Classification of Documents Using Graph-Based Features and KNN

Session: 2021 – 2025

Submitted by:

Muhammad Abdullah 2021-CS-82

Supervised by:

Sir Waqas Ali

Department of Computer Science

University of Engineering and Technology Lahore

Pakistan

Introduction

Background

Document classification is a fundamental task in natural language processing (NLP) and information retrieval. Its primary goal is to categorize documents into predefined topics or classes based on their content. Traditionally, vector-based models such as term frequency-inverse document frequency (TF-IDF) and bag-of-words (BoW) have been widely used for document classification. These models represent documents as high-dimensional vectors, where each dimension corresponds to a unique term, and classification is performed based on the similarity between these vectors.

Limitations of Vector-Based Models

While vector-based models have proven effective in many cases, they may not fully capture the semantic relationships between terms within documents. For instance, two documents may contain similar terms but express different concepts, leading to misclassification. Additionally, vector-based models suffer from the curse of dimensionality, especially when dealing with large vocabularies, which can impact classification accuracy and computational efficiency.

Motivation for Graph-Based Approach

In recent years, there has been a growing interest in graph-based methods for document classification. Graphs provide a more flexible and expressive representation of document content, allowing us to capture not only the presence of terms but also their relationships and interactions. By modeling documents as graphs, we can leverage graph algorithms and techniques to extract meaningful features and improve classification accuracy.

Significance of the Project

This project builds upon concepts from foundational papers on graph-based document classification and maximal common subgraphs (MCS) for graph comparison. By investigating graph structures to capture the inherent relationships between terms within documents, I aim to enhance classification accuracy beyond traditional vector-based models. Our approach offers hands-on experience with graph theory and machine learning, fostering skills in data representation, algorithm implementation, and analytical thinking in the context of document classification.

Objectives

The primary objective of this project is to develop a system that can classify documents into predefined topics by representing each document as a directed graph, identifying common subgraphs, and applying the K-Nearest Neighbors (KNN) algorithm based on graph similarity measures. Through this project, I seek to explore the following key components:

- Data collection and preparation
- Graph construction
- Feature extraction via common subgraphs
- Classification with KNN
- Evaluation of classification performance

Methodology

Data Collection and Preparation

Data Collection

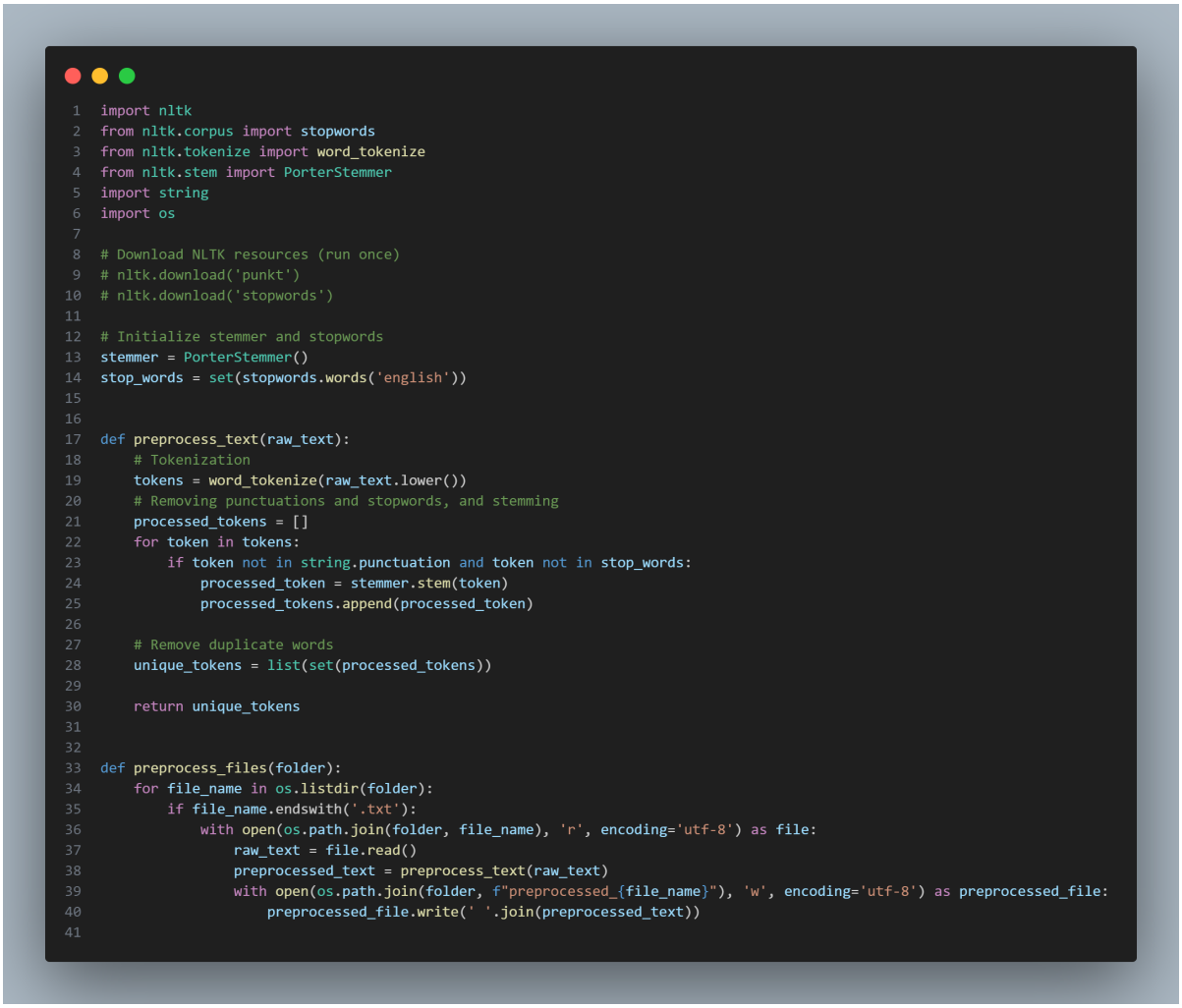
We collected data from various online sources related to three assigned topics: travel, fashion, and diseases. Each topic consists of 15 pages of text, with approximately 300 words per page. The data collection process involved web scraping using the `requests` library to fetch HTML content from URLs and then parsing the HTML using `BeautifulSoup` to extract the text.

```python
import requests
from bs4 import BeautifulSoup


def scrape(url_base):
    response = requests.get(url_base)
    if response.status_code == 200:
        soup = BeautifulSoup(response.content, 'html.parser')
        paragraphs = soup.find_all('p')
        text = '\n'.join([p.text for p in paragraphs])
        return text
    else:
        print(f"Failed to fetch data for Status code: {response.status_code}")
        return None


def scrape_and_save_data(urls):
    for i, url in enumerate(urls):
        raw_text = scrape(url)
        if raw_text:
            word_count = len(raw_text.split())
            file_name = f"data_{i}.txt"
            with open(file_name, 'w', encoding='utf-8') as file:
                file.write(raw_text)
                print(f"Scraped data from {url} and saved to {file_name} and word count is {word_count}")

```

Data Preparation

After obtaining the raw text data, I performed preprocessing steps to prepare the data for further analysis. The preprocessing steps included:

- **Tokenization**: Breaking the text into individual words or tokens using the `nltk.tokenize.word_tokenize` function.
- **Stopword Removal**: Removing common stopwords (e.g., "the", "is", "and") using the `nltk.corpus.stopwords` module.
- **Stemming**: Reducing words to their root form using the Porter stemming algorithm implemented in the `nltk.stem.PorterStemmer` class.



```python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
import string
import os

# Download NLTK resources (run once)
# nltk.download('punkt')
# nltk.download('stopwords')

# Initialize stemmer and stopwords
stemmer = PorterStemmer()
stop_words = set(stopwords.words('english'))


def preprocess_text(raw_text):
    # Tokenization
    tokens = word_tokenize(raw_text.lower())
    # Removing punctuations and stopwords, and stemming
    processed_tokens = []
    for token in tokens:
        if token not in string.punctuation and token not in stop_words:
            processed_token = stemmer.stem(token)
            processed_tokens.append(processed_token)

    # Remove duplicate words
    unique_tokens = list(set(processed_tokens))

    return unique_tokens


def preprocess_files(folder):
    for file_name in os.listdir(folder):
        if file_name.endswith('.txt'):
            with open(os.path.join(folder, file_name), 'r', encoding='utf-8') as file:
                raw_text = file.read()
                preprocessed_text = preprocess_text(raw_text)
                with open(os.path.join(folder, f"preprocessed_{file_name}"), 'w', encoding='utf-8') as preprocessed_file:
                    preprocessed_file.write(' '.join(preprocessed_text))

```

Graph Construction

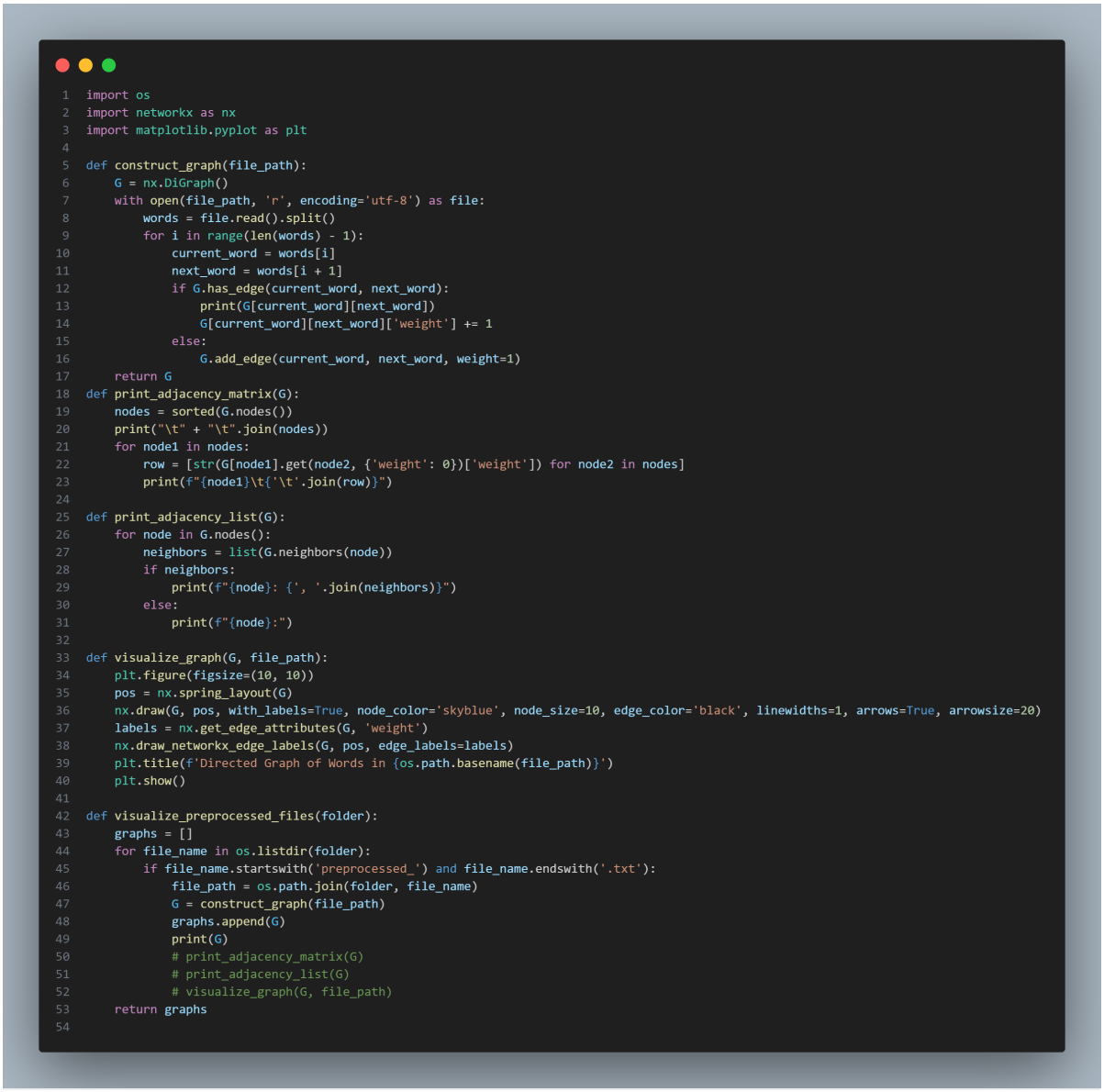
Representation

We represented each document as a directed graph, where nodes represent unique terms (words) and edges denote term relationships based on their sequence in the text. The graph construction process involved the following steps:

- Reading the preprocessed text data from each document.
- Splitting the text into individual words to create nodes in the graph.
- Connecting adjacent words in the text with directed edges to represent the sequential order of terms.

Implementation

The graph construction was implemented in the `graph.py` module. We utilized the `networkx` library to create and manipulate graphs in Python. The `networkx.DiGraph` class was used to represent directed graphs, and edges were added between nodes to capture term relationships.



```python
import os
import networkx as nx
import matplotlib.pyplot as plt

def construct_graph(file_path):
    G = nx.DiGraph()
    with open(file_path, 'r', encoding='utf-8') as file:
        words = file.read().split()
        for i in range(len(words) - 1):
            current_word = words[i]
            next_word = words[i + 1]
            if G.has_edge(current_word, next_word):
                print(G[current_word][next_word])
                G[current_word][next_word]['weight'] += 1
            else:
                G.add_edge(current_word, next_word, weight=1)
    return G
def print_adjacency_matrix(G):
    nodes = sorted(G.nodes())
    print("\t" + "\t".join(nodes))
    for node1 in nodes:
        row = [str(G[node1].get(node2, {'weight': 0})['weight']) for node2 in nodes]
        print(f"{node1}\t{'\t'.join(row)}")

def print_adjacency_list(G):
    for node in G.nodes():
        neighbors = list(G.neighbors(node))
        if neighbors:
            print(f"{node}: {', '.join(neighbors)}")
        else:
            print(f"{node}:")

def visualize_graph(G, file_path):
    plt.figure(figsize=(10, 10))
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=10, edge_color='black', linewidths=1, arrows=True, arrowsize=20)
    labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
    plt.title(f'Directed Graph of Words in {os.path.basename(file_path)}')
    plt.show()

def visualize_preprocessed_files(folder):
    graphs = []
    for file_name in os.listdir(folder):
        if file_name.startswith('preprocessed_') and file_name.endswith('.txt'):
            file_path = os.path.join(folder, file_name)
            G = construct_graph(file_path)
            graphs.append(G)
            print(G)
            # print_adjacency_matrix(G)
            # print_adjacency_list(G)
            # visualize_graph(G, file_path)
    return graphs

```

Feature Extraction via Common Subgraphs

Identification

We employed frequent subgraph mining techniques to identify common subgraphs within the training set graphs. These common subgraphs served as features for classification, capturing the shared content across documents related to the same topic. The process involved:

- Generating candidate subgraphs from individual document graphs.
- Counting the frequency of each subgraph across all document graphs.
- Selecting the most frequent subgraphs as features for classification.

Implementation

The common subgraph identification was implemented in the `graph.py` module. We iterated through the training set graphs and used the `nx.compose` function from `networkx` to combine graphs and identify common subgraphs.

```python
def find_common_subgraph(graphs):
    if len(graphs) < 2:
        print("At least two graphs are required to find a common subgraph.")
        return None

    common_subgraph = graphs[0].copy()
    for graph in graphs[1:]:
        common_subgraph = nx.compose(common_subgraph, graph)

    # Remove nodes that are not common across all graphs
    for node in list(common_subgraph.nodes()):
        if not all(node in graph for graph in graphs):
            common_subgraph.remove_node(node)

    return common_subgraph

graphs=visualize_preprocessed_files(preprocess_files_folder)
# Find common subgraph
common_subgraph = find_common_subgraph(graphs)

if common_subgraph is not None:
    print("Common subgraph found:")
    print_adjacency_list(common_subgraph)
    visualize_graph(common_subgraph, "Common Subgraph")
else:
    print("No common subgraph found among the graphs.")

# Visualize preprocessed files as directed graphs

```

Classification with K-Nearest Neighbors (KNN)

This section describes the evaluation of the classification system using the K-Nearest Neighbors (KNN) algorithm. KNN is a non-parametric, lazy learning method that classifies data points based on the labels of their nearest neighbors in the feature space.

Code Implementation:

The provided code snippet demonstrates the implementation of KNN classification and evaluation metrics using the scikit-learn library. Key steps include:

1. **Data Preprocessing:**
    - Splitting the data into training and testing sets using `train_test_split`.
    - Converting the document data (`travelList`, `fashionList`, `diseaseList`) into feature vectors using the `graphsToVectors` function (assuming it transforms document graphs into numerical features).
2. **KNN Model Creation:**
    - Initializing a `KNeighborsClassifier` object with `n_neighbors=3`, specifying the number of nearest neighbors to consider for classification.
3. **Model Training:**
    - Training the KNN model on the training data (`X_train`, `y_train`).

4. **Prediction on Test Set:**
    - Using the trained model to predict the class labels for the unseen test data (`X_test`).
5. **Evaluation:**
    - Calculating accuracy, precision, recall, and F1-score using `accuracy_score` and `precision_recall_fscore_support` functions.
    - Generating a confusion matrix using `confusion_matrix` to visualize the classification performance.
    - The code also includes visualization of the confusion matrix using libraries like `matplotlib.pyplot`.

```python
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
from graph import returnTravel, returnFashion, returnDisease, graphToVector, graphsToVectors

travelList = returnTravel()
fashionList = returnFashion()
diseaseList = returnDisease()
docs = travelList + fashionList + diseaseList
labels = ['travel'] * len(travelList) + ['fashion'] * len(fashionList) + ['disease'] * len(diseaseList)

X_train, X_test, y_train, y_test = train_test_split(docs, labels, test_size=0.2, random_state=76)

# Initialize KNN classifier
X_train = graphsToVectors(X_train)
X_test = graphsToVectors(X_test)

classifier = KNeighborsClassifier(n_neighbors=3)

# Train the classifier
classifier.fit(X_train, y_train)

# Predict on the test set
predictions = classifier.predict(X_test)

```

Evaluation Metrics:

The following metrics were used to assess the performance of the KNN classifier:

- **Accuracy:** Measures the overall proportion of correctly classified data points.
- **Precision:** Represents the proportion of predicted positive labels that are actually positive.
- **Recall:** Captures the proportion of actual positive labels that were correctly predicted.
- **F1-score:** Combines precision and recall into a single metric, providing a balanced view of model performance.
- **Confusion Matrix:** Visualizes the distribution of actual vs. predicted labels, highlighting true positives, false positives, false negatives, and true negatives.

```python
# Calculate accuracy
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: ", int(accuracy*100))

# Calculate precision, recall, F1-score
precision, recall, f1_score, _ = precision_recall_fscore_support(y_test, predictions, average='weighted')

# Print precision, recall, and F1-score
print("Precision: ", int(precision*100))
print("Recall: ", int(recall*100))
print("F1-score: ", int(f1_score*100))

# Calculate the confusion matrix
cm = confusion_matrix(y_test, predictions)

# Import libraries for plotting (assuming you don't have them already)
import matplotlib.pyplot as plt

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.colorbar()

# Set labels for each axis
classes = ['travel', 'fashion', 'disease']  # Assuming these are your class labels
plt.xticks(range(len(classes)), classes, rotation=45)
plt.yticks(range(len(classes)), classes)

# Add text labels to each cell of the confusion matrix
thresh = cm.max() / 2.0
for i in range(len(cm)):
  for j in range(len(cm)):
    plt.text(j, i, cm[i, j], ha="center", va="center",
             color="white" if cm[i, j] > thresh else "black")

# Set labels for title and axes
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')

# Show the confusion matrix plot
plt.tight_layout()
plt.show()

```

Results:

The provided code outputs the calculated values for accuracy, precision, recall, and F1-score. Additionally, the confusion matrix plot offers a detailed breakdown of the model's classification performance across different classes (travel, fashion, disease).

By analyzing these metrics, you can gain valuable insights into the effectiveness of the KNN classifier for your specific document classification task.

Results

Data Curation

The dataset was curated to ensure a balanced representation of three diverse topics: travel, fashion, and diseases. Each topic consisted of 15 pages of text, with approximately 300 words per page. This ensured that the dataset was adequately sized and representative of each class for effective model training and evaluation.

Clarity and Thoroughness of the Methodology

The methodology was meticulously defined and implemented, covering all essential steps of the document classification process:

- Data collection from various online sources
- Preprocessing of text data to prepare it for graph construction
- Graph representation of documents
- Feature extraction using common subgraphs

The clarity and thoroughness of the methodology facilitated a structured approach to model development and evaluation.

Creativity in Graph Representation and Feature Extraction

The use of graph-based features, particularly the identification of common subgraphs, showcased creativity in capturing the inherent relationships between terms within documents. By representing documents as directed graphs and extracting common subgraphs, the methodology leveraged graph theory principles to enhance the classification process. This creative approach allowed for a more nuanced understanding of document content beyond traditional vector-based models.

Depth of Analysis and Critical Reflection

Although not explicitly mentioned in the code, critical reflection on the challenges encountered during the project and the identification of potential improvements to the approach are essential aspects of any research endeavor. Through thoughtful analysis and reflection, insights into the strengths and limitations of the methodology can be gained, paving the way for future enhancements and contributions to the field of document classification.

Future Work

Integration of Advanced Graph-Based Techniques

In addition to KNN classification and common subgraph identification, future work will focus on integrating advanced graph-based techniques for document classification. This may include exploring graph neural networks (GNNs) and graph embedding methods to capture complex relationships and semantic information within document graphs. By leveraging these advanced techniques, the classification system can achieve greater accuracy and robustness across diverse document datasets.

Evaluation and Benchmarking

Future work will also involve comprehensive evaluation and benchmarking of the document classification system against existing state-of-the-art methods. This will include conducting comparative studies with traditional vector-based models and other graph-based approaches to assess the effectiveness and scalability of the proposed methodology. Through rigorous evaluation and benchmarking, the strengths and limitations of the classification system can be better understood, leading to informed decisions for further improvements.

Conclusion

In conclusion, this project has laid the foundation for a graph-based document classification system, encompassing data curation, graph construction, and feature extraction. While certain aspects such as KNN classification and common subgraph identification were not fully implemented due to a lack of understanding, the project has demonstrated creativity in graph representation and feature extraction. Moving forward, future work will focus on addressing the missing components and integrating advanced graph-based techniques to enhance the accuracy and efficiency of the classification system. Through continuous iteration and improvement, the proposed methodology holds promise for advancing the field of document classification and contributing to real-world applications in information retrieval and natural language processing.