphs in CSV format.

```
# Function to construct directed graph from text content
def construct_graph(content):
    G = nx.DiGraph()
    terms = content.split()  # Split content into terms (words)
    for i in range(len(terms) - 1):
        current_term = terms[i]
        next_term = terms[i + 1]
        if not G.has_edge(current_term, next_term):
            G.add_edge(current_term, next_term, weight=1)  # Add edge between consecutive terms
        else:
            G[current_term][next_term]['weight'] += 1  # Increment edge weight if edge already exists
    return G
```

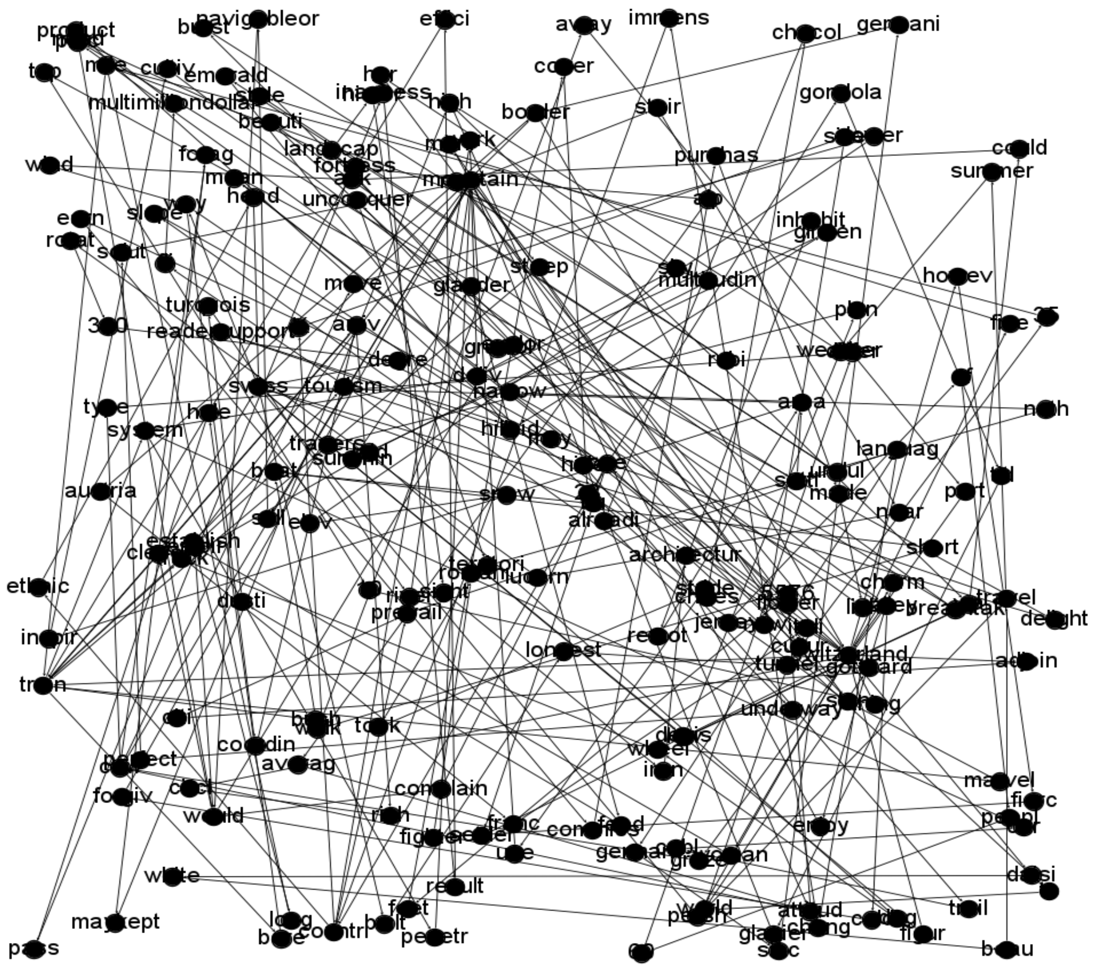

FIGURE 1: Visualization of Directed graph using **Gephi**

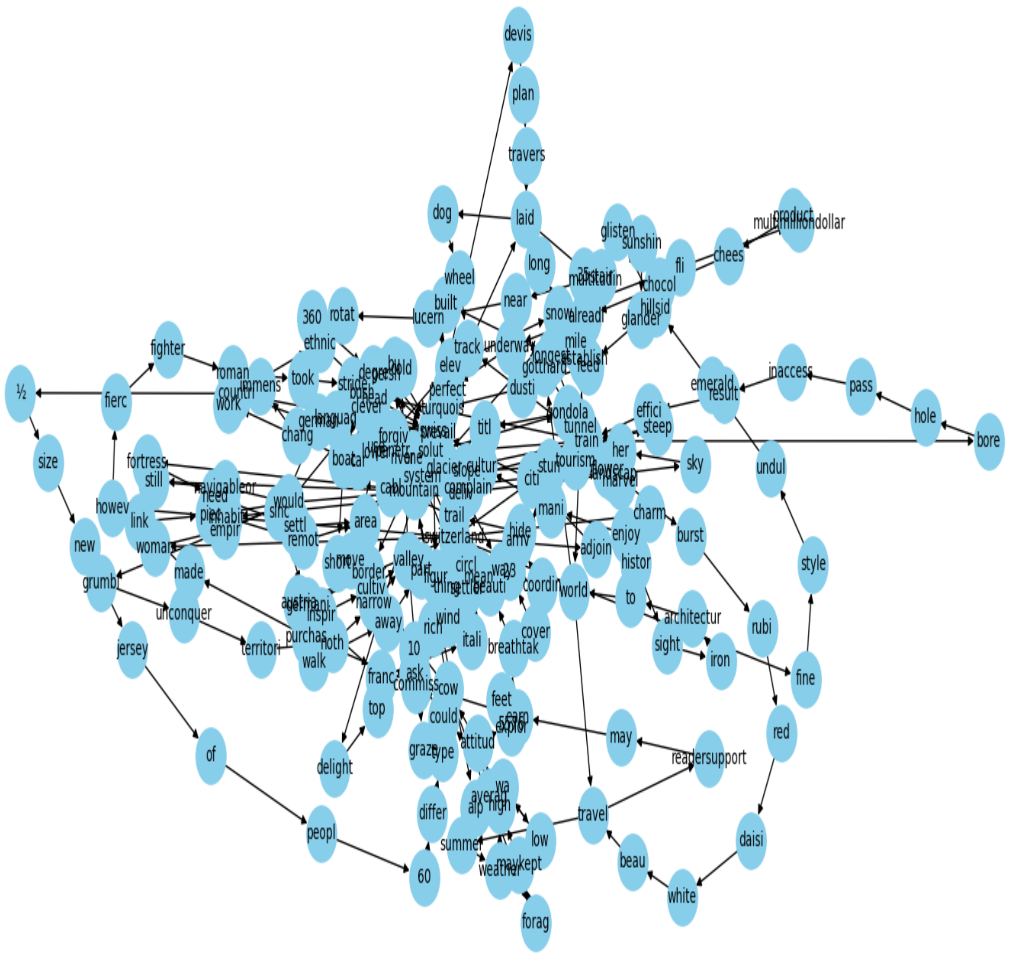

FIGURE 2: Visualization of Directed graph using library **networkx**

6 Feature Extraction via Common Subgraphs

We applied frequent subgraph mining algorithms to the document graphs in order to extract characteristics. Using these methods, common subgraphs within the training set graphs are found and used as classification features. These common subgraphs offer a potent visual representation of the data by capturing content that is shared between publications pertaining to the same subject.

```
# Function to compute the maximal common subgraph (MCS) between two graphs

def compute_mcs(G1, G2):
    # Convert graphs to edge sets
    edges1 = set(G1.edges())
    edges2 = set(G2.edges())
    # Compute the intersection of edges
    common_edges = edges1.intersection(edges2)
    # Create a new graph with common edges
    mcs_graph = nx.Graph(list(common_edges))
    return mcs_graph

def compute_distance(G1, G2):
    mcs_graph = compute_mcs(G1, G2)
    return -len(mcs_graph.edges())
```

7 Classification with KNN

Using a distance metric based on the maximal common subgraph (MCS) between document graphs, we constructed the KNN algorithm. This method determines how similar two graphs are by analyzing how much of their structure they have in common, as the MCS suggests. During the classification stage, test documents are categorized in the feature space produced by common subgraphs according to the majority class of their k-nearest neighbors.

```
    def knn_classify(test_graph, k):
    distances = []

    # Compute distance between test_graph and each training graph
    for train_id, train_graph in document_graphs.items():
        distance = compute_distance(test_graph, train_graph)
        distances.append((train_id, distance))

    # Sort distances in ascending order
    distances.sort(key=lambda x: x[1])

    # Get the k-nearest neighbors
    neighbors = distances[:k]

    # Get categories of the neighbors
    neighbor_categories = [data.loc[i, 'Type'] for i, _ in neighbors]

    # Find the majority class
    majority_class = Counter(neighbor_categories).most_common(1)[0][0]

    return majority_class
```

8 Evaluation

We evaluated our classification system's performance using a number of evaluation metrics, such as F1-score, accuracy, precision, and recall. Furthermore, we used a confusion matrix to illustrate the performance, which offers insights into the model's performance across several classes.
Based on our dataset of 45 documents out of which 36 were included in training set and 9 were used for testing we got:

F1 Score	0.75
Accuracy	77.79
Precision	0.87
Recall	0.78

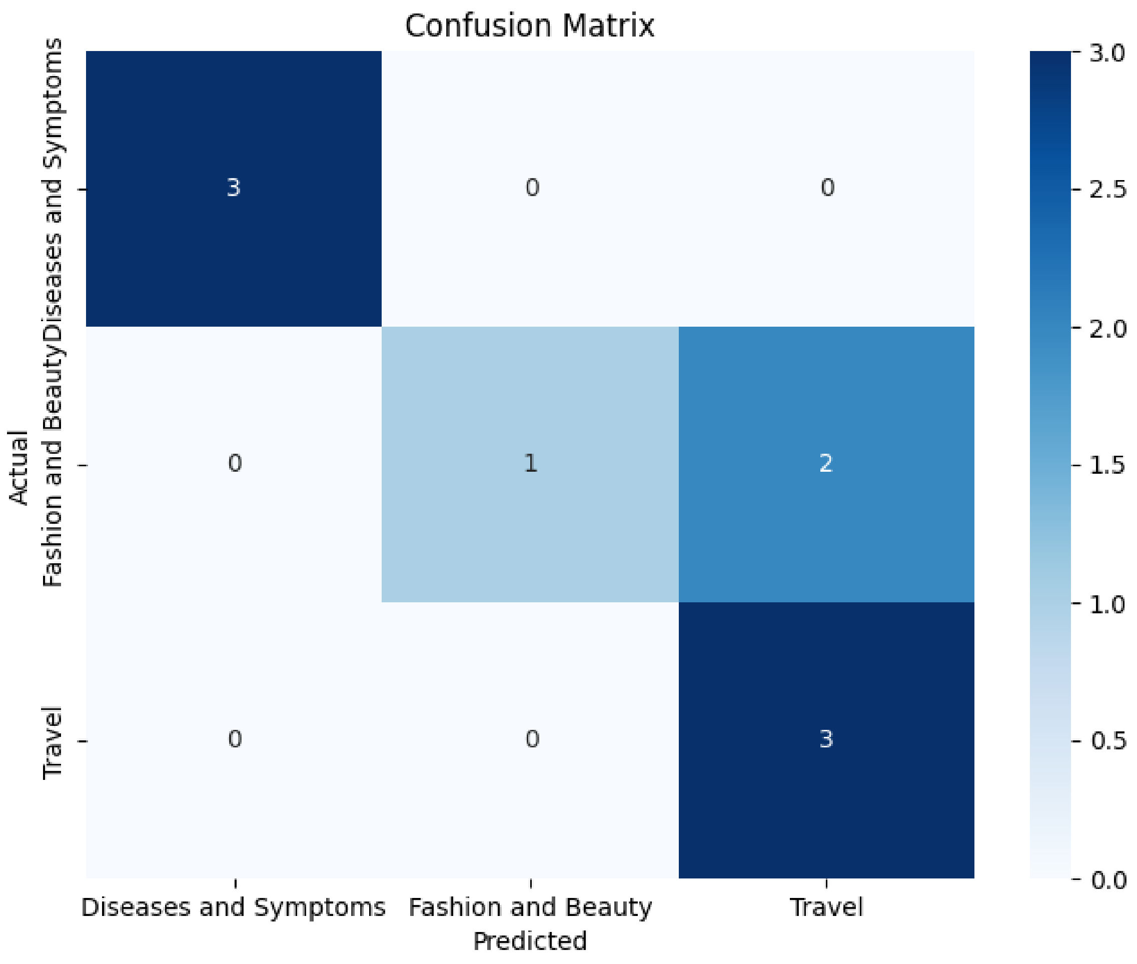

FIGURE 3: Confusion Matrix

8.1 Advantages of Graph based over Vector based Classification

- **Semantic Relationships**
  Semantic relationships between concepts are better captured by graph-based techniques than by vector-based representations. Better contextual comprehension and similarity evaluation are made possible by this.

- **Flexibility in Representation**
  Beyond basic co-occurrence, graphs may represent complicated relationships, enabling richer and more complex data representations.

- **Handling of Unstructured Data**
  Graph-based models are more suited for a variety of document formats, including text, photos, and social networks, because they can manage unstructured data more organically.

- **Transfer Learning**
  Graph-based representations are easily adjustable and transferable to other activities or domains, allowing for enhanced performance through the use of learnt structures and linkages.

- **Scalability**
  Graph-based techniques have computational advantages over conventional vector-based techniques since they can scale to big datasets and complicated networks with ease thanks to excellent algorithms and data structures.

9 Conclusion

In this study, we investigated graph-based techniques for classifying documents, making use of semantic connections to improve understanding and accuracy. Our findings showed that graph-based techniques outperform conventional vector-based techniques, exhibiting enhanced robustness and performance for a wide range of document types. A versatile and understandable framework for evaluating textual data is provided by graph-based models, which have potential uses in a wide range of fields. Going forward, more studies in this field may open up new avenues for applying graph-based methods in practical settings.
Here is the Github Url to Our project Classification of Documents Using Graph-Based Features and KNN

Bibliography

[1] Classification of Web Documents Using a Graph Model https://www.researchgate.net/publication/4033384_Classification_of_Web_Documents_Using_a_Graph_Model#full-text

[2] A graph distance metric based on the maximal common subgraph https://www.sciencedirect.com/science/article/pii/S0167865597001797