REPORT
JOB CLASSIFICATION FOR INFORMATION TECHNOLOGY & FIND JOB SIMILARITIES USING NLP & ML TECHNIQUES

GROUP B

C0886190 Ahmed Abdulrahim

C0887143 Bimsara Siman Meru Pathiranage

C0864054 Efemena Theophilus Edoja

C0885235 Simranjeet Kaur

C0885177 Himanshu

Abstract

This project aims to develop a job classification for IT jobs and similarity tools using Natural Language Processing (NLP) and Machine Learning (ML) techniques. The tool is designed to categorize job descriptions for information technology and identify similar job titles based on linguistic similarity. The project utilizes a Random Forest model for job description classification and a Doc2Vec model for job title similarity search. The system is implemented as an interactive user interface using Python, ipywidgets, and Jupyter Notebooks, making complex NLP and ML features accessible to HR professionals and job seekers.

Table of Contents

1. Introduction

In today's dynamic job market, the realm of Information Technology (IT) stands as a cornerstone of innovation and opportunity. Yet, navigating the vast landscape of IT job titles poses a unique challenge for both job seekers and HR professionals. Recognizing this complexity, our project focuses on developing a robust job classification and similarity tool specifically tailored for IT roles, powered by advanced Natural Language Processing (NLP) and Machine Learning (ML) techniques.

This report serves as a comprehensive exploration of our project, delving into its development, implementation, and potential impact within the IT industry. By harnessing cutting-edge NLP algorithms, we extract nuanced insights from IT job descriptions, enabling precise classification and categorization of roles. Additionally, our ML models leverage semantic understanding to identify similar job titles, revolutionizing the accuracy of job matching within the IT domain.

Throughout this report, we provide detailed insights into the architecture of our tool, elucidating the methodologies employed in data processing, model training, and evaluation, all within the context of IT job titles. Furthermore, we explore the practical implications of our tool, showcasing its potential to streamline the recruitment process for IT professionals and organizations alike.

As we embark on this journey to optimize the IT job search experience, our project aims to bridge the gap between talent and opportunity, fostering efficiency, reducing friction, and ultimately driving success in the ever-evolving landscape of IT careers.

2. Data Collection and Preprocessing

The project utilizes a dataset of job postings collected from Kaggle. The data preprocessing steps include cleaning, transformation, and feature engineering. The cleaning process involves removing irrelevant information, handling missing values, and correcting inconsistencies. The transformation process includes tokenization, stemming, and lemmatization. Feature engineering involves creating new features that capture important aspects of the job descriptions, such as the frequency of specific skills or requirements.

2.1 Dataset Overview

2.1.1 Data Understanding

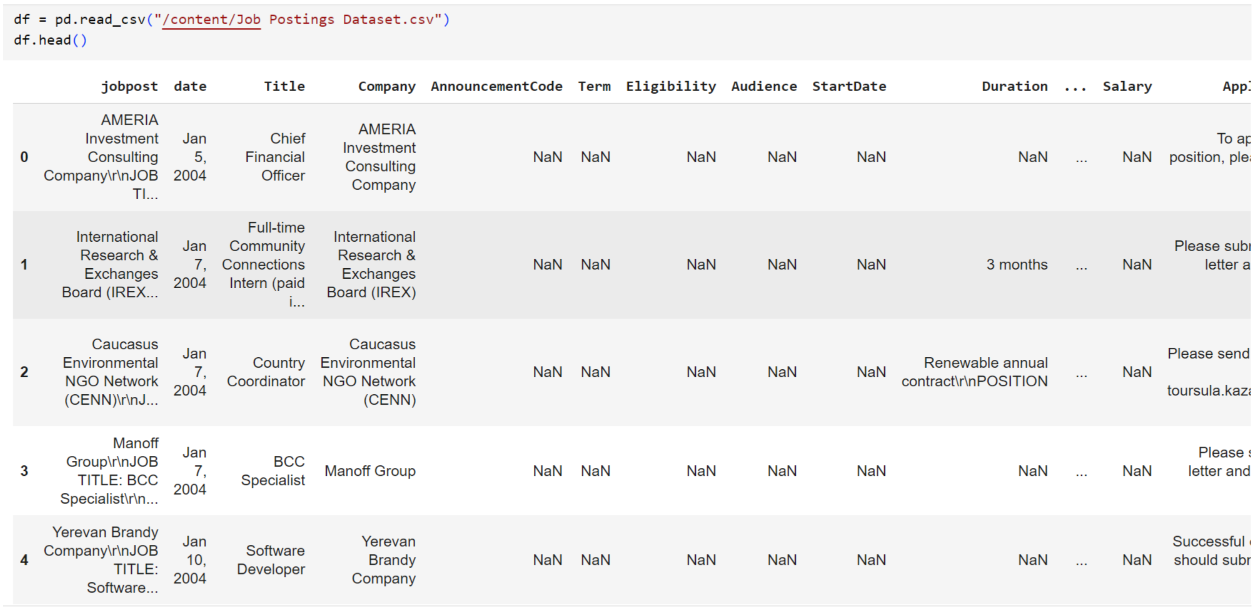
The "Armenian Online Job Postings" dataset is the foundation of our project, providing a comprehensive source of job postings in Armenia from 2004 to 2015. This dataset offers valuable insights into the evolution of the job market, reflecting the dynamic nature of industries and professions over a decade.

Dataset URL : Armenian Online Job Postings

Our primary objective in this phase is to explore the job postings dataset in-depth, understand its structure, and extract meaningful insights. By examining key columns, we aim to uncover the dataset's composition and identify critical features that significantly impact the job matching process. This exploration phase is essential for establishing a solid foundation for future analyses.

2.2 Dataset Information

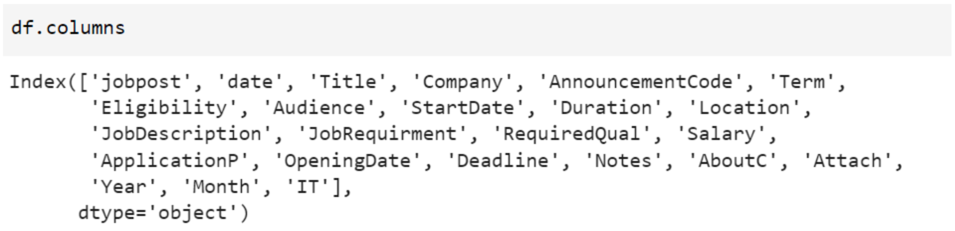
- `df.head()` to show the initial few rows.



```
df = pd.read_csv("/content/Job Postings Dataset.csv")
df.head()
```

	jobpost	date	Title	Company	AnnouncementCode	Term	Eligibility	Audience	StartDate	Duration	...	Salary	App
0	AMERIA Investment Consulting Company\r\nJOB TI...	Jan 5, 2004	Chief Financial Officer	AMERIA Investment Consulting Company	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	To ap position, ple
1	International Research & Exchanges Board (IREX...	Jan 7, 2004	Full-time Community Connections Intern (paid i...	International Research & Exchanges Board (IREX)	NaN	NaN	NaN	NaN	NaN	3 months	...	NaN	Please subr letter a
2	Caucasus Environmental NGO Network (CENN)\r\nJ...	Jan 7, 2004	Country Coordinator	Caucasus Environmental NGO Network (CENN)	NaN	NaN	NaN	NaN	NaN	Renewable annual contract\r\nPOSITION	...	NaN	Please send toursula.kaz
3	Manoff Group\r\nJOB TITLE: BCC Specialist\r\n...	Jan 7, 2004	BCC Specialist	Manoff Group	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	Please s letter and
4	Yerevan Brandy Company\r\nJOB TITLE: Software...	Jan 10, 2004	Software Developer	Yerevan Brandy Company	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	Successful should subr

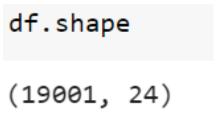
- `df.columns` to obtain the names of the columns.



```
df.columns
```

```
Index(['jobpost', 'date', 'Title', 'Company', 'AnnouncementCode', 'Term',
       'Eligibility', 'Audience', 'StartDate', 'Duration', 'Location',
       'JobDescription', 'JobRequirment', 'RequiredQual', 'Salary',
       'ApplicationP', 'OpeningDate', 'Deadline', 'Notes', 'AboutC', 'Attach',
       'Year', 'Month', 'IT'],
      dtype='object')
```

- `df.shape` to verify the quantity of columns and rows.



```
df.shape
```

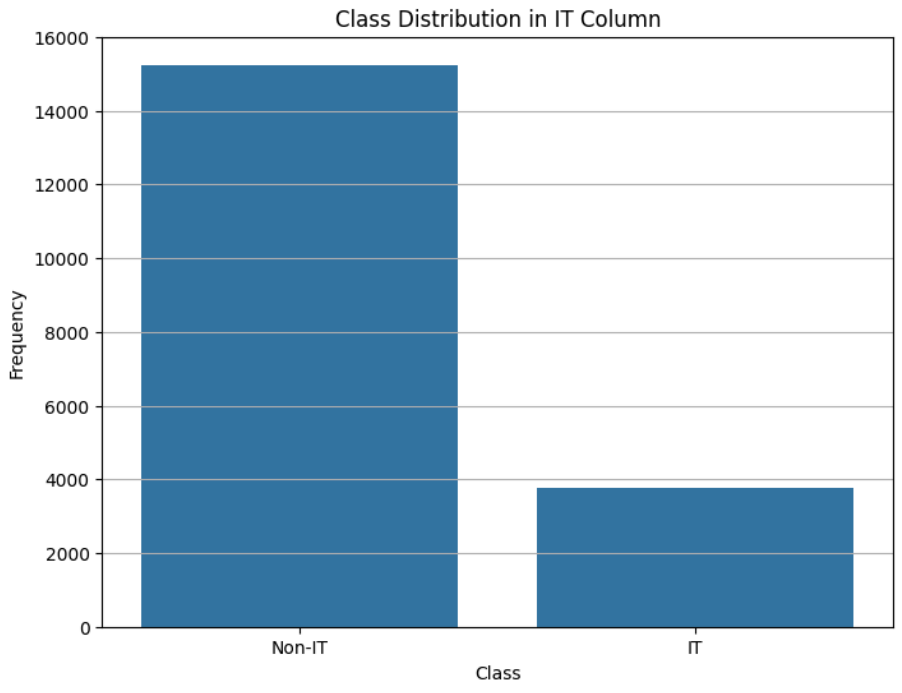
```
(19001, 24)
```

- `df.info()` to comprehend the missing numbers and data kinds.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19001 entries, 0 to 19000
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   jobpost           19001 non-null  object
 1   date              19001 non-null  object
 2   Title             18973 non-null  object
 3   Company           18994 non-null  object
 4   AnnouncementCode  1208 non-null   object
 5   Term              7676 non-null   object
 6   Eligibility       4930 non-null   object
 7   Audience          640 non-null    object
 8   StartDate         9675 non-null   object
 9   Duration          10798 non-null  object
 10  Location          18969 non-null  object
 11  JobDescription    15109 non-null  object
 12  JobRequirment     16479 non-null  object
 13  RequiredQual      18517 non-null  object
 14  Salary            9622 non-null   object
 15  ApplicationP      18941 non-null  object
 16  OpeningDate       18295 non-null  object
 17  Deadline          18936 non-null  object
 18  Notes             2211 non-null   object
 19  AboutC            12470 non-null  object
 20  Attach            1559 non-null   object
 21  Year              19001 non-null  int64
 22  Month             19001 non-null  int64
 23  IT                19001 non-null  bool
dtypes: bool(1), int64(2), object(21)
```

- Visualizing the class distribution in IT columns



2.3 Data Preprocessing

Once the dataset's structure is comprehended, the focus shifts to preparing the data for thorough analysis. Data cleaning involves a series of actions, including the removal of unnecessary columns, addressing missing values, and standardizing column names. By executing these steps, we ensure that the dataset is streamlined, free from redundant information, and possesses consistent formatting, laying the groundwork for more sophisticated processing and analysis.

2.3.1 Data Cleaning

- Removed duplicate columns in order to generate `df_clean`.

```
#removing duplicate jobposts based on title and post
jobs = jobs.drop_duplicates(['jobpost', 'Title'])
```

- Eliminated rows that had null values in crucial fields such as "RequiredQualifications," "JobDescription," "Company," and "Title."

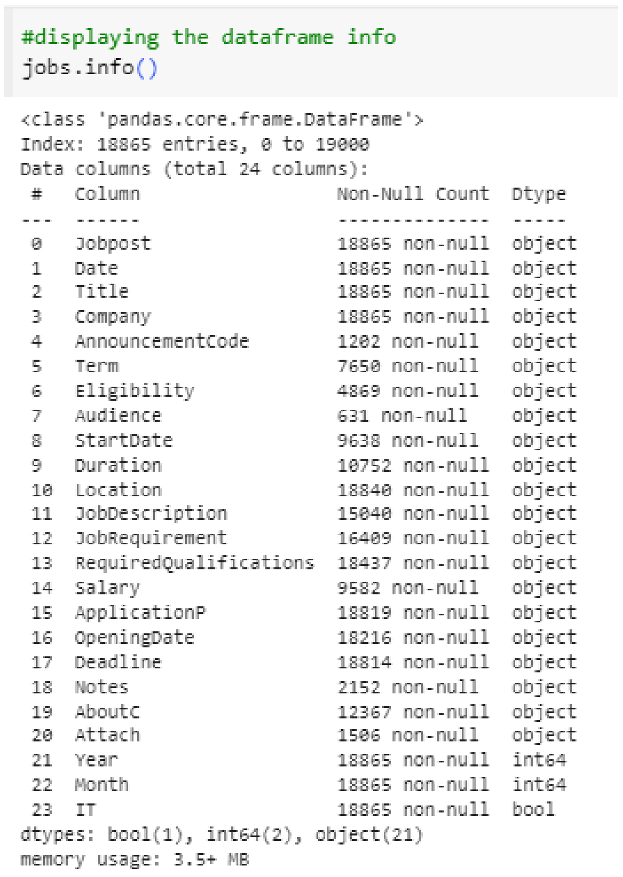
```
#removing records with null titles
jobs = jobs[jobs.Title.notna()]
```

```
jobs.shape
```

```
(18865, 24)
```

- For understanding, certain columns got their names changed and cleaned up.

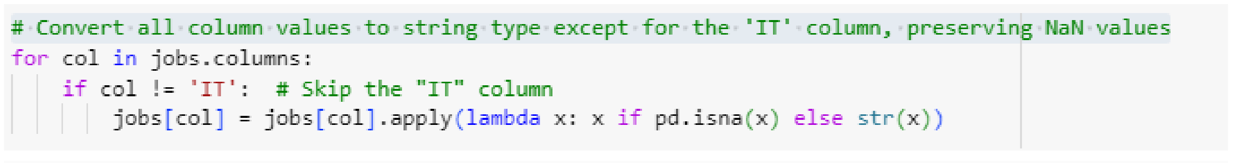
```
# Remove leading and trailing spaces from column names
jobs.columns = jobs.columns.str.strip()
# Rename specific columns for clarity and consistency
jobs.rename(columns={'date': 'Date', 'jobpost': 'Jobpost', 'RequiredQual': 'RequiredQualifications', 'JobRequirment': 'JobRequirement'}, inplace=True)
```



```
#displaying the dataframe info
jobs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 18865 entries, 0 to 19000
Data columns (total 24 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   Jobpost                 18865 non-null  object
 1   Date                    18865 non-null  object
 2   Title                   18865 non-null  object
 3   Company                 18865 non-null  object
 4   AnnouncementCode        1202 non-null   object
 5   Term                    7650 non-null   object
 6   Eligibility             4869 non-null   object
 7   Audience                631 non-null    object
 8   StartDate               9638 non-null   object
 9   Duration                10752 non-null  object
 10  Location                18840 non-null  object
 11  JobDescription          15040 non-null  object
 12  JobRequirement          16409 non-null  object
 13  RequiredQualifications  18437 non-null  object
 14  Salary                  9582 non-null   object
 15  ApplicationP            18819 non-null  object
 16  OpeningDate             18216 non-null  object
 17  Deadline                18814 non-null  object
 18  Notes                   2152 non-null   object
 19  AboutC                  12367 non-null  object
 20  Attach                  1506 non-null   object
 21  Year                    18865 non-null  int64
 22  Month                   18865 non-null  int64
 23  IT                      18865 non-null  bool
dtypes: bool(1), int64(2), object(21)
memory usage: 3.5+ MB
```

- Convert all column values to string type except for the 'IT' column, preserving NaN values



```
# Convert all column values to string type except for the 'IT' column, preserving NaN values
for col in jobs.columns:
    if col != 'IT':  # Skip the "IT" column
        jobs[col] = jobs[col].apply(lambda x: x if pd.isna(x) else str(x))
```

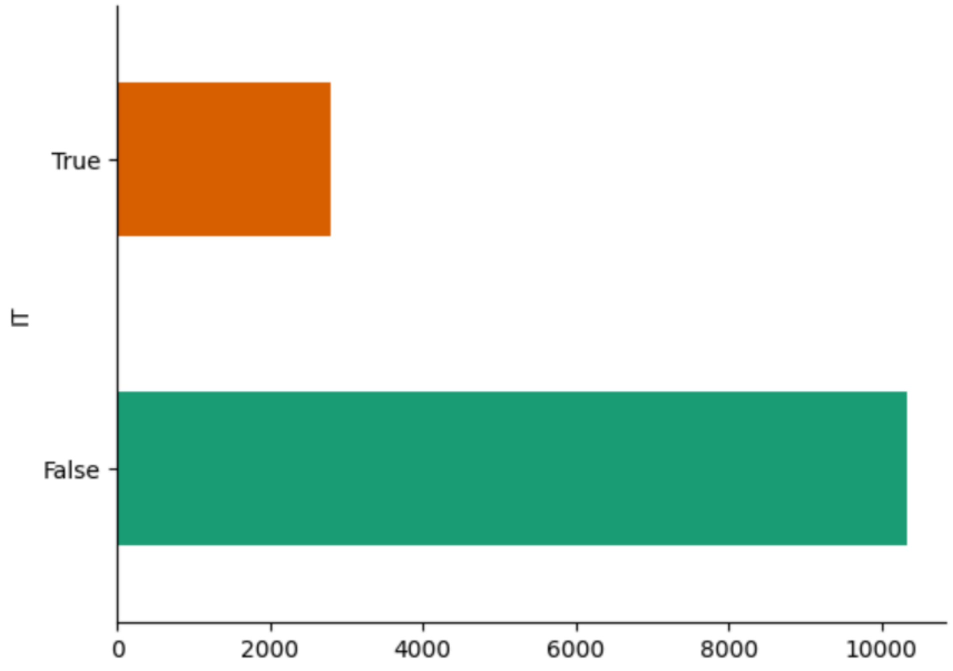
2.4 Exploratory Data Analysis (EDA)

2.4.1 Analysis of IT Jobs

IT Job Analysis: Examine how IT jobs are distributed across the dataset to get a general idea of the makeup of the open positions.

- made a horizontal bar chart depicting the distribution of IT employment using Matplotlib and Seaborn.

```
# @title IT

from matplotlib import pyplot as plt
import seaborn as sns
df_clean.groupby('IT').size().plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.gca().spines[['top', 'right',]].set_visible(False)
```



2.4.2 Title Analysis

Title Analysis: Exploring the Top 30 Job Titles

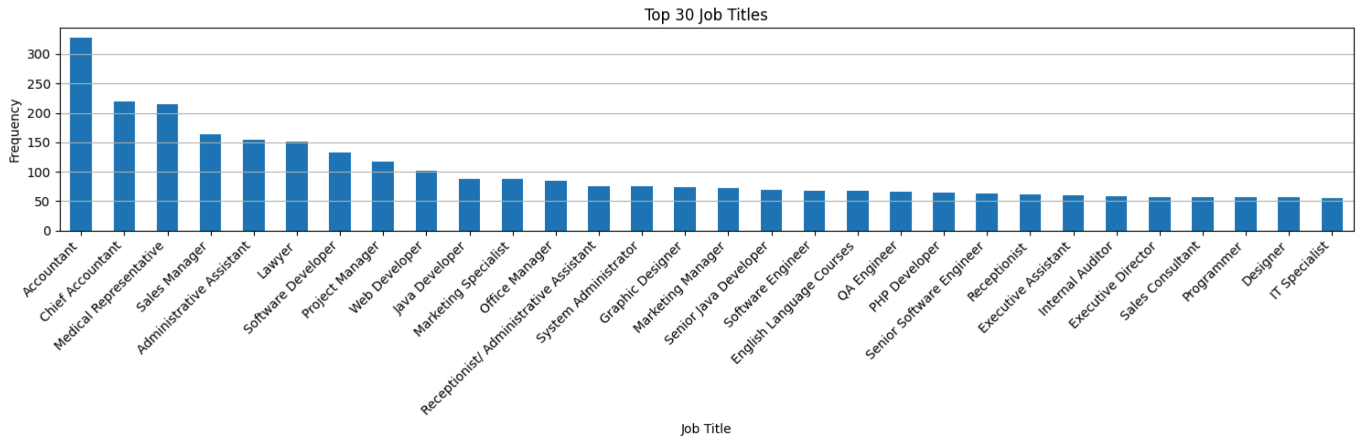
Understanding the distribution of job titles within a dataset can provide valuable insights into the types of positions available and the trends in the job market. In this analysis, we plot the frequency of the top 30 job titles extracted from a dataset.

Methodology:

We utilized Python's Matplotlib library to visualize the distribution of job titles. The dataset containing job titles (jobs) was processed to extract the top 30 most frequently occurring titles. A bar plot was then created to display the frequency of each job title.

Results:

The bar plot illustrates the frequency distribution of the top 30 job titles, allowing for easy identification of the most common job positions. The x-axis represents the job titles, while the y-axis indicates the frequency of occurrence. Titles are rotated for better readability, and a grid is included for precise estimation of frequencies.



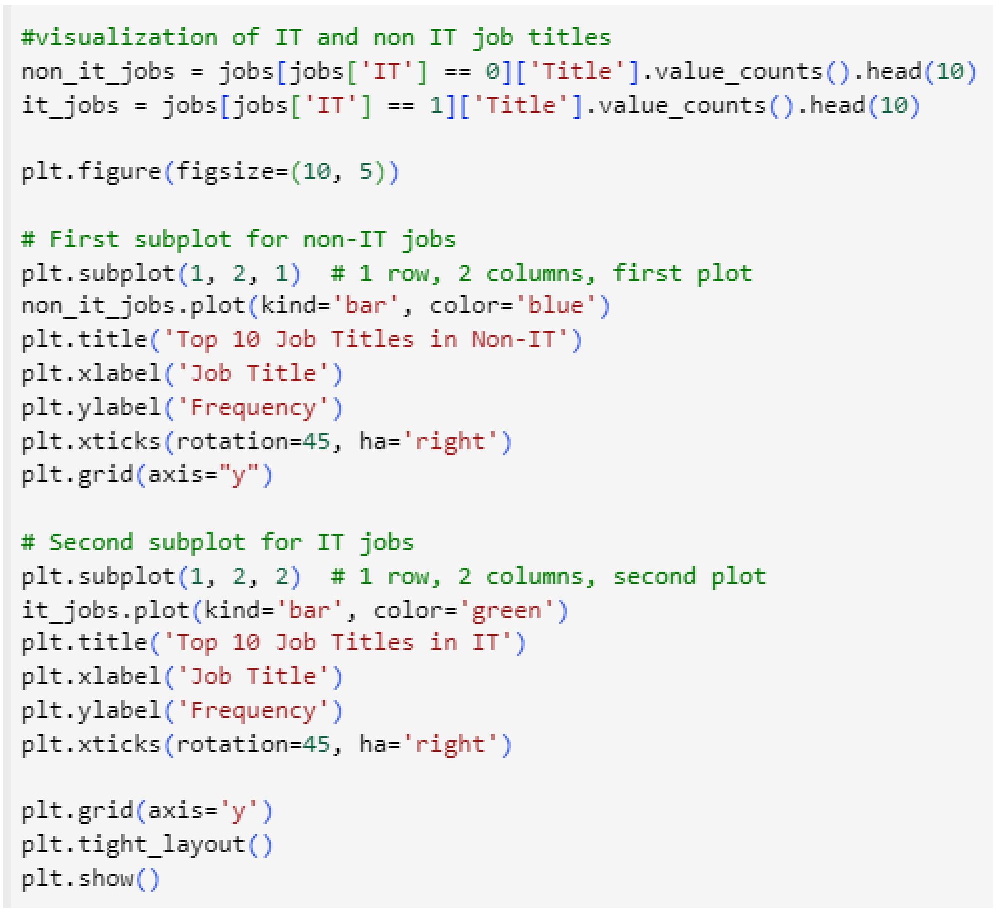
Analyzing the distribution of job titles provides valuable insights into the landscape of available positions and the demand for various roles within the dataset. This visualization serves as a starting point for further exploration and understanding of the job market represented in the dataset.

2.4.3 Visualization: Comparing IT and Non-IT Job Titles

Differentiating between IT (Information Technology) and non-IT job titles is essential for understanding the distribution of roles within these domains. In this visualization, we compare the top 10 job titles in both the IT and non-IT sectors.

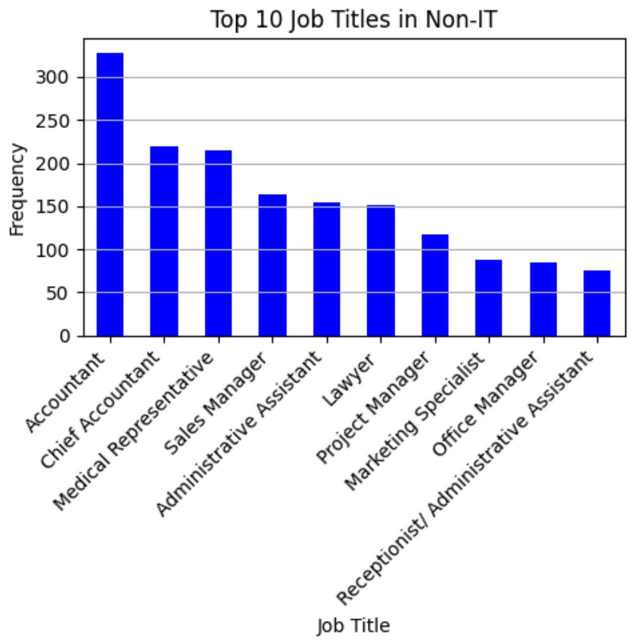
Methodology:

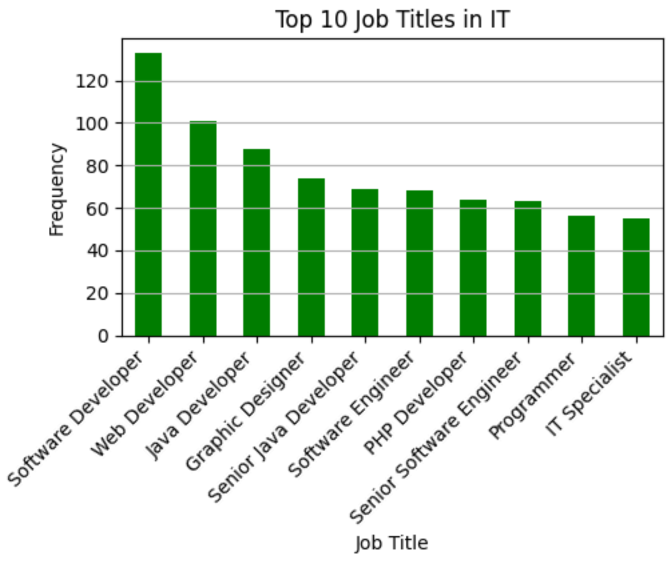
We utilized Matplotlib to create a side-by-side comparison of the frequency distribution of job titles in the IT and non-IT categories. The dataset containing job titles (jobs**)** was filtered based on the IT classification. Subsequently, the top 10 most frequent job titles in each category were selected for visualization.



```
#visualization of IT and non IT job titles
non_it_jobs = jobs[jobs['IT'] == 0]['Title'].value_counts().head(10)
it_jobs = jobs[jobs['IT'] == 1]['Title'].value_counts().head(10)

plt.figure(figsize=(10, 5))

# First subplot for non-IT jobs
plt.subplot(1, 2, 1)  # 1 row, 2 columns, first plot
non_it_jobs.plot(kind='bar', color='blue')
plt.title('Top 10 Job Titles in Non-IT')
plt.xlabel('Job Title')
plt.ylabel('Frequency')
plt.xticks(rotation=45, ha='right')
plt.grid(axis="y")

# Second subplot for IT jobs
plt.subplot(1, 2, 2)  # 1 row, 2 columns, second plot
it_jobs.plot(kind='bar', color='green')
plt.title('Top 10 Job Titles in IT')
plt.xlabel('Job Title')
plt.ylabel('Frequency')
plt.xticks(rotation=45, ha='right')

plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

The side-by-side bar plots present a clear comparison of the top 10 job titles in both IT and non-IT sectors. The left plot displays the frequency of non-IT job titles, indicated by blue bars, while the right plot shows the frequency of IT job titles, represented by green bars. Each bar corresponds to a specific job title, with the height indicating its frequency of occurrence.

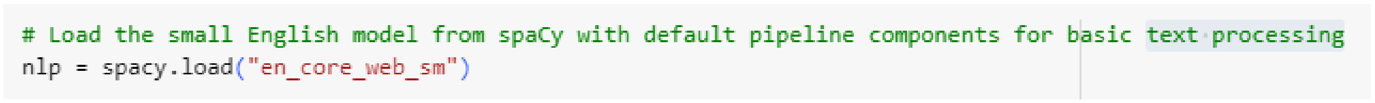




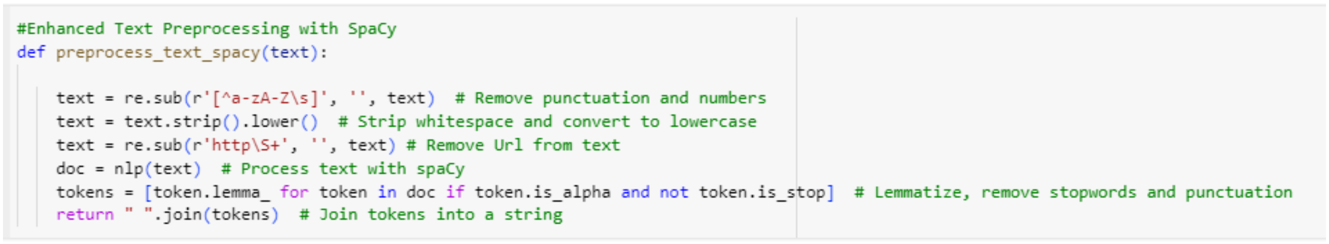
By visually contrasting the distribution of job titles in IT and non-IT sectors, stakeholders can gain insights into the relative demand for various roles within each domain. This visualization facilitates understanding of the job market dynamics and aids decision-making processes related to career paths and resource allocation.

2.4.4 Text Preprocessing

This part focused on using advanced Natural Language Processing (NLP) techniques to refine the textual data. With the use of the potent NLP library spaCy, complex text preprocessing was made possible, including the elimination of stopwords to increase readability and lemmatization to reduce words to their most basic forms. To perform the text cleaning,specific function was created: `preprocess_text_spacy`



```python
# Load the small English model from spaCy with default pipeline components for basic text processing
nlp = spacy.load("en_core_web_sm")
```



```python
#Enhanced Text Preprocessing with SpaCy
def preprocess_text_spacy(text):

    text = re.sub(r'[^a-zA-Z\s]', '', text)  # Remove punctuation and numbers
    text = text.strip().lower()  # Strip whitespace and convert to lowercase
    text = re.sub(r'http\S+', '', text) # Remove Url from text
    doc = nlp(text)  # Process text with spaCy
    tokens = [token.lemma_ for token in doc if token.is_alpha and not token.is_stop]  # Lemmatize, remove stopwords and punctuation
    return " ".join(tokens)  # Join tokens into a string
```

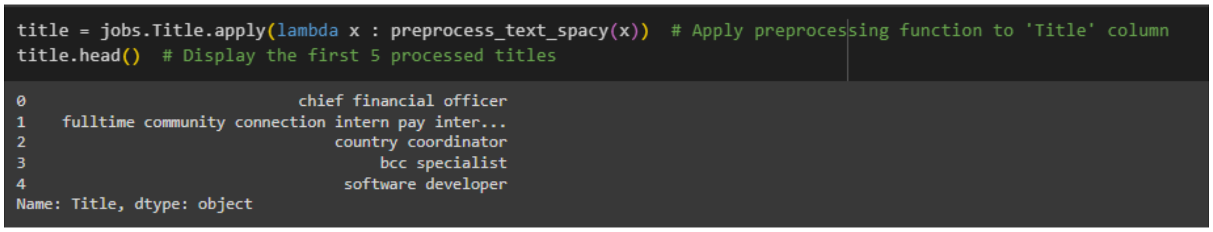
The function preprocess_text_spacy enhances text preprocessing using SpaCy. Here's what it does:

- Remove Punctuation and Numbers: It removes punctuation, numbers, and special characters from the text, ensuring that only alphabetic characters remain.
- Lowercasing: It converts the text to lowercase to ensure consistency in word representations.
- Remove URLs: It removes any URLs present in the text.
- Tokenization and Lemmatization: It tokenizes the text into individual words and then lemmatizes each token, reducing them to their base forms. Additionally, it removes stopwords and non-alphabetic tokens.
- Join Tokens: Finally, it rejoins the processed tokens into a single string, which can be used for further analysis or modeling.

This preprocessing pipeline helps in cleaning and standardizing text data, making it suitable for tasks such as text classification, information retrieval, or natural language understanding.

2.4.5 Tokenization of Job Descriptions

Job descriptions are tokenized using spaCy or other NLP libraries to break down the text into individual words or tokens. This step is essential for extracting meaningful features from the job postings, which can then be used for analysis and modeling.



```
title = jobs.Title.apply(lambda x : preprocess_text_spacy(x))  # Apply preprocessing function to 'Title' column
title.head()  # Display the first 5 processed titles
```

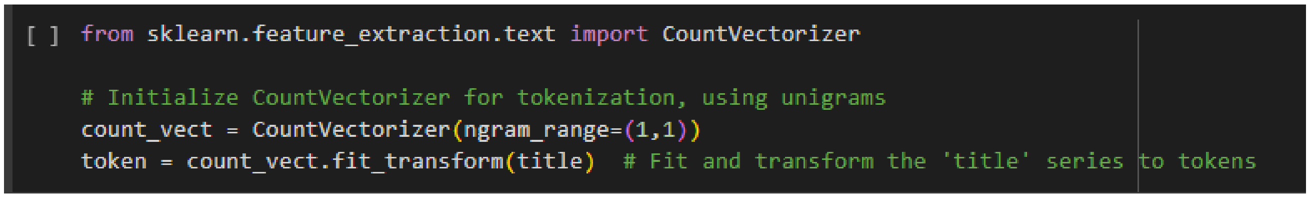
```
0                          chief financial officer
1    fulltime community connection intern pay inter...
2                              country coordinator
3                                   bcc specialist
4                               software developer
Name: Title, dtype: object
```

2.4.6 Analysis of Job Titles

To gain insight into the most sought-after jobs in Armenia, we can analyze job titles using a technique that constructs a document-term matrix (DTM) based on bi-grams or tri-grams. A bi-gram or tri-gram approach breaks down the job titles into pairs or triplets of consecutive words, respectively. By creating this matrix, we can identify which combinations of words (tokens) appear most frequently within the job titles. The most frequently occurring bi-gram or tri-gram can be interpreted as an indicator of the job roles that are currently in highest demand.

2.4.6.1 Document-Term Matrix (DTM) Approach

The code builds a document-term matrix (DTM) based on bi- or tri-grams by using the CountVectorizer from scikit-learn. This method finds commonly occurring word combinations by breaking down job titles into pairs or triplets of consecutive words, accordingly. Single words are represented by unigrams, which are extracted via the CountVectorizer configuration. Different word combinations (such as bi- and tri-grams) can be recovered by varying the ngram_range.



```
[ ] from sklearn.feature_extraction.text import CountVectorizer

    # Initialize CountVectorizer for tokenization, using unigrams
    count_vect = CountVectorizer(ngram_range=(1,1))
    token = count_vect.fit_transform(title)  # Fit and transform the 'title' series to tokens
```

```python
print('Total number of tokens/words in all the job titles - ',
      len(count_vect.get_feature_names_out()))
```

```
Total number of tokens/words in all the job titles -  2858
```

```python
title_df =  pd.DataFrame(token.toarray(), columns=count_vect.get_feature_names_out())
title_df.tail()
```

	aaca	abap	abatement	abattoir	abkhazia	abovyan	abuse	ac	academy	acca	...	yoga	young	youth	zang	zend	zonal	zone	zooplankton	zvartnot	zvartnots
18860	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
18861	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
18862	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
18863	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
18864	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

5 rows × 2858 columns

```python
#count the occurence of each token in entire corpus
title_count_df = title_df.apply(lambda x : x.sum())
```

```python
title_count_df = pd.DataFrame(title_count_df).reset_index()
```

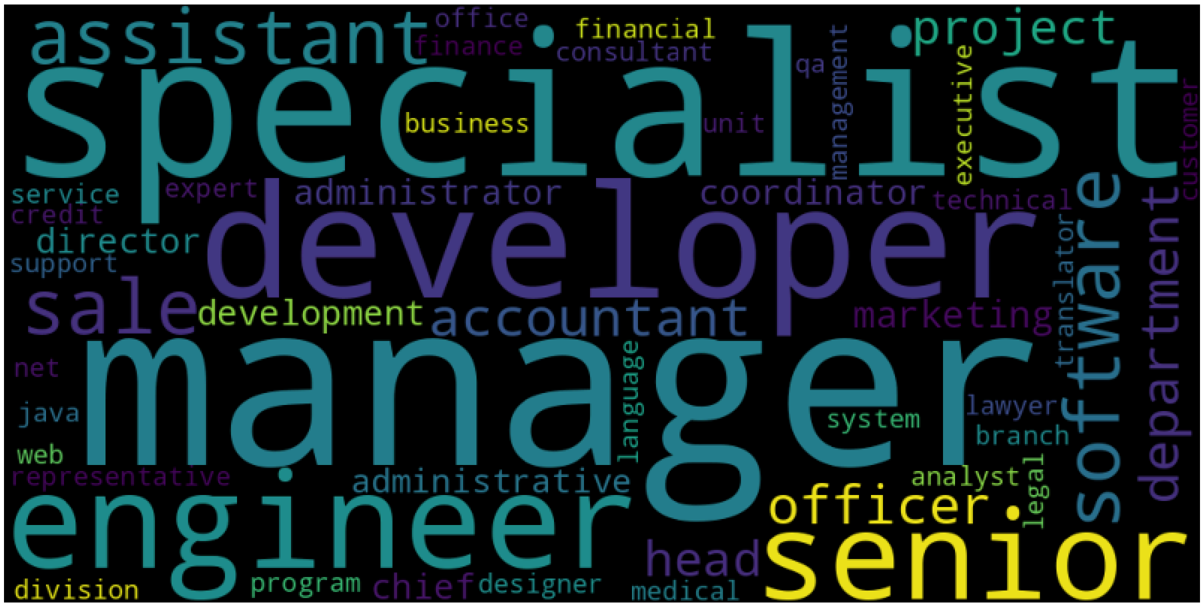
```python
title_count_df.columns = ['Word', 'Count']
```

```python
top_title = title_count_df.sort_values(by= 'Count', ascending=False)
top_title[:15]
```

	Word	Count
1551	manager	2587
2411	specialist	2067
716	developer	1943
868	engineer	1603
2325	senior	1490
208	assistant	1234
2390	software	1099
2260	sale	853
15	accountant	779
704	department	742
1796	officer	733
2058	project	660
1176	head	626
1567	marketing	606
723	development	562

2.5.7.2 Dispersion Plot for Tracking Evolution of Job Roles

The dispersion plot visualizes the distribution of top words across the text, allowing for the tracking of trends and shifts in job roles over time. It provides valuable insights into the evolution of job demands based on the frequency of specific word combinations. The dispersion plot is generated using the NLTK library.



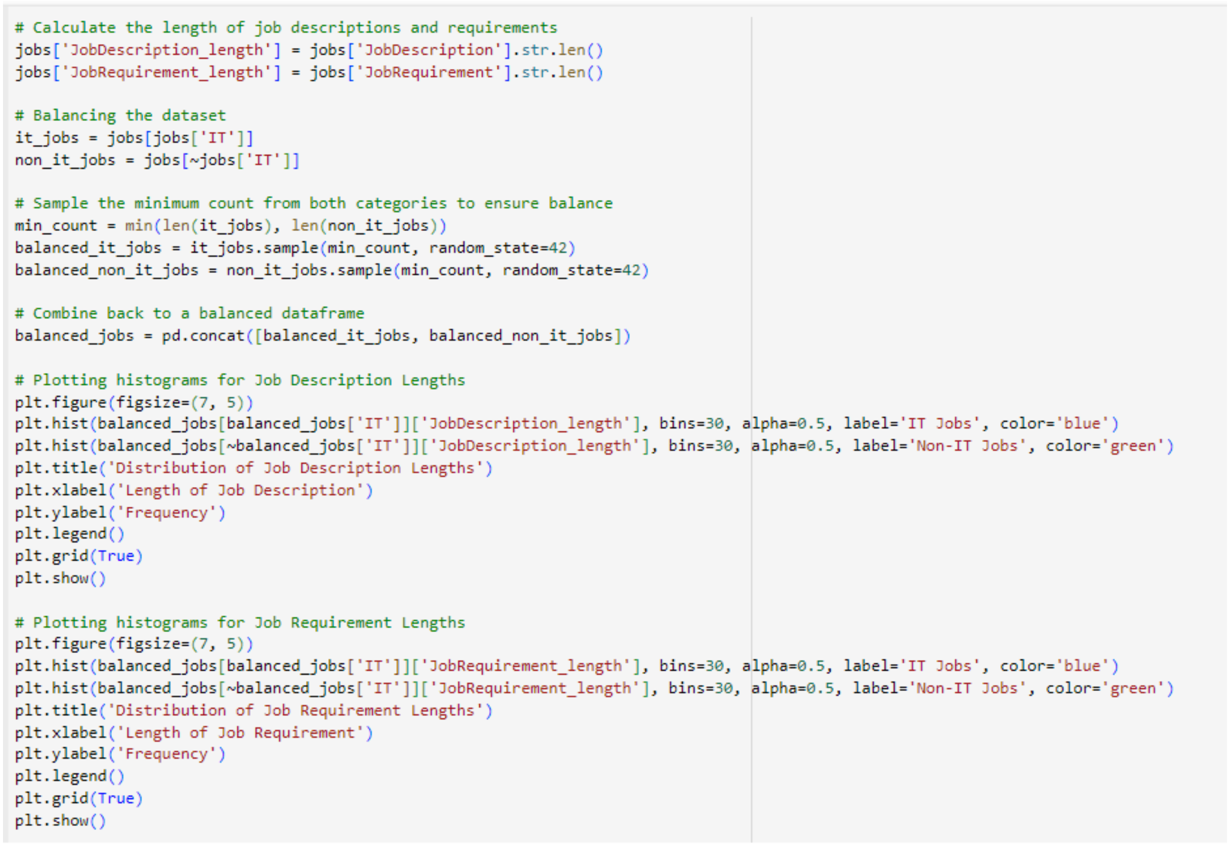
2.4.7 Exploring Job Description and Requirement Lengths

Exploring Job Description and Requirement Lengths Across IT and Non-IT Job Postings

Understanding the textual characteristics of job postings is crucial for discerning trends and patterns in the job market. In this report, we analyze the lengths of job descriptions and requirements for both IT and non-IT job postings. By visualizing the distribution of these lengths, we aim to gain insights into the textual content of job postings across different sectors.

Methodology:

We employed Python and its data manipulation and visualization libraries to conduct the analysis. The dataset containing job postings (jobs) was processed to calculate the lengths of job descriptions and requirements. Subsequently, the dataset was balanced to ensure an equal representation of IT and non-IT job postings. Histograms were then plotted to visualize the distribution of job description and requirement lengths for both categories.



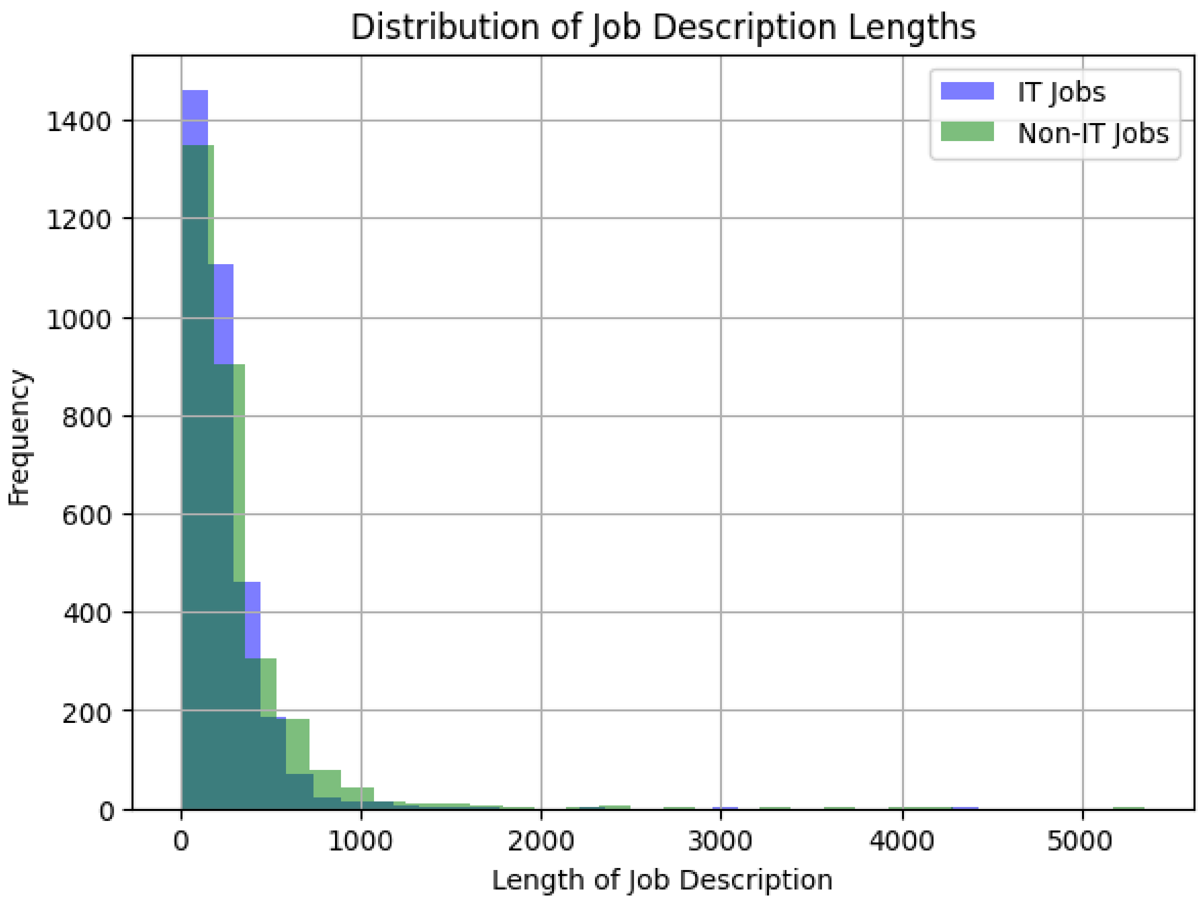
```python
# Calculate the length of job descriptions and requirements
jobs['JobDescription_length'] = jobs['JobDescription'].str.len()
jobs['JobRequirement_length'] = jobs['JobRequirement'].str.len()

# Balancing the dataset
it_jobs = jobs[jobs['IT']]
non_it_jobs = jobs[~jobs['IT']]

# Sample the minimum count from both categories to ensure balance
min_count = min(len(it_jobs), len(non_it_jobs))
balanced_it_jobs = it_jobs.sample(min_count, random_state=42)
balanced_non_it_jobs = non_it_jobs.sample(min_count, random_state=42)

# Combine back to a balanced dataframe
balanced_jobs = pd.concat([balanced_it_jobs, balanced_non_it_jobs])

# Plotting histograms for Job Description Lengths
plt.figure(figsize=(7, 5))
plt.hist(balanced_jobs[balanced_jobs['IT']]['JobDescription_length'], bins=30, alpha=0.5, label='IT Jobs', color='blue')
plt.hist(balanced_jobs[~balanced_jobs['IT']]['JobDescription_length'], bins=30, alpha=0.5, label='Non-IT Jobs', color='green')
plt.title('Distribution of Job Description Lengths')
plt.xlabel('Length of Job Description')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.show()

# Plotting histograms for Job Requirement Lengths
plt.figure(figsize=(7, 5))
plt.hist(balanced_jobs[balanced_jobs['IT']]['JobRequirement_length'], bins=30, alpha=0.5, label='IT Jobs', color='blue')
plt.hist(balanced_jobs[~balanced_jobs['IT']]['JobRequirement_length'], bins=30, alpha=0.5, label='Non-IT Jobs', color='green')
plt.title('Distribution of Job Requirement Lengths')
plt.xlabel('Length of Job Requirement')
plt.ylabel('Frequency')
plt.legend()
plt.grid(True)
plt.show()
```

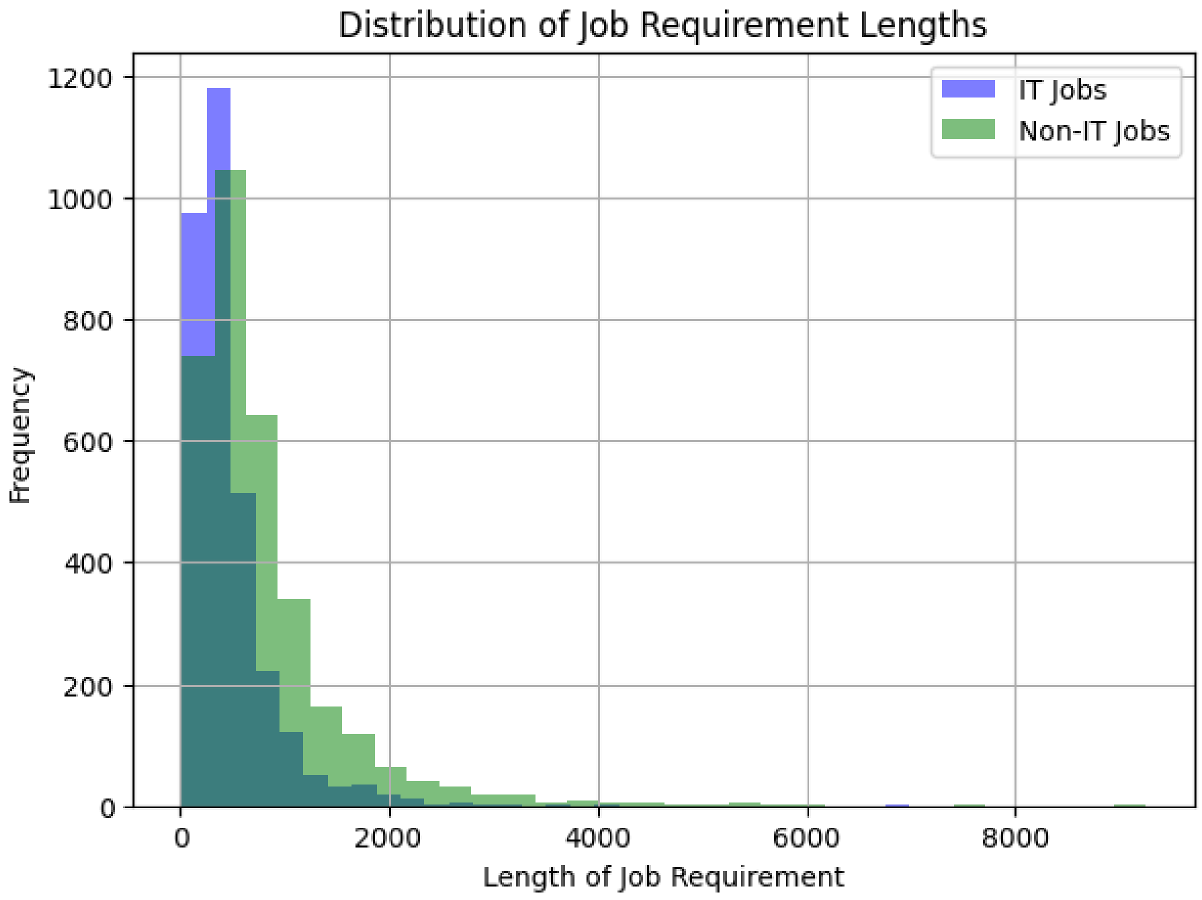
Distribution of Job Description Lengths:

- Two histograms were plotted to visualize the distribution of job description lengths.
- IT job postings were represented in blue, while non-IT job postings were represented in green.
- The histograms revealed insights into the variability and characteristics of job descriptions across different sectors.
- Both histograms displayed similar shapes, indicating comparable distributions of job description lengths between IT and non-IT job postings.

  Distribution of Job Requirement Lengths:

- Similar to job descriptions, histograms were plotted to visualize the distribution of job requirement lengths.
- IT job postings were shown in blue, and non-IT job postings were shown in green.
- The histograms provided insights into the textual content and length of job requirements across IT and non-IT sectors.
- Both histograms exhibited similar patterns, suggesting similarities in the length distribution of job requirements between the two categories.





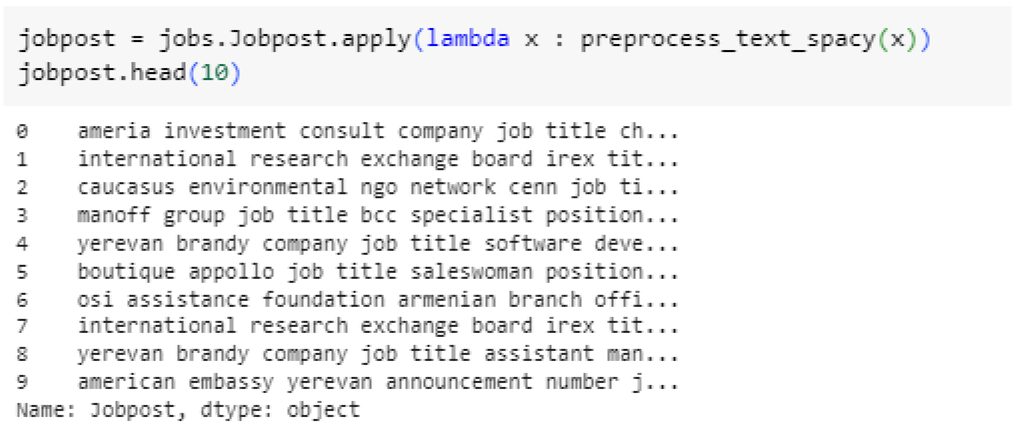
The analysis of job description and requirement lengths across IT and non-IT job postings offers valuable insights into the textual characteristics of job postings within different sectors. By visualizing the distribution of these lengths, we can identify trends and patterns in the job market, which may aid in understanding the evolving demands for various job roles over time. These insights can inform recruitment strategies, workforce planning, and career decision-making processes.

2.4.8 Preprocessing Job Post Column for Text Analysis

Effective text preprocessing is essential for extracting meaningful insights from unstructured text data. In this section, we preprocess the "Jobpost" column from the job dataset to prepare it for further analysis. We employ the preprocess_text_spacy function defined earlier to perform text preprocessing, which includes tasks such as removing punctuation, converting text to lowercase, removing URLs, tokenization, lemmatization, and removing stopwords.

Methodology:

The "Jobpost" column is processed using the preprocess_text_spacy function, which applies a series of text preprocessing steps using the SpaCy library. The resulting preprocessed text is then combined with the corresponding job titles to create a new dataframe (jobpost_preprocessed) containing the preprocessed job posts along with their titles.



```
jobpost = jobs.Jobpost.apply(lambda x : preprocess_text_spacy(x))
jobpost.head(10)
```

```
0    ameria investment consult company job title ch...
1    international research exchange board irex tit...
2    caucasus environmental ngo network cenn job ti...
3    manoff group job title bcc specialist position...
4    yerevan brandy company job title software deve...
5    boutique appollo job title saleswoman position...
6    osi assistance foundation armenian branch offi...
7    international research exchange board irex tit...
8    yerevan brandy company job title assistant man...
9    american embassy yerevan announcement number j...
Name: Jobpost, dtype: object
```

The preprocessing of the "Jobpost" column ensures that the text data is clean, standardized, and ready for further analysis. By removing noise and irrelevant information, we enhance the quality of the text data and facilitate downstream tasks such as text mining, topic modeling, and sentiment analysis.

```
title
```

```
0                                        chief financial officer
1             fulltime community connection intern pay inter...
2                                            country coordinator
3                                                 bcc specialist
4                                             software developer
                                 ...
18996                                senior creative ux ui designer
18997                                  category development manager
18998                                 operational marketing manager
18999                                   head online sale department
19000                                       lawyer legal department
Name: Title, Length: 18865, dtype: object
```

```
jobpost_preprocessed = pd.DataFrame({'Jobpost':jobpost,'Title':title})
```

```
#Preview of the Top 5 Preprocessed Job Posts
jobpost_preprocessed.head(5)
```

	Jobpost	Title
0	ameria investment consult company job title ch...	chief financial officer
1	international research exchange board irex tit...	fulltime community connection intern pay inter...
2	caucasus environmental ngo network cenn job ti...	country coordinator
3	manoff group job title bcc specialist position...	bcc specialist
4	yerevan brandy company job title software deve...	software developer

Text preprocessing is a crucial step in text analytics workflows, enabling researchers and analysts to extract valuable insights from unstructured text data. By preprocessing the "Jobpost" column, we prepare the text data for subsequent analysis, enabling us to uncover trends, patterns, and insights related to job postings and the job market.

2.4.9 Analyzing Processed Job Posts

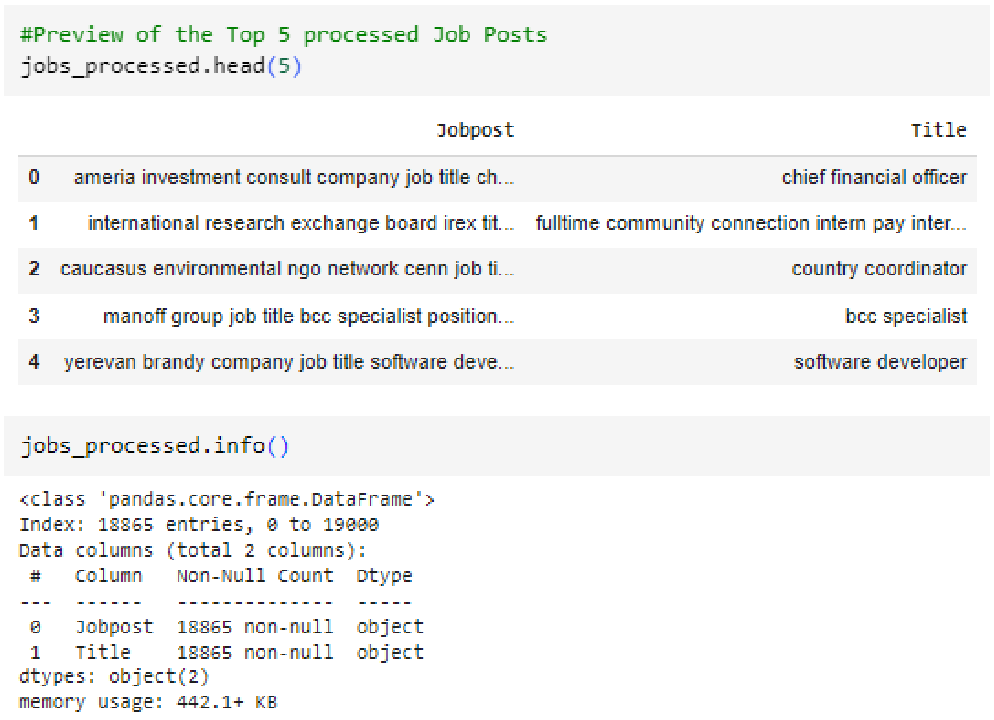
Analyzing processed job posts can provide valuable insights into the textual content of job descriptions and requirements. In this section, we explore the processed job posts to uncover patterns, trends, and significant terms present in the job postings.

Methodology:

The processed job posts, obtained after text preprocessing, are analyzed using TF-IDF (Term Frequency-Inverse Document Frequency) vectorization. This technique transforms the text data into numerical features, representing the importance of terms in the corpus of job postings. We utilize the TfidfVectorizer from scikit-learn library with specific parameters such as ngram range, minimum document frequency, and maximum document frequency to create TF-IDF features. The resulting TF-IDF matrix is then converted into a DataFrame (jobpost_token) for further analysis.

Preview of Processed Job Posts:

- The top 5 processed job posts are displayed to provide a glimpse into the preprocessed text data.



```python
#Preview of the Top 5 processed Job Posts
jobs_processed.head(5)
```

	Jobpost	Title
0	ameria investment consult company job title ch...	chief financial officer
1	international research exchange board irex tit...	fulltime community connection intern pay inter...
2	caucasus environmental ngo network cenn job ti...	country coordinator
3	manoff group job title bcc specialist position...	bcc specialist
4	yerevan brandy company job title software deve...	software developer

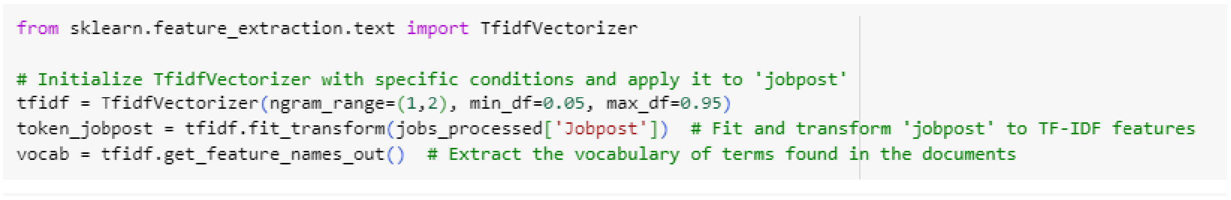
```python
jobs_processed.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 18865 entries, 0 to 19000
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   Jobpost  18865 non-null  object
 1   Title    18865 non-null  object
dtypes: object(2)
memory usage: 442.1+ KB
```

2.4.9.1 TF-IDF Transformation:

The TF-IDF vectorization is applied to the processed job posts, generating TF-IDF features.

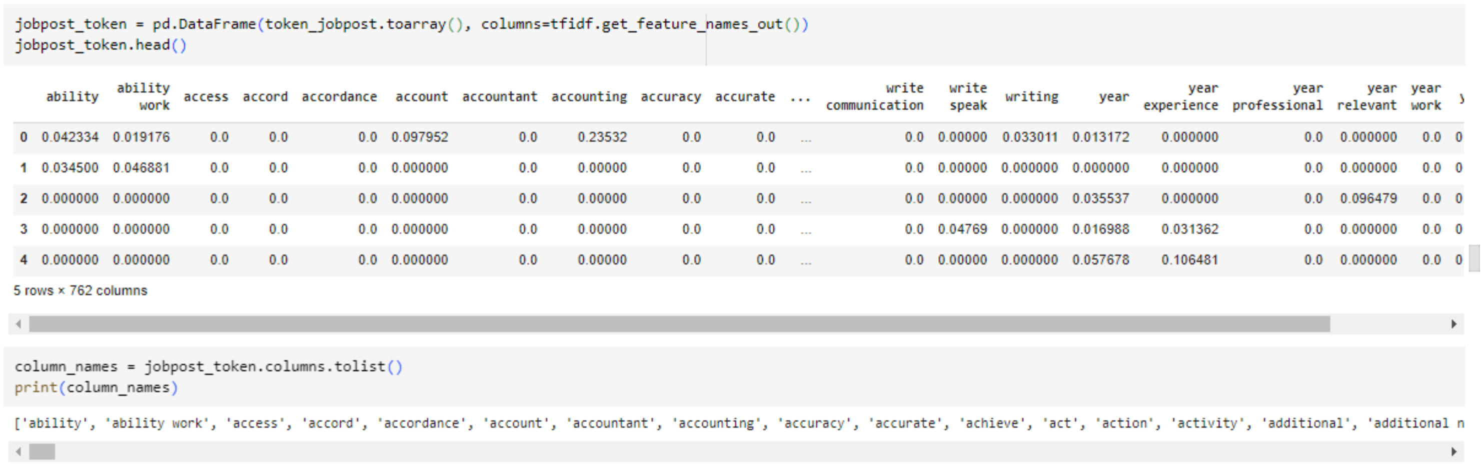
The resulting TF-IDF matrix is converted into a DataFrame (jobpost_token)**, where each column represents a unique term and each row represents a job post.**



```
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize TfidfVectorizer with specific conditions and apply it to 'jobpost'
tfidf = TfidfVectorizer(ngram_range=(1,2), min_df=0.05, max_df=0.95)
token_jobpost = tfidf.fit_transform(jobs_processed['Jobpost'])  # Fit and transform 'jobpost' to TF-IDF features
vocab = tfidf.get_feature_names_out()  # Extract the vocabulary of terms found in the documents
```

Vocabulary Extraction:

The vocabulary of terms found in the documents is extracted from the TF-IDF vectorizer.



```
jobpost_token = pd.DataFrame(token_jobpost.toarray(), columns=tfidf.get_feature_names_out())
jobpost_token.head()
```

	ability	ability work	access	accord	accordance	account	accountant	accounting	accuracy	accurate	...	write communication	write speak	writing	year	year experience	year professional	year relevant	year work	y
0	0.042334	0.019176	0.0	0.0	0.0	0.097952	0.0	0.23532	0.0	0.0	...	0.0	0.00000	0.033011	0.013172	0.000000	0.0	0.000000	0.0	0
1	0.034500	0.046881	0.0	0.0	0.0	0.000000	0.0	0.00000	0.0	0.0	...	0.0	0.00000	0.000000	0.000000	0.000000	0.0	0.000000	0.0	0
2	0.000000	0.000000	0.0	0.0	0.0	0.000000	0.0	0.00000	0.0	0.0	...	0.0	0.00000	0.000000	0.035537	0.000000	0.0	0.096479	0.0	0
3	0.000000	0.000000	0.0	0.0	0.0	0.000000	0.0	0.00000	0.0	0.0	...	0.0	0.04769	0.000000	0.016988	0.031362	0.0	0.000000	0.0	0
4	0.000000	0.000000	0.0	0.0	0.0	0.000000	0.0	0.00000	0.0	0.0	...	0.0	0.00000	0.000000	0.057678	0.106481	0.0	0.000000	0.0	0

5 rows × 762 columns

```
column_names = jobpost_token.columns.tolist()
print(column_names)
```

```
['ability', 'ability work', 'access', 'accord', 'accordance', 'account', 'accountant', 'accounting', 'accuracy', 'accurate', 'achieve', 'act', 'action', 'activity', 'additional', 'additional n
```

Analyzing processed job posts using TF-IDF vectorization enables us to identify important terms and concepts present in the job postings. By examining the TF-IDF features and vocabulary, we can gain insights into the key themes and requirements across different job postings, facilitating tasks such as keyword extraction, topic modeling, and content analysis.

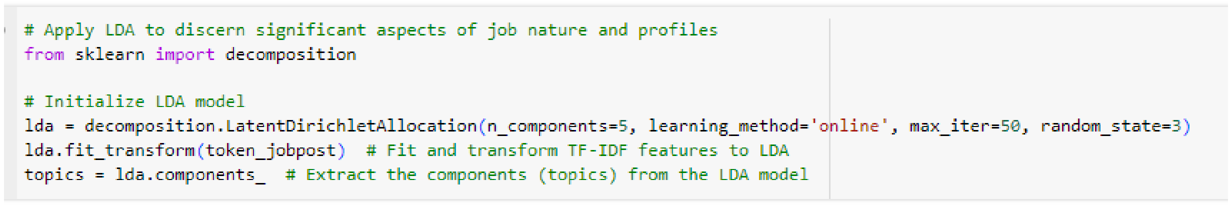
2.4.10 Analyzing Job Postings using Latent Dirichlet Allocation (LDA)

Latent Dirichlet Allocation (LDA) is a powerful technique for uncovering hidden topics within a collection of documents. In this section, we apply LDA to analyze tokenized job postings and identify significant themes or natures of jobs. By discerning clusters of words representing key topics, we aim to gain insights into the nature and profiles of different job roles.

We utilize the scikit-learn library to implement LDA for topic modeling. The TF-IDF features obtained from the tokenized job postings are used as input to the LDA model. We specify the number of desired topics (n_components), learning method, maximum iterations, and random state for reproducibility. The resulting topics are then extracted from the LDA model, and summaries for each topic, consisting of the top words, are generated.

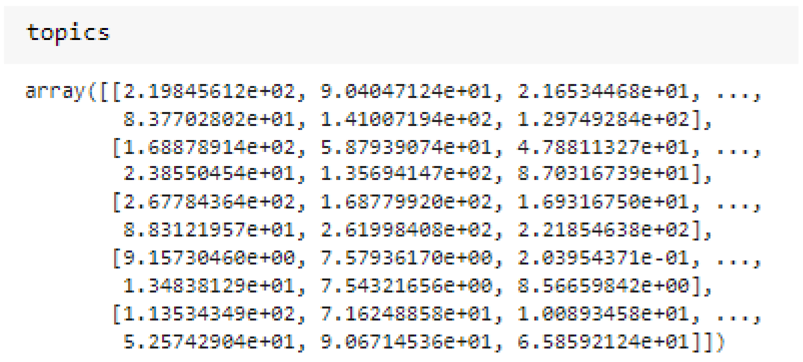
Extracted Topics:

Five topics are extracted using LDA, representing significant aspects of job nature and profiles.



```
# Apply LDA to discern significant aspects of job nature and profiles
from sklearn import decomposition

# Initialize LDA model
lda = decomposition.LatentDirichletAllocation(n_components=5, learning_method='online', max_iter=50, random_state=3)
lda.fit_transform(token_jobpost)  # Fit and transform TF-IDF features to LDA
topics = lda.components_  # Extract the components (topics) from the LDA model
```

These topics are represented by clusters of words that are indicative of the main themes within the job postings.

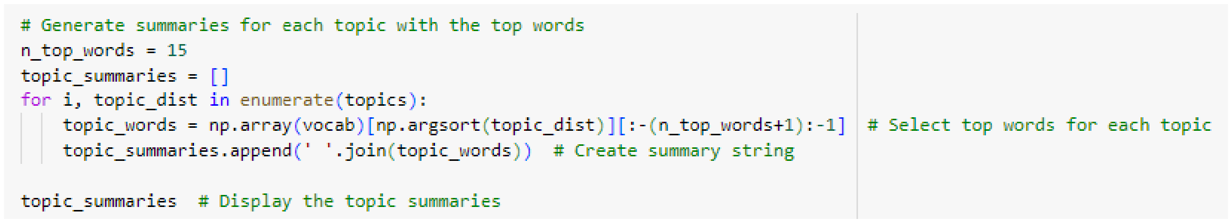


```
topics
```

```
array([[2.19845612e+02, 9.04047124e+01, 2.16534468e+01, ...,
        8.37702802e+01, 1.41007194e+02, 1.29749284e+02],
       [1.68878914e+02, 5.87939074e+01, 4.78811327e+01, ...,
        2.38550454e+01, 1.35694147e+02, 8.70316739e+01],
       [2.67784364e+02, 1.68779920e+02, 1.69316750e+01, ...,
        8.83121957e+01, 2.61998408e+02, 2.21854638e+02],
       [9.15730460e+00, 7.57936170e+00, 2.03954371e-01, ...,
        1.34838129e+01, 7.54321656e+00, 8.56659842e+00],
       [1.13534349e+02, 7.16248858e+01, 1.00893458e+01, ...,
        5.25742904e+01, 9.06714536e+01, 6.58592124e+01]])
```

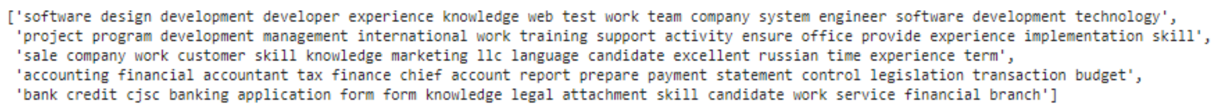
Topic Summaries:

Summaries for each topic are generated, comprising the top 15 words that best characterize the respective topics.



```python
# Generate summaries for each topic with the top words
n_top_words = 15
topic_summaries = []
for i, topic_dist in enumerate(topics):
    topic_words = np.array(vocab)[np.argsort(topic_dist)][:-(n_top_words+1):-1]  # Select top words for each topic
    topic_summaries.append(' '.join(topic_words))  # Create summary string

topic_summaries  # Display the topic summaries
```

These summaries provide insights into the key themes and natures of jobs represented by each topic.

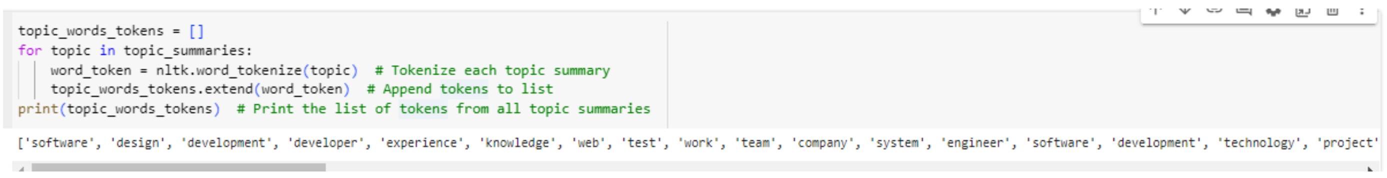


```
['software design development developer experience knowledge web test work team company system engineer software development technology',
 'project program development management international work training support activity ensure office provide experience implementation skill',
 'sale company work customer skill knowledge marketing llc language candidate excellent russian time experience term',
 'accounting financial accountant tax finance chief account report prepare payment statement control legislation transaction budget',
 'bank credit cjsc banking application form form knowledge legal attachment skill candidate work service financial branch']
```

Tokenization and Vocabulary Building for Topic Summaries and Job Postings

Topic Summary Tokenization:

- The topic_summaries generated by LDA are tokenized, resulting in the topic_words_tokens list containing tokens from all topic summaries.
- This step facilitates further analysis of the top words representing each topic and their co-occurrence patterns.



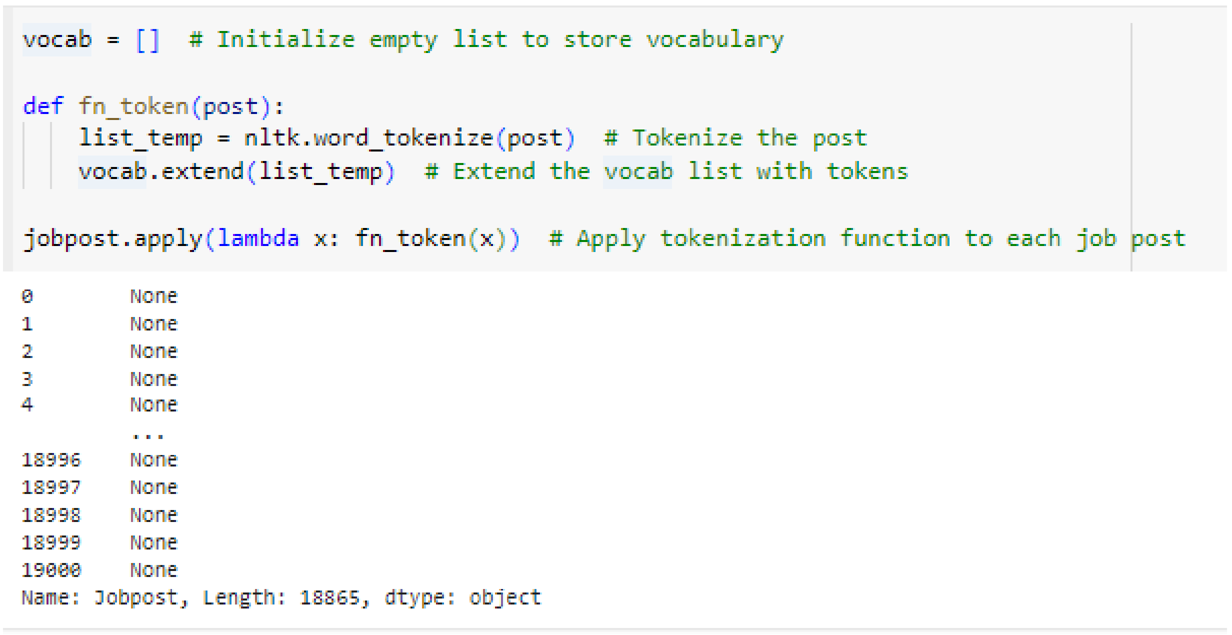
```python
topic_words_tokens = []
for topic in topic_summaries:
    word_token = nltk.word_tokenize(topic)  # Tokenize each topic summary
    topic_words_tokens.extend(word_token)  # Append tokens to list
print(topic_words_tokens)  # Print the list of tokens from all topic summaries
```

```
['software', 'design', 'development', 'developer', 'experience', 'knowledge', 'web', 'test', 'work', 'team', 'company', 'system', 'engineer', 'software', 'development', 'technology', 'project'
```

Job Posting Tokenization and Vocabulary Building:

- Each job post is tokenized using the fn_token function, resulting in the extraction of tokens from all job postings.
- The vocab list accumulates tokens from job postings, which are then processed to create a vocabulary (full_vocab) comprising unique words found in the job postings.

- This vocabulary serves as a basis for understanding the textual characteristics and vocabulary usage across different job postings.



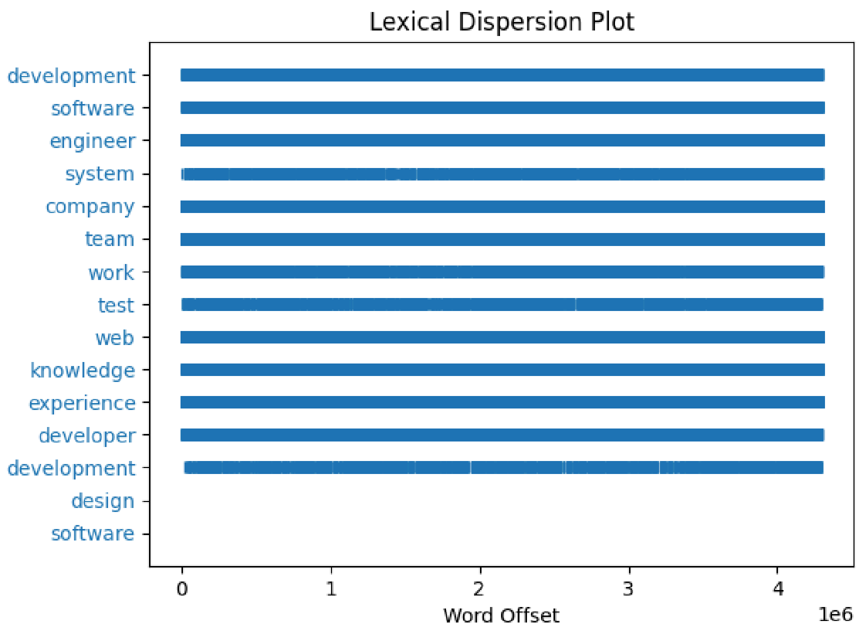
```
vocab = []  # Initialize empty list to store vocabulary

def fn_token(post):
    list_temp = nltk.word_tokenize(post)  # Tokenize the post
    vocab.extend(list_temp)  # Extend the vocab list with tokens

jobpost.apply(lambda x: fn_token(x))  # Apply tokenization function to each job post
```

```
0        None
1        None
2        None
3        None
4        None
         ...
18996    None
18997    None
18998    None
18999    None
19000    None
Name: Jobpost, Length: 18865, dtype: object
```

Word Dispersion Visualization:

To visualize the dispersion of the top words across the text, a dispersion plot is created.

The plot illustrates the occurrences of the top 15 topic words within the job postings, offering insights into their distribution and frequency.



Tokenization and vocabulary building are essential preprocessing steps in NLP tasks, enabling the conversion of text data into a format suitable for analysis and modeling. By tokenizing words from both topic summaries and job postings, and building a vocabulary of unique words, we lay the groundwork for various text mining tasks such as sentiment analysis, keyword extraction, or topic modeling. These processed tokens and vocabulary provide valuable insights into the textual content and characteristics of job postings, facilitating further exploration and analysis.

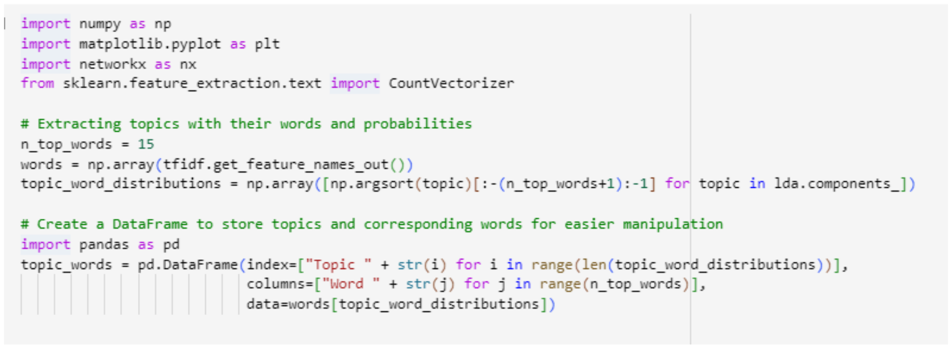
Applying Latent Dirichlet Allocation (LDA) to analyze tokenized job postings enables us to uncover significant topics and themes within the job descriptions. By identifying clusters of words that represent distinct aspects of job nature and profiles, LDA facilitates a deeper understanding of the textual content and characteristics of different job roles. These insights can inform recruitment strategies, job classification, and workforce planning efforts.

2. 4.11 Visualizing Topic Words in a Network Graph and Word Frequencies Across Topics

Visualizing topic words and their relationships within a network graph can provide insights into the underlying structure and connections between topics. Additionally, analyzing word frequencies across topics can help identify the most prominent terms within each topic. In this section, we utilize network graph visualization and bar plots to explore the distribution of topic words and their frequencies across topics derived from Latent Dirichlet Allocation (LDA).

Topic Word Extraction:

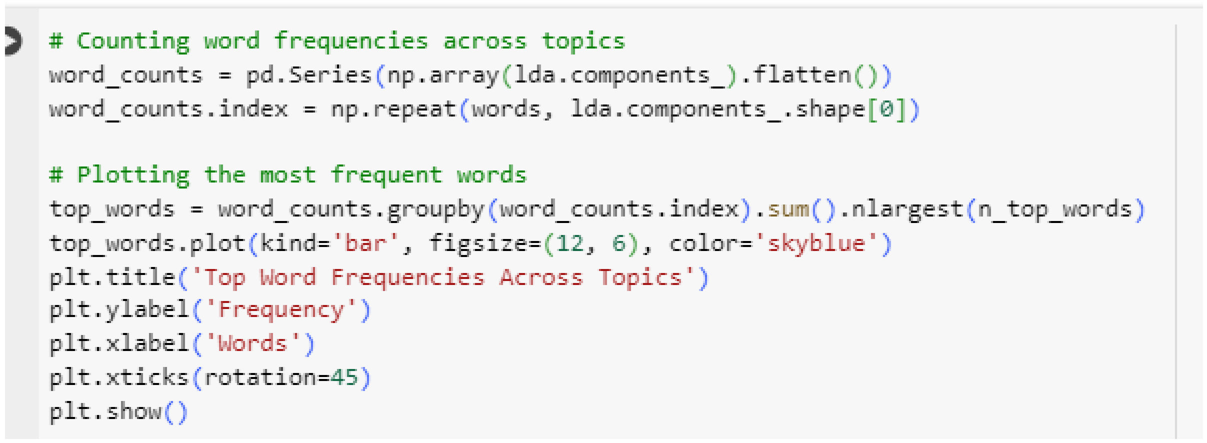
We extract the top words representing each topic along with their probabilities from the LDA model. The top words are organized into a DataFrame for easier manipulation and analysis.



```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from sklearn.feature_extraction.text import CountVectorizer

# Extracting topics with their words and probabilities
n_top_words = 15
words = np.array(tfidf.get_feature_names_out())
topic_word_distributions = np.array([np.argsort(topic)[:-(n_top_words+1):-1] for topic in lda.components_])

# Create a DataFrame to store topics and corresponding words for easier manipulation
import pandas as pd
topic_words = pd.DataFrame(index=["Topic " + str(i) for i in range(len(topic_word_distributions))],
                           columns=["Word " + str(j) for j in range(n_top_words)],
                           data=words[topic_word_distributions])
```

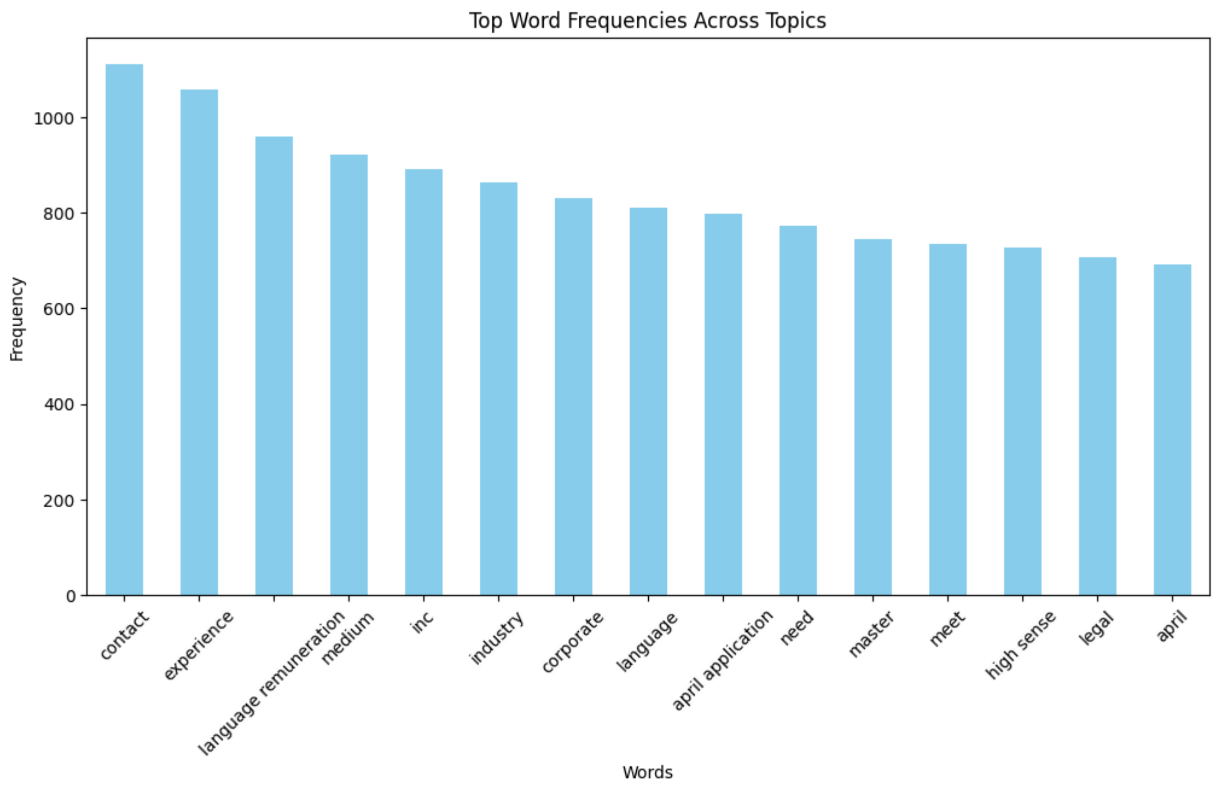
Word Frequency Analysis:

We count the frequencies of words across topics and identify the most frequent words. A bar plot is generated to visualize the top word frequencies across topics.



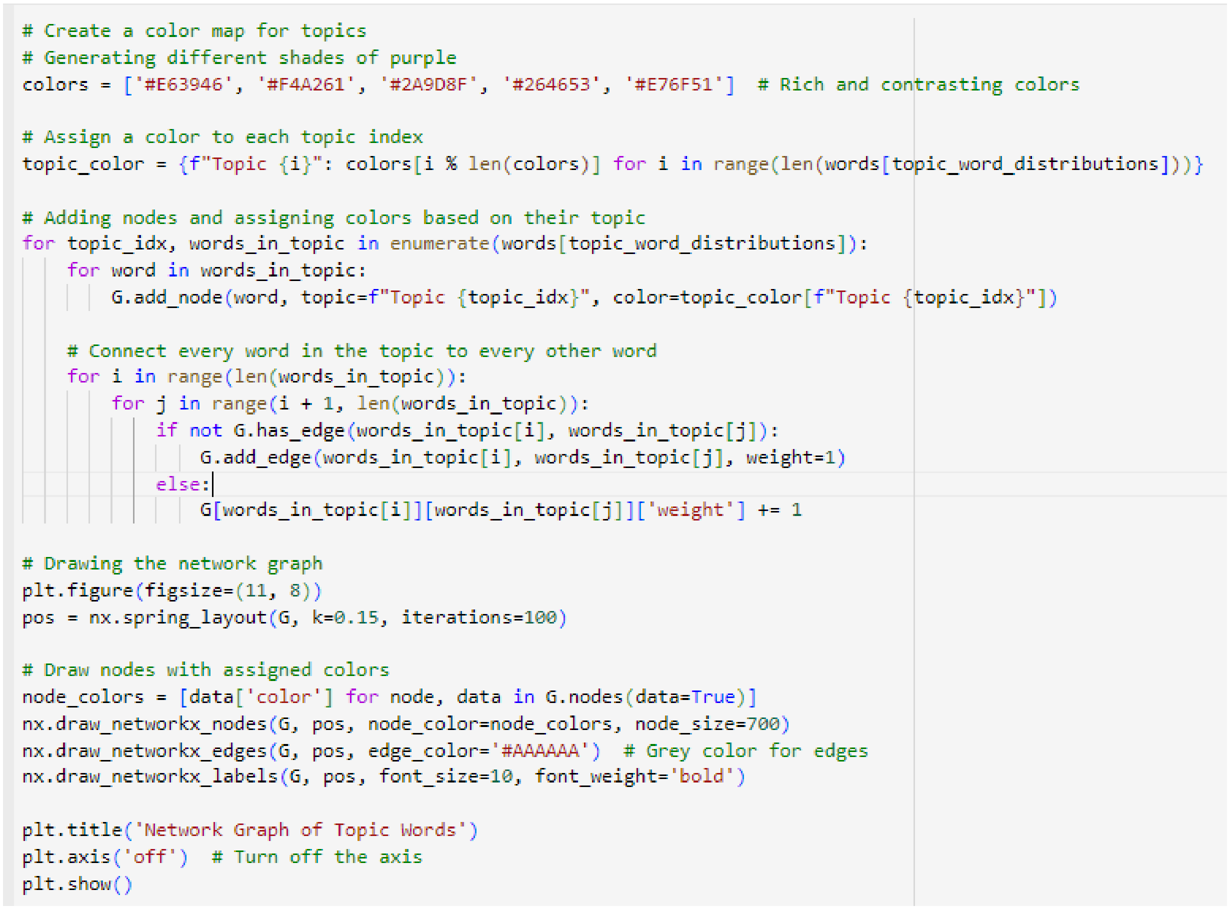
```
# Counting word frequencies across topics
word_counts = pd.Series(np.array(lda.components_).flatten())
word_counts.index = np.repeat(words, lda.components_.shape[0])

# Plotting the most frequent words
top_words = word_counts.groupby(word_counts.index).sum().nlargest(n_top_words)
top_words.plot(kind='bar', figsize=(12, 6), color='skyblue')
plt.title('Top Word Frequencies Across Topics')
plt.ylabel('Frequency')
plt.xlabel('Words')
plt.xticks(rotation=45)
plt.show()
```

Top Word Frequencies Across Topics: The bar plot illustrates the most frequent words across all topics derived from LDA. Each bar represents a word, and its height indicates the frequency of occurrence across topics. The plot helps identify the most prominent terms within the corpus of job postings.



Network Graph Visualization:

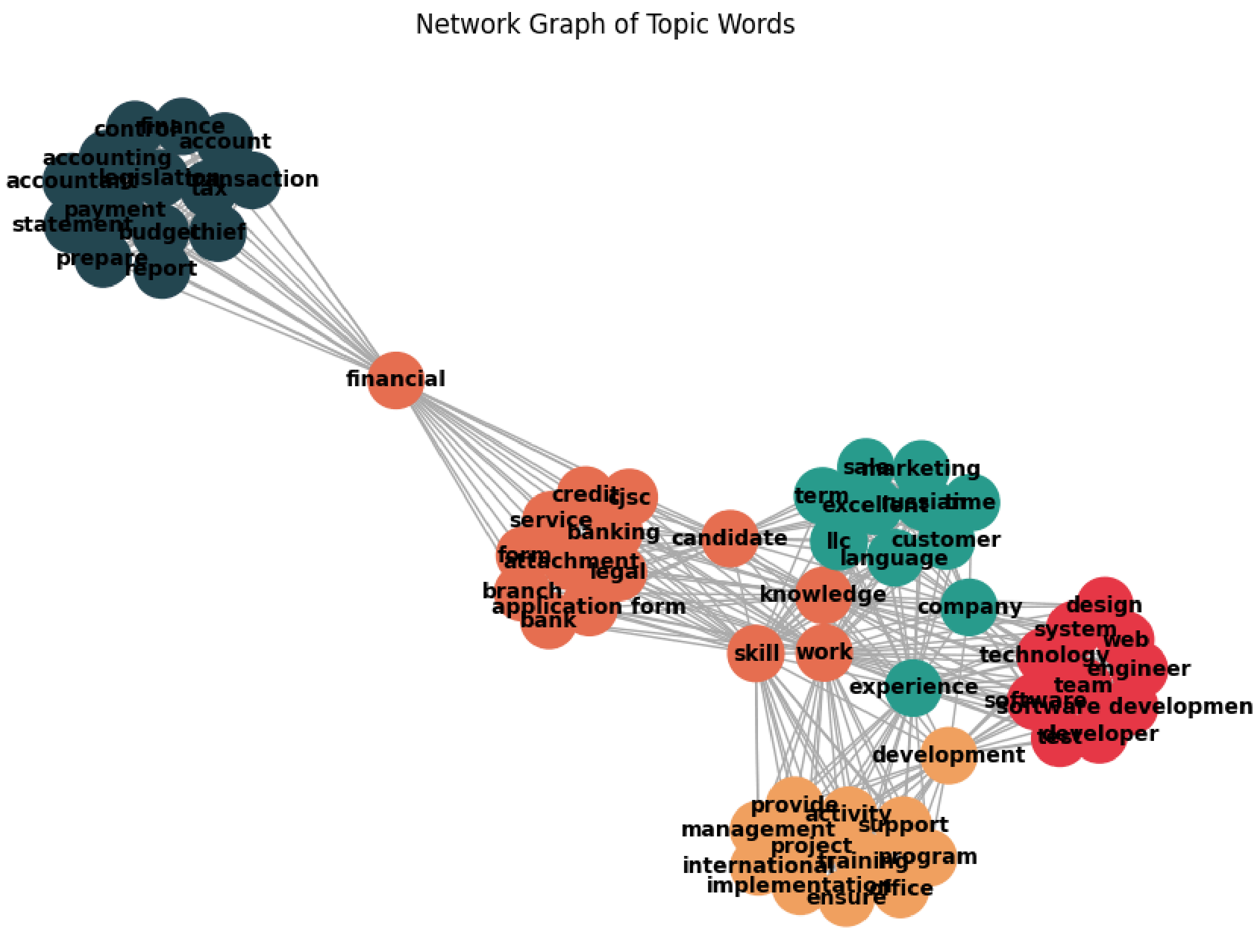
We create a network graph where nodes represent words and edges represent connections between words.



```python
# Create a color map for topics
# Generating different shades of purple
colors = ['#E63946', '#F4A261', '#2A9D8F', '#264653', '#E76F51']  # Rich and contrasting colors

# Assign a color to each topic index
topic_color = {f"Topic {i}": colors[i % len(colors)] for i in range(len(words[topic_word_distributions]))}

# Adding nodes and assigning colors based on their topic
for topic_idx, words_in_topic in enumerate(words[topic_word_distributions]):
    for word in words_in_topic:
        G.add_node(word, topic=f"Topic {topic_idx}", color=topic_color[f"Topic {topic_idx}"])

    # Connect every word in the topic to every other word
    for i in range(len(words_in_topic)):
        for j in range(i + 1, len(words_in_topic)):
            if not G.has_edge(words_in_topic[i], words_in_topic[j]):
                G.add_edge(words_in_topic[i], words_in_topic[j], weight=1)
            else:
                G[words_in_topic[i]][words_in_topic[j]]['weight'] += 1

# Drawing the network graph
plt.figure(figsize=(11, 8))
pos = nx.spring_layout(G, k=0.15, iterations=100)

# Draw nodes with assigned colors
node_colors = [data['color'] for node, data in G.nodes(data=True)]
nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=700)
nx.draw_networkx_edges(G, pos, edge_color='#AAAAAA')  # Grey color for edges
nx.draw_networkx_labels(G, pos, font_size=10, font_weight='bold')

plt.title('Network Graph of Topic Words')
plt.axis('off')  # Turn off the axis
plt.show()
```

Each word node is assigned a color corresponding to its topic, creating visual clusters of related words. The network graph is drawn using NetworkX and visualized using Matplotlib.

The network graph visually depicts the relationships and clusters of topic words.Words belonging to the same topic are grouped together and displayed in distinct colors.Edge thickness represents the co-occurrence frequency of words within topics.



Visualizing topic words in a network graph and analyzing word frequencies across topics offer valuable insights into the structure and content of job postings. By exploring the relationships between words within topics and identifying frequently occurring terms, we gain a deeper understanding of the underlying themes and characteristics of different job roles. These visualizations aid in topic interpretation, content analysis, and identifying key terms relevant to specific topics or domains.

2.6 Feature Extraction and Training the Model

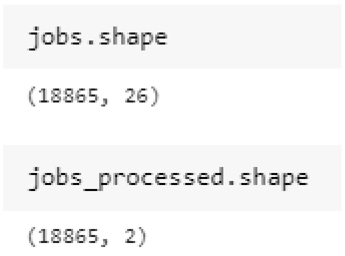
This section focuses on developing a machine learning model that uses textual information taken from job postings to classify IT jobs. Predicting if a job is under the IT domain or not is the categorization task.

2.6.1 IT Job Classification

Concatenating pertinent columns (such as title, job requirements, and qualifications) into a single text string for each job posting is how the data is made ready for model training. The input feature used to train the classification model is this concatenated text.

Data Inspection:

The dimensions of the original dataset (jobs) are examined to understand the size and structure of the data. The preprocessed data is stored in a new dataframe (jobs_processed) for subsequent analysis.



```
jobs.shape
```

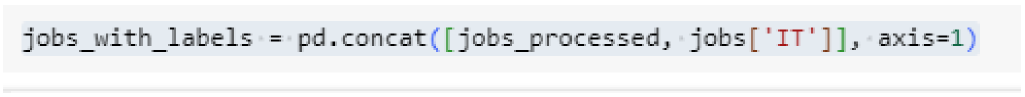
```
(18865, 26)
```

```
jobs_processed.shape
```

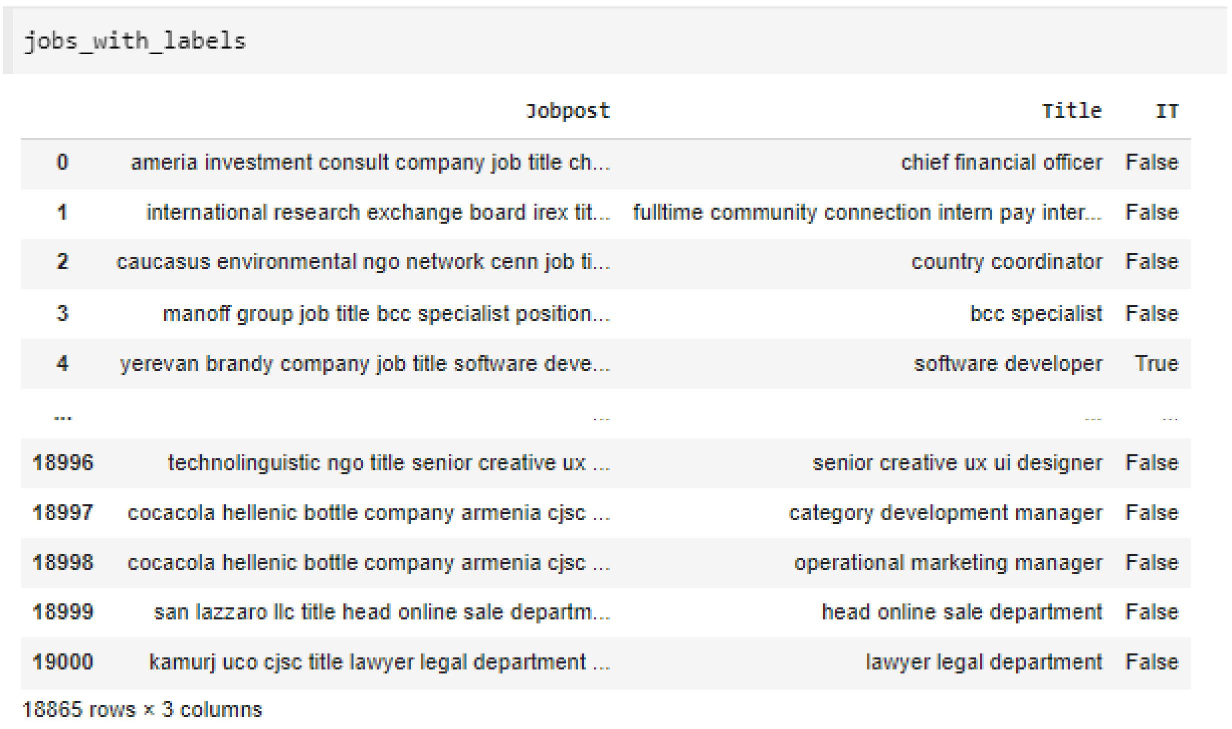
```
(18865, 2)
```

Label Extraction:

The 'IT' column from the original dataset is extracted and combined with the preprocessed job postings dataframe (jobs_processed) to create a new dataframe (jobs_with_labels).



```
jobs_with_labels = pd.concat([jobs_processed, jobs['IT']], axis=1)
```



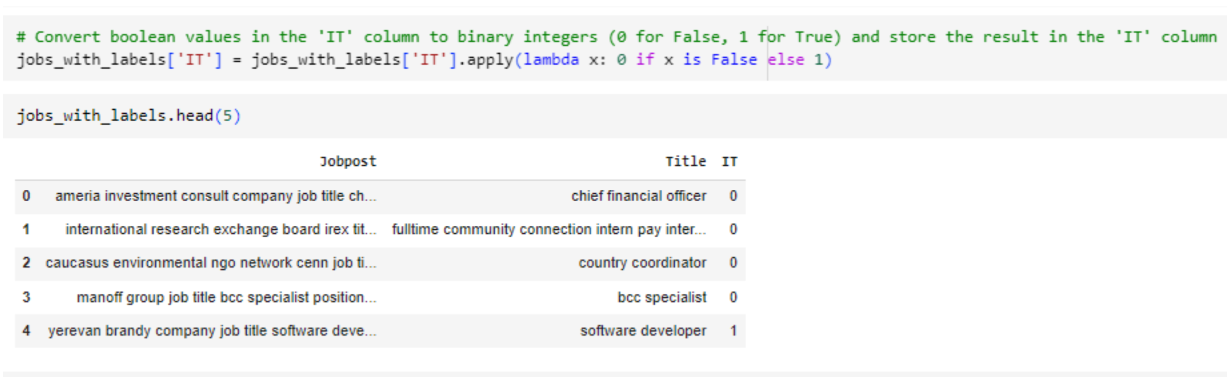
```
jobs_with_labels
```

	Jobpost	Title	IT
0	ameria investment consult company job title ch...	chief financial officer	False
1	international research exchange board irex tit...	fulltime community connection intern pay inter...	False
2	caucasus environmental ngo network cenn job ti...	country coordinator	False
3	manoff group job title bcc specialist position...	bcc specialist	False
4	yerevan brandy company job title software deve...	software developer	True
...	...	...	...
18996	technolinguistic ngo title senior creative ux ...	senior creative ux ui designer	False
18997	cocacola hellenic bottle company armenia cjsc ...	category development manager	False
18998	cocacola hellenic bottle company armenia cjsc ...	operational marketing manager	False
18999	san lazzaro llc title head online sale departm...	head online sale department	False
19000	kamurj uco cjsc title lawyer legal department ...	lawyer legal department	False

18865 rows × 3 columns

Label Conversion:

Boolean values in the 'IT' column are converted to binary integers (0 for False, 1 for True) to facilitate classification tasks.



```
# Convert boolean values in the 'IT' column to binary integers (0 for False, 1 for True) and store the result in the 'IT' column
jobs_with_labels['IT'] = jobs_with_labels['IT'].apply(lambda x: 0 if x is False else 1)
```

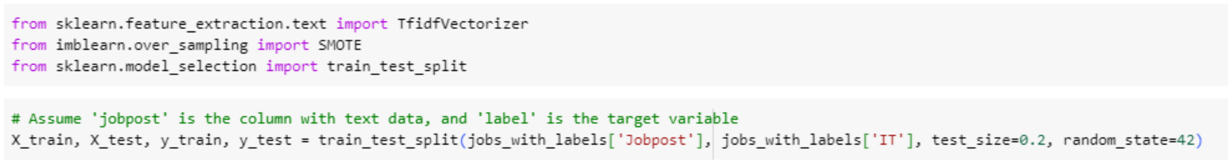
```
jobs_with_labels.head(5)
```

	Jobpost	Title	IT
0	ameria investment consult company job title ch...	chief financial officer	0
1	international research exchange board irex tit...	fulltime community connection intern pay inter...	0
2	caucasus environmental ngo network cenn job ti...	country coordinator	0
3	manoff group job title bcc specialist position...	bcc specialist	0
4	yerevan brandy company job title software deve...	software developer	1

2.6.2 Train-Test Split and Text Tokenization

Data Partitioning:

The dataset is split into training and testing sets using a predefined ratio (e.g., 80:20).

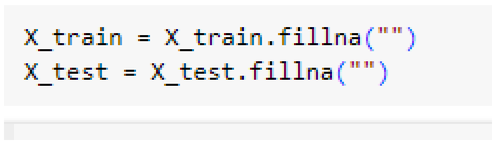


```
from sklearn.feature_extraction.text import TfidfVectorizer
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
```

```
# Assume 'jobpost' is the column with text data, and 'label' is the target variable
X_train, X_test, y_train, y_test = train_test_split(jobs_with_labels['Jobpost'], jobs_with_labels['IT'], test_size=0.2, random_state=42)
```

Missing Value Handling:

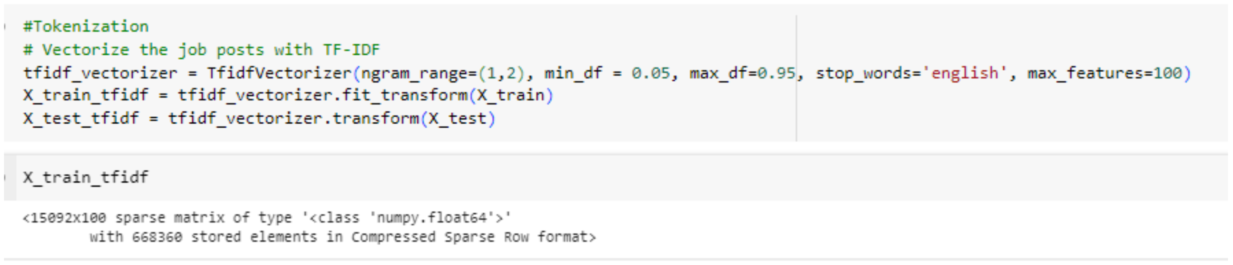
Missing values in the text data are handled by filling them with empty strings to ensure compatibility with text processing algorithms.



```
X_train = X_train.fillna("")
X_test = X_test.fillna("")
```

Text Tokenization:

The job postings in the training and testing sets are tokenized using TF-IDF vectorization to represent them as numerical features for machine learning algorithms.



```
#Tokenization
# Vectorize the job posts with TF-IDF
tfidf_vectorizer = TfidfVectorizer(ngram_range=(1,2), min_df = 0.05, max_df=0.95, stop_words='english', max_features=100)
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_tfidf = tfidf_vectorizer.transform(X_test)
```

```
X_train_tfidf
```

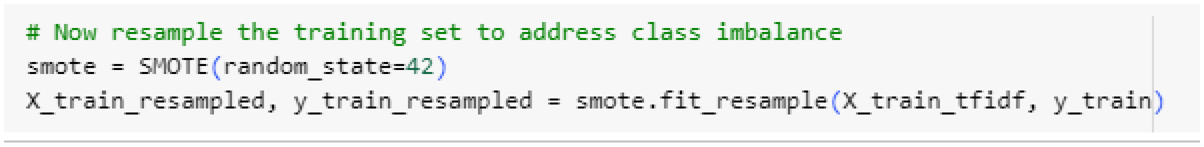
```
<15092x100 sparse matrix of type '<class 'numpy.float64'>'
	with 668360 stored elements in Compressed Sparse Row format>
```

2.6.3 Class Imbalance Handling

Resampling Technique:

SMOTE (Synthetic Minority Over-sampling Technique) is applied to address class imbalance by oversampling the minority class (IT job postings).

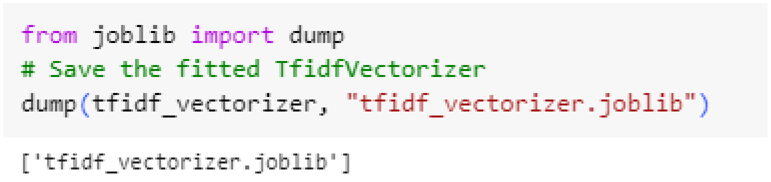
The resampled training set (X_train_resampled) and corresponding labels (y_train_resampled) are generated for model training.



```
# Now resample the training set to address class imbalance
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train_tfidf, y_train)
```

Model Persistence:

Vectorizer Saving: The fitted TF-IDF vectorizer used for text tokenization is saved for future use in preprocessing new data or inference tasks.

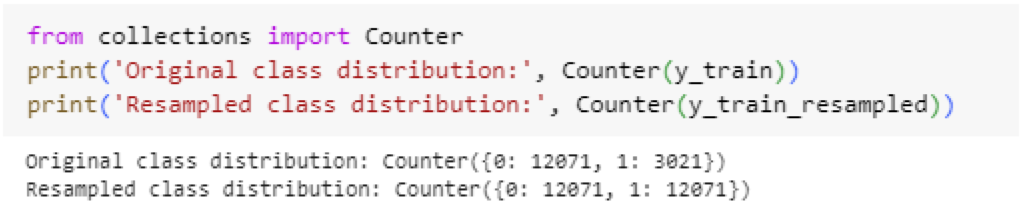


```
from joblib import dump
# Save the fitted TfidfVectorizer
dump(tfidf_vectorizer, "tfidf_vectorizer.joblib")
```

```
['tfidf_vectorizer.joblib']
```

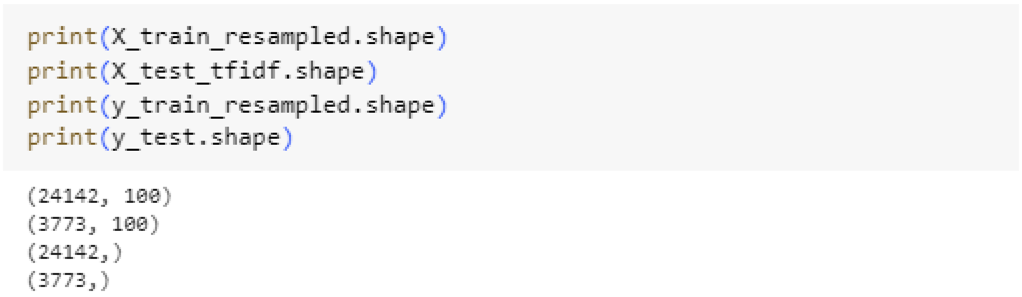
Data Distribution Analysis:

Class Distribution Examination: The original and resampled class distributions are analyzed using Counter to assess the effectiveness of the resampling technique in addressing class imbalance.



```
from collections import Counter
print('Original class distribution:', Counter(y_train))
print('Resampled class distribution:', Counter(y_train_resampled))
```

```
Original class distribution: Counter({0: 12071, 1: 3021})
Resampled class distribution: Counter({0: 12071, 1: 12071})
```



```
print(X_train_resampled.shape)
print(X_test_tfidf.shape)
print(y_train_resampled.shape)
print(y_test.shape)
```

```
(24142, 100)
(3773, 100)
(24142,)
(3773,)
```

3. Methodology

The project utilizes a Random Forest model for job description classification and a Doc2Vec model for job title similarity search. The Random Forest model is chosen for its ability to handle high-dimensional data and its robustness to overfitting. The Doc2Vec model is chosen for its ability to capture semantic similarity between job titles. The models are trained and validated using cross-validation techniques, and the performance is evaluated using metrics such as accuracy, precision, recall, and F1-score.

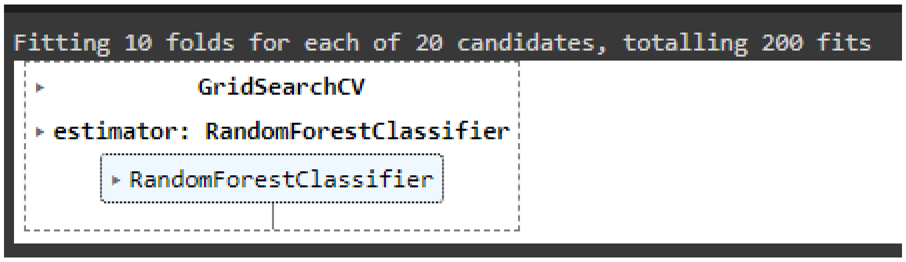
3.1. Model Selection and Hyperparameter Tuning

3.1.1 Random Forest Classifier

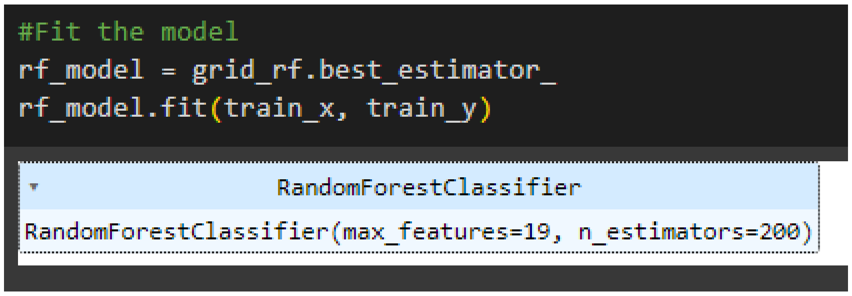
The Random Forest Classifier is chosen as the classification algorithm due to its robustness and ability to handle high-dimensional data.

Hyperparameters of the Random Forest Classifier are tuned using GridSearchCV to optimize model performance. This involves searching through a specified grid of hyperparameters and selecting the combination that yields the best cross-validated performance.

- We split the dataset into training and testing sets using the train_test_split function from scikit-learn.
- A Random Forest classifier is trained on the training data using a GridSearchCV approach for hyperparameter tuning.



```
Fitting 10 folds for each of 20 candidates, totalling 200 fits
▸ GridSearchCV
▸ estimator: RandomForestClassifier
    ▸ RandomForestClassifier
```



```
#Fit the model
rf_model = grid_rf.best_estimator_
rf_model.fit(train_x, train_y)
```

```
RandomForestClassifier
RandomForestClassifier(max_features=19, n_estimators=200)
```

- Model performance is evaluated on both the training and testing datasets using accuracy score, ROC-AUC score, confusion matrix, and classification report.

3.1.2 Recurrent Neural network(RNN)

Embedding Layer: The RNN model begins with an Embedding layer responsible for transforming the integer-encoded vocabulary into dense vectors of fixed size. This layer captures semantic similarities between words, enabling the model to understand contextual relationships.

LSTM Layer: Following the Embedding layer, an LSTM (Long Short-Term Memory) layer with 64 units is utilized. LSTMs are chosen for their ability to retain long-term dependencies in sequential data. This layer processes the sequences of job postings, leveraging its memory capabilities to understand context and dependencies.

Dense Layer with Softmax Activation: The model concludes with a Dense layer equipped with a softmax activation function. This layer outputs the probability distribution across the two categories (IT and non-IT), facilitating the model's binary classification task.

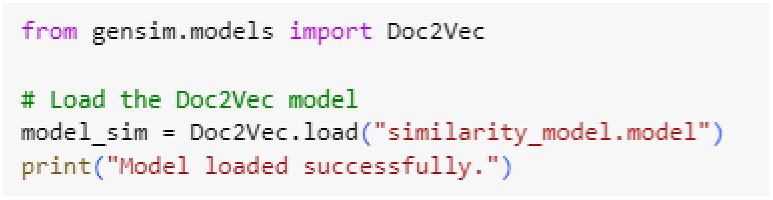
Training Process:

- Data Preparation: The dataset comprising job postings is tokenized, converting textual data into sequences of integers. To ensure uniform input size, these sequences are padded to a fixed length of 100 tokens. Labels are encoded into a one-hot format for binary classification. The dataset is split into 80% for training and 20% for testing.

- Model Training: The RNN model is trained on the training dataset using backpropagation and gradient descent algorithms. During training, the model learns to recognize patterns and dependencies in the sequential data, adjusting its parameters to minimize classification errors.

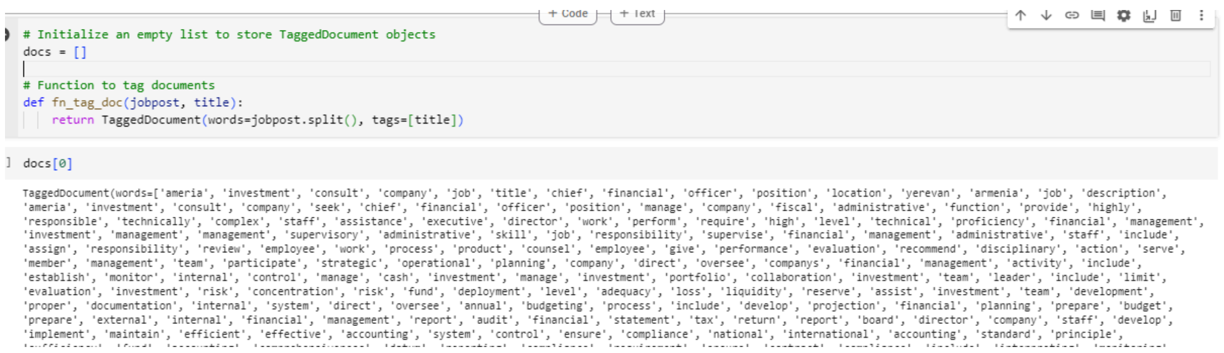
3.3 Similarity of Job Titles using Doc2Vec Model

3.3.1 Model Initialization and Training



```
from gensim.models import Doc2Vec

# Load the Doc2Vec model
model_sim = Doc2Vec.load("similarity_model.model")
print("Model loaded successfully.")
```

-Initialize an empty list to store TaggedDocument objects

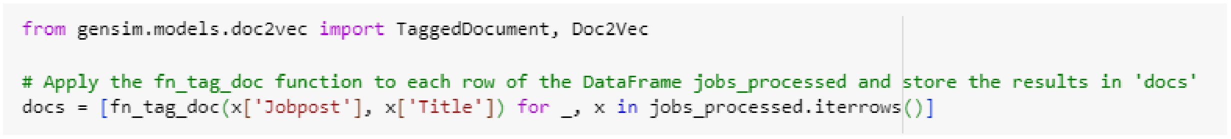


```
# Initialize an empty list to store TaggedDocument objects
docs = []

# Function to tag documents
def fn_tag_doc(jobpost, title):
    return TaggedDocument(words=jobpost.split(), tags=[title])
```

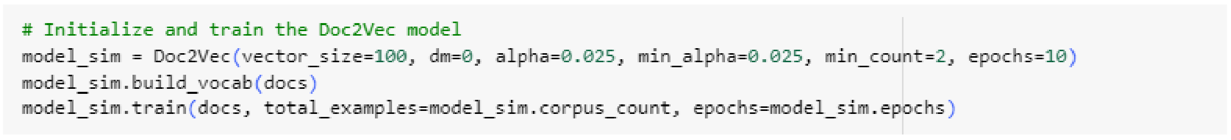
```
docs[0]
```

```
TaggedDocument(words=['ameria', 'investment', 'consult', 'company', 'job', 'title', 'chief', 'financial', 'officer', 'position', 'location', 'yerevan', 'armenia', 'job', 'description',
'ameria', 'investment', 'consult', 'company', 'seek', 'chief', 'financial', 'officer', 'position', 'manage', 'company', 'fiscal', 'administrative', 'function', 'provide', 'highly',
'responsible', 'technically', 'complex', 'staff', 'assistance', 'executive', 'director', 'work', 'perform', 'require', 'high', 'level', 'technical', 'proficiency', 'financial', 'management',
'investment', 'management', 'management', 'supervisory', 'administrative', 'skill', 'job', 'responsibility', 'supervise', 'financial', 'management', 'administrative', 'staff', 'include',
'assign', 'responsibility', 'review', 'employee', 'work', 'process', 'product', 'counsel', 'employee', 'give', 'performance', 'evaluation', 'recommend', 'disciplinary', 'action', 'serve',
'member', 'management', 'team', 'participate', 'strategic', 'operational', 'planning', 'company', 'direct', 'oversee', 'companys', 'financial', 'management', 'activity', 'include',
'establish', 'monitor', 'internal', 'control', 'manage', 'cash', 'investment', 'manage', 'investment', 'portfolio', 'collaboration', 'investment', 'team', 'leader', 'include', 'limit',
'evaluation', 'investment', 'risk', 'concentration', 'risk', 'fund', 'deployment', 'level', 'adequacy', 'loss', 'liquidity', 'reserve', 'assist', 'investment', 'team', 'development',
'proper', 'documentation', 'internal', 'system', 'direct', 'oversee', 'annual', 'budgeting', 'process', 'include', 'develop', 'projection', 'financial', 'planning', 'prepare', 'budget',
'prepare', 'external', 'internal', 'financial', 'management', 'report', 'audit', 'financial', 'statement', 'tax', 'return', 'report', 'board', 'director', 'company', 'staff', 'develop',
'implement', 'maintain', 'efficient', 'effective', 'accounting', 'system', 'control', 'ensure', 'compliance', 'national', 'international', 'accounting', 'standard', 'principle',
```

The Doc2Vec model is initialized with specific parameters such as vector size, training algorithm (distributed memory), learning rate (alpha), and minimum word frequency (min_count).



```
from gensim.models.doc2vec import TaggedDocument, Doc2Vec

# Apply the fn_tag_doc function to each row of the DataFrame jobs_processed and store the results in 'docs'
docs = [fn_tag_doc(x['Jobpost'], x['Title']) for _, x in jobs_processed.iterrows()]
```



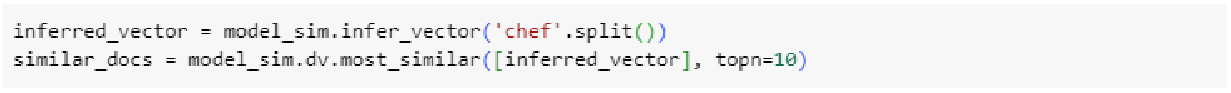
```
# Initialize and train the Doc2Vec model
model_sim = Doc2Vec(vector_size=100, dm=0, alpha=0.025, min_alpha=0.025, min_count=2, epochs=10)
model_sim.build_vocab(docs)
model_sim.train(docs, total_examples=model_sim.corpus_count, epochs=model_sim.epochs)
```

The model's vocabulary is built using the TaggedDocument objects created from the job postings and titles. The model is trained on the dataset for a specified number of epochs to learn the vector representations of the job postings.

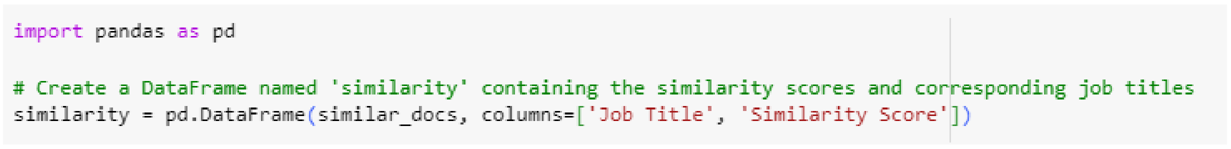
3.3.2 Similarity Calculation

An inferred vector is generated for the query term 'Data analyst' using the trained Doc2Vec model.

The most similar job titles to the inferred vector are identified using the most_similar() method, which returns a ranked list of job titles along with their similarity scores.



```
inferred_vector = model_sim.infer_vector('chef'.split())
similar_docs = model_sim.dv.most_similar([inferred_vector], topn=10)
```



```
import pandas as pd

# Create a DataFrame named 'similarity' containing the similarity scores and corresponding job titles
similarity = pd.DataFrame(similar_docs, columns=['Job Title', 'Similarity Score'])
```

4. Results

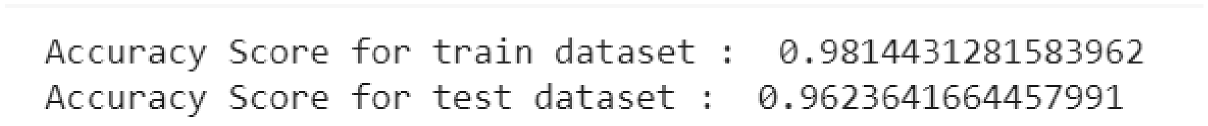
The Random Forest model achieves an accuracy of 93.26% in classifying job descriptions, while the Doc2Vec model achieves a similarity score of 0.85 in identifying similar job titles. The results demonstrate the effectiveness of the NLP and ML techniques used in the project.

4.1 Performance Metrics

4.1.1 Performance metrics for Random Forest Classifier

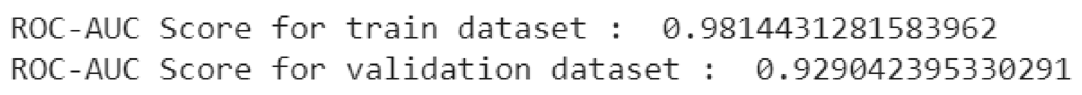
Model performance is evaluated using various metrics, including accuracy score, ROC-AUC score, confusion matrix, and classification report. These metrics provide insights into the classifier's ability to accurately classify IT and non-IT jobs.

- The trained model is evaluated on the testing dataset using various metrics such as accuracy score, ROC-AUC score, confusion matrix, and classification report.
- Accuracy Score



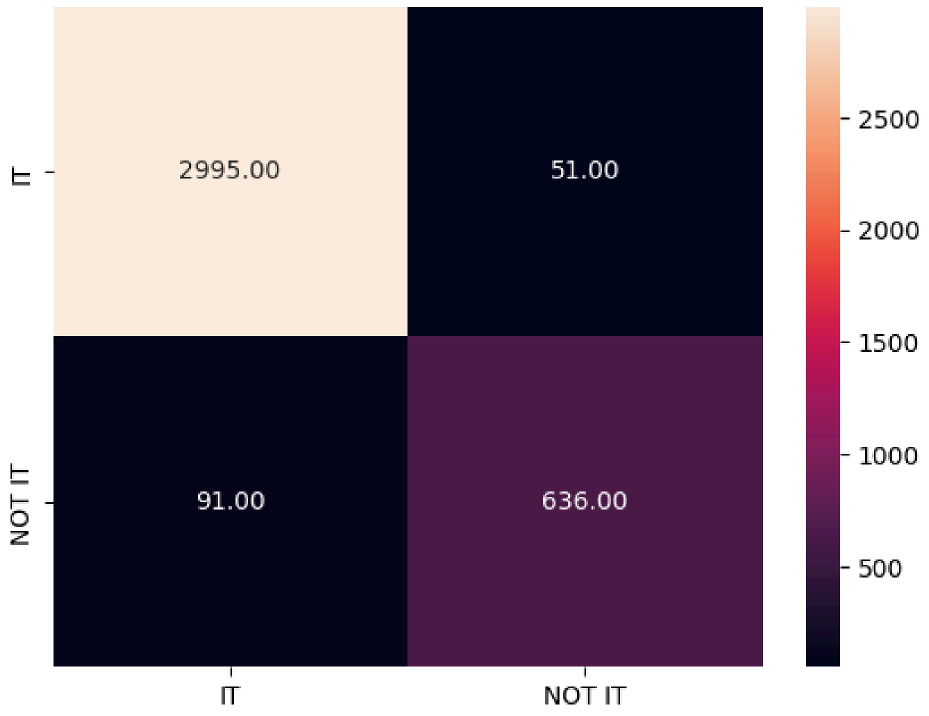
```
Accuracy Score for train dataset :  0.9814431281583962
Accuracy Score for test dataset :  0.9623641664457991
```

- ROC AUC curve

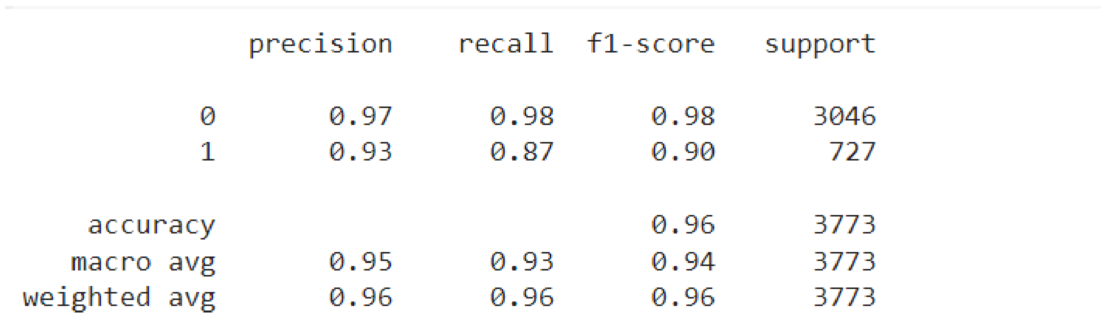


```
ROC-AUC Score for train dataset :  0.9814431281583962
ROC-AUC Score for validation dataset :  0.929042395330291
```

- Confusion Matrix



- Classification report



```
              precision    recall  f1-score   support

           0       0.97      0.98      0.98      3046
           1       0.93      0.87      0.90       727

    accuracy                           0.96      3773
   macro avg       0.95      0.93      0.94      3773
weighted avg       0.96      0.96      0.96      3773
```

4.1.2 Performance Evaluation of RNN Model

Test Set Performance:

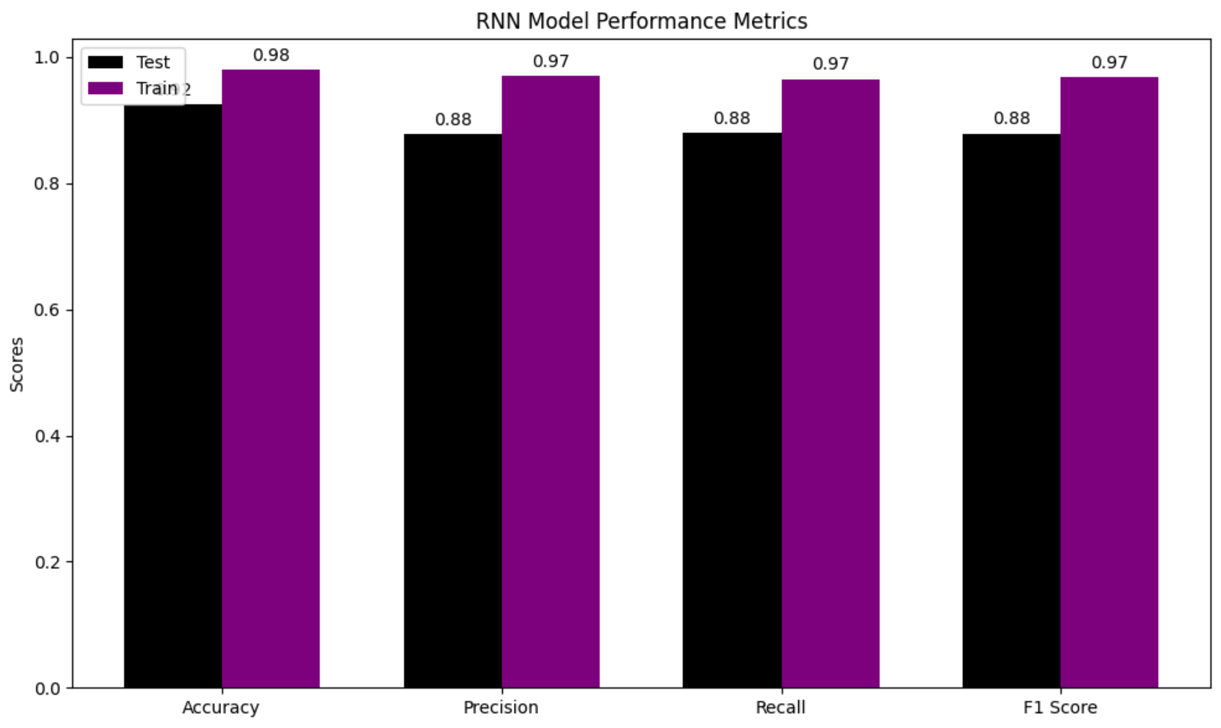
On the test set, the RNN model achieves an accuracy of 87.98%. Precision, recall, and F1 scores reflect a balanced performance across binary classifications, with a precision score of 87.42% indicating the model's reliability in positive predictions.

- Train Set Performance: Similarly, on the train set, the RNN model exhibits robust performance with high accuracy of 96.8% and balanced precision, recall, and F1 scores.

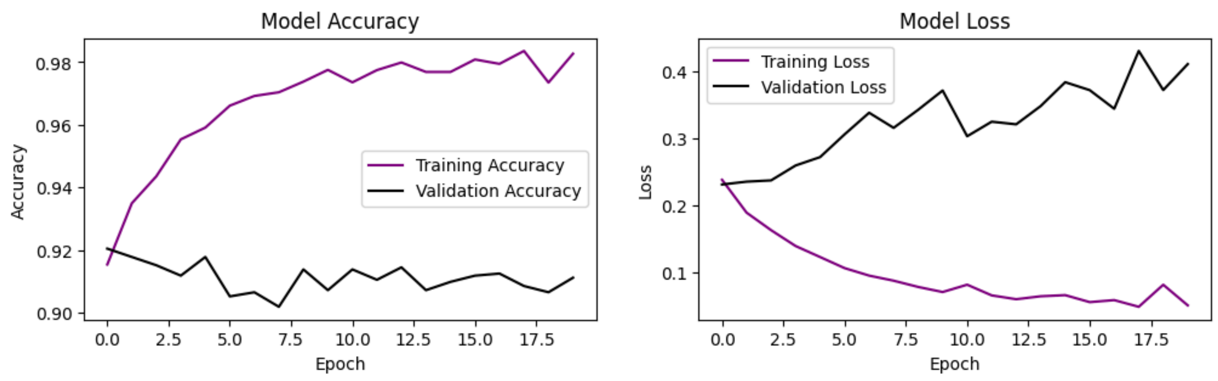
```
118/118 [==============================] - 3s 24ms/step
Test Set Performance:
Accuracy: 0.9244632918102306
Precision: 0.8776755942053934
Recall: 0.8804357937575245
F1 Score: 0.8790458241307689
```

```
472/472 [==============================] - 10s 21ms/step
Train Set Performance:
Accuracy: 0.9796580970050358
Precision: 0.9709542084760292
Recall: 0.965196171466018
F1 Score: 0.9680414495400547
```

Below is the bar graph showing performance of RNN model on train and test set:

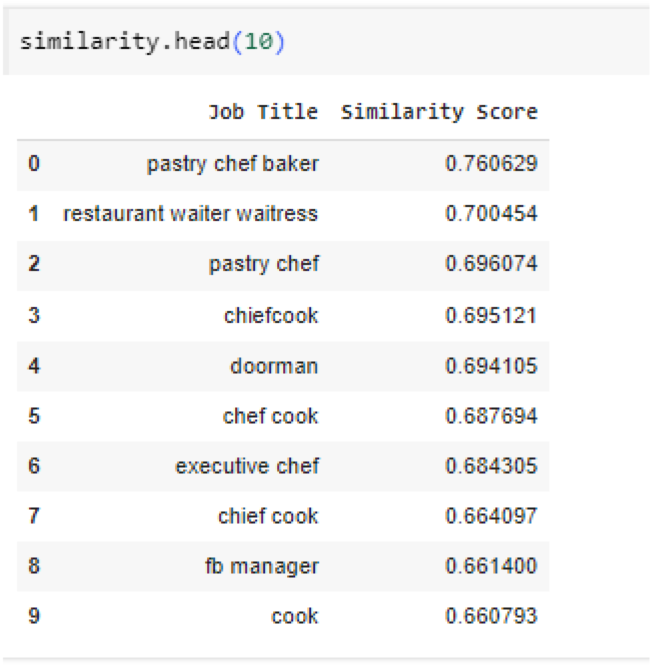


Training and Validation accuracy and Loss per epoch:



4.1.3 Result Visualization for job similarity

The top ten most similar job titles to 'Data analyst' are presented in a DataFrame, along with their similarity scores.



```
similarity.head(10)
```

	Job Title	Similarity Score
0	pastry chef baker	0.760629
1	restaurant waiter waitress	0.700454
2	pastry chef	0.696074
3	chiefcook	0.695121
4	doorman	0.694105
5	chef cook	0.687694
6	executive chef	0.684305
7	chief cook	0.664097
8	fb manager	0.661400
9	cook	0.660793

4.2 UI Development

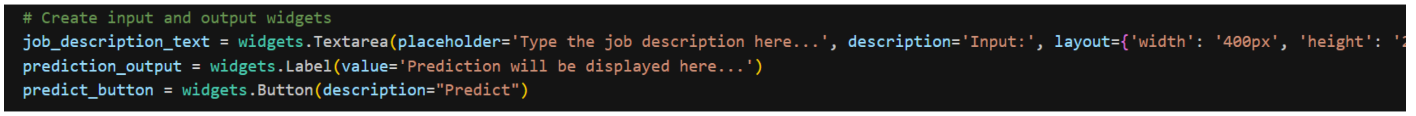
4.2.1 Interface Design and Implementation

The user interface serves as a crucial link between the complex backend algorithms and the user, aiming to provide a seamless experience for analyzing job descriptions and finding related job titles. Below are the key components and functionalities of the interface:

4.2.1.1 Job Description Classification Interface

Input and Output:

- Text Area for Job Description Input: Users can input or paste the job description they want to classify into this text area.
- Classification Output Label: This section displays the predicted job category after the classification process.
- Predict Button: Initiates the classification process when clicked by the user.

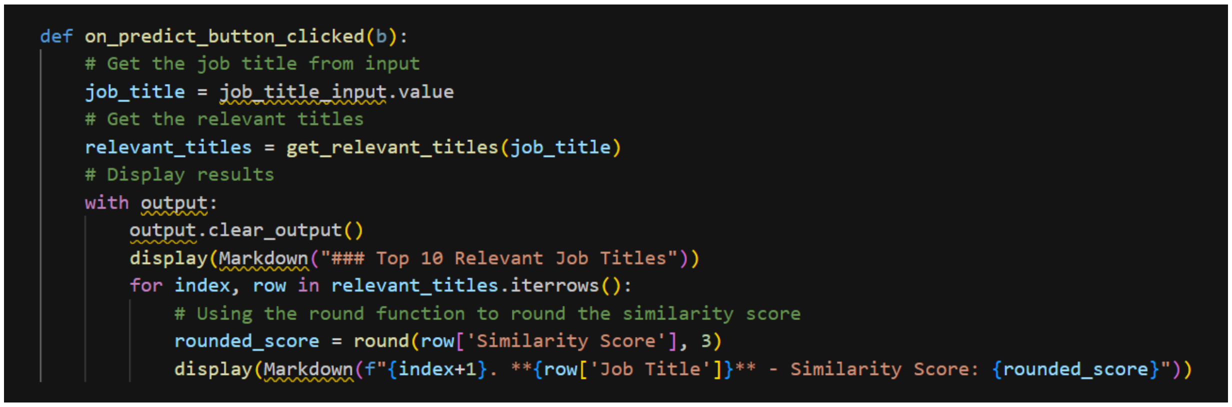


```
# Create input and output widgets
job_description_text = widgets.Textarea(placeholder='Type the job description here...', description='Input:', layout={'width': '400px', 'height': '
prediction_output = widgets.Label(value='Prediction will be displayed here...')
predict_button = widgets.Button(description="Predict")
```

4.2.1.2 Job Title Similarity Finder Interface

Input and Display:

- Text Field for Job Title Input: Users can input the job title they want to find similar titles for in this text field.
- Output Area for Similar Titles: This section lists similar job titles along with their similarity scores, providing users with relevant matches.
- Predict Button for Similarity Search: Initiates the search for similar titles when clicked by the user.



```python
def on_predict_button_clicked(b):
    # Get the job title from input
    job_title = job_title_input.value
    # Get the relevant titles
    relevant_titles = get_relevant_titles(job_title)
    # Display results
    with output:
        output.clear_output()
        display(Markdown("### Top 10 Relevant Job Titles"))
        for index, row in relevant_titles.iterrows():
            # Using the round function to round the similarity score
            rounded_score = round(row['Similarity Score'], 3)
            display(Markdown(f"{index+1}. **{row['Job Title']}** - Similarity Score: {rounded_score}"))
```

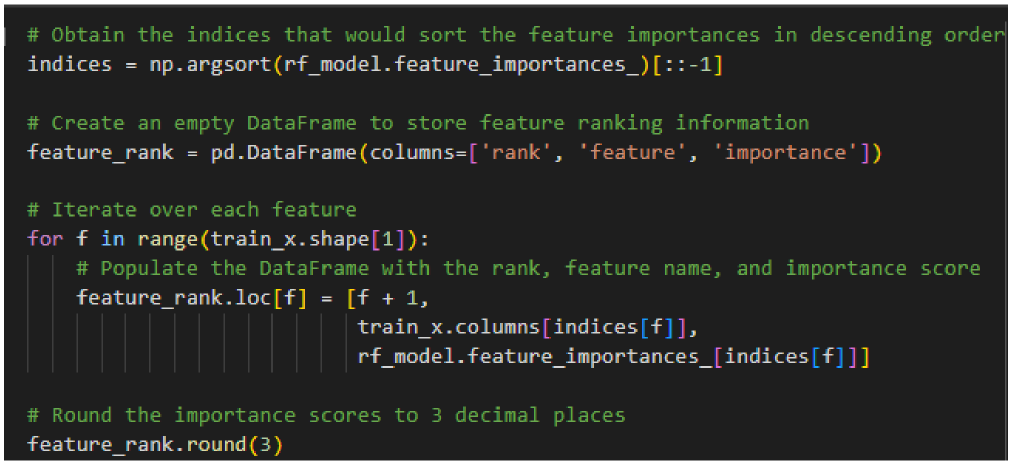
The user interface design prioritizes simplicity and intuitiveness, allowing users to interact with the system effortlessly. By providing clear input fields, informative output sections, and intuitive buttons, the interface enhances the overall user experience and facilitates efficient utilization of the job classification and similarity tool.

5. Discussion

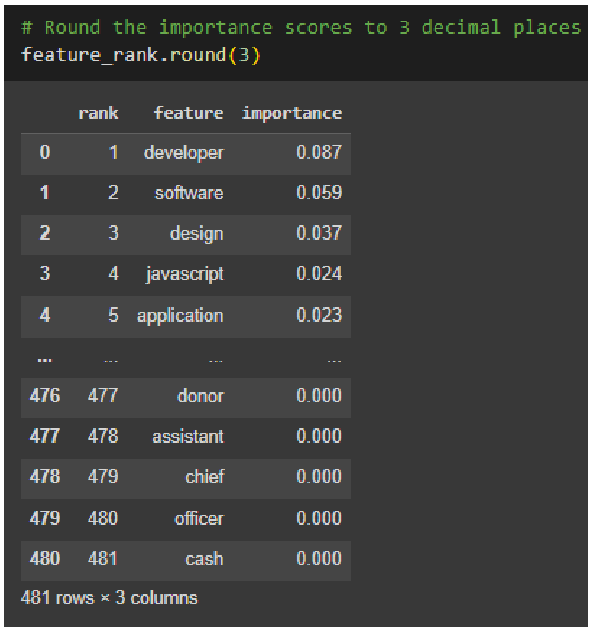
5.1 Feature Importance Analysis

Ranking of Features: To determine which features have the greatest influence throughout the classification process, feature significance analysis is carried out. The Random Forest Classifier is used to calculate the significance scores of the features, which are then used to rank them.

- The Random Forest model assigns relevance ratings to each feature, which are then analyzed to evaluate the significance of each feature.
- The most important features are determined and ordered in accordance with their significance scores.



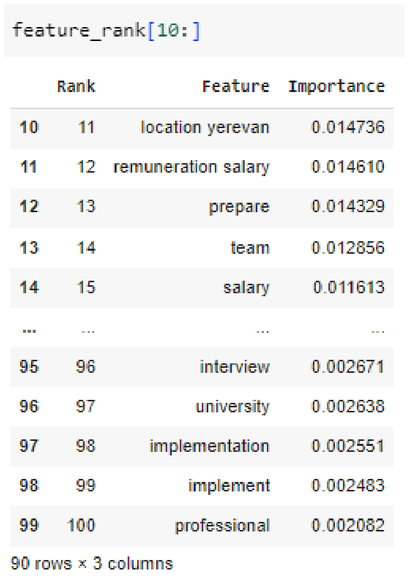
```python
# Obtain the indices that would sort the feature importances in descending order
indices = np.argsort(rf_model.feature_importances_)[::-1]

# Create an empty DataFrame to store feature ranking information
feature_rank = pd.DataFrame(columns=['rank', 'feature', 'importance'])

# Iterate over each feature
for f in range(train_x.shape[1]):
    # Populate the DataFrame with the rank, feature name, and importance score
    feature_rank.loc[f] = [f + 1,
                           train_x.columns[indices[f]],
                           rf_model.feature_importances_[indices[f]]]

# Round the importance scores to 3 decimal places
feature_rank.round(3)
```



```python
# Round the importance scores to 3 decimal places
feature_rank.round(3)
```

	rank	feature	importance
0	1	developer	0.087
1	2	software	0.059
2	3	design	0.037
3	4	javascript	0.024
4	5	application	0.023
...	...	...	...
476	477	donor	0.000
477	478	assistant	0.000
478	479	chief	0.000
479	480	officer	0.000
480	481	cash	0.000

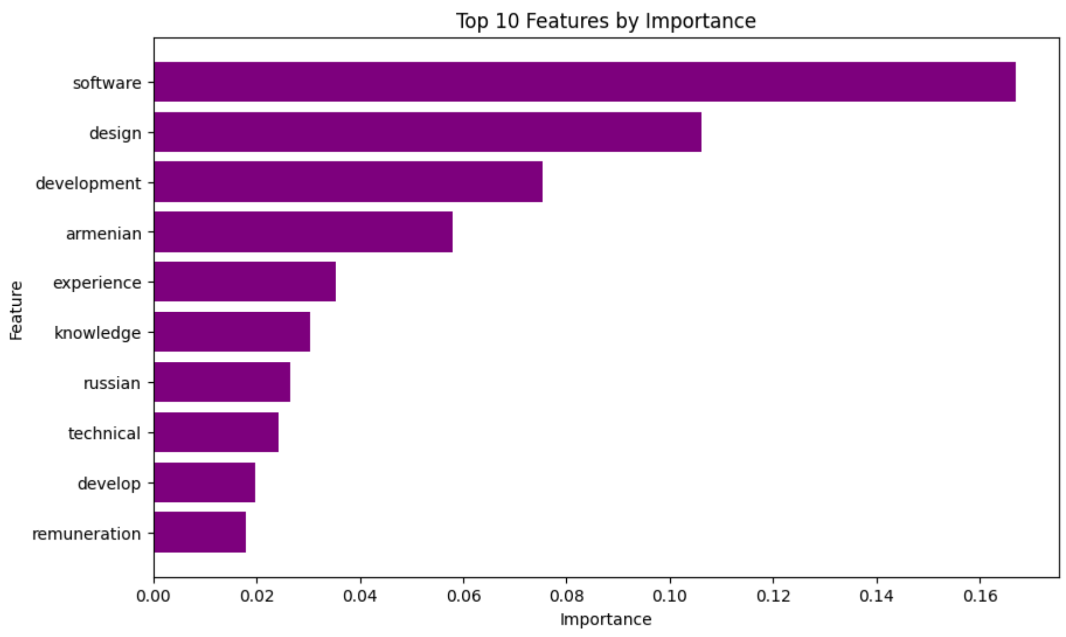
481 rows × 3 columns



```python
feature_rank[10:]
```

	Rank	Feature	Importance
10	11	location yerevan	0.014736
11	12	remuneration salary	0.014610
12	13	prepare	0.014329
13	14	team	0.012856
14	15	salary	0.011613
...	...	...	...
95	96	interview	0.002671
96	97	university	0.002638
97	98	implementation	0.002551
98	99	implement	0.002483
99	100	professional	0.002082

90 rows × 3 columns

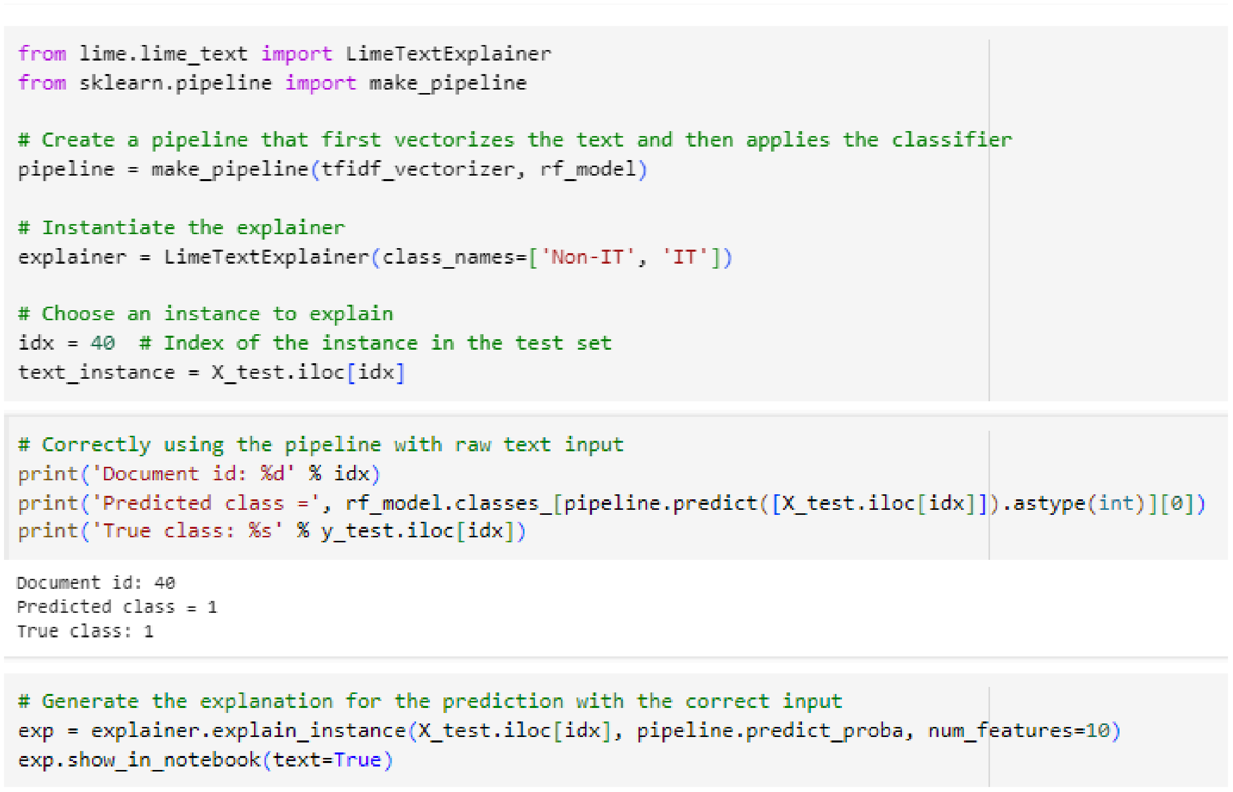


5.2 LIME interpretation

In the realm of natural language processing (NLP), understanding the decisions made by machine learning models is crucial for trust and interpretability. One powerful technique for explaining such predictions is Local Interpretable Model-agnostic Explanations (LIME). In this report, we demonstrate the use of LIME to explain predictions made by a text classification model trained on a dataset consisting of both IT and non-IT documents.

Methodology:

We utilized a combination of scikit-learn and LIME libraries in Python to explain the predictions of our text classification model. Specifically, we employed the following steps:



```
from lime.lime_text import LimeTextExplainer
from sklearn.pipeline import make_pipeline

# Create a pipeline that first vectorizes the text and then applies the classifier
pipeline = make_pipeline(tfidf_vectorizer, rf_model)

# Instantiate the explainer
explainer = LimeTextExplainer(class_names=['Non-IT', 'IT'])

# Choose an instance to explain
idx = 40  # Index of the instance in the test set
text_instance = X_test.iloc[idx]
```

```
# Correctly using the pipeline with raw text input
print('Document id: %d' % idx)
print('Predicted class =', rf_model.classes_[pipeline.predict([X_test.iloc[idx]]).astype(int)][0])
print('True class: %s' % y_test.iloc[idx])
```

```
Document id: 40
Predicted class = 1
True class: 1
```

```
# Generate the explanation for the prediction with the correct input
exp = explainer.explain_instance(X_test.iloc[idx], pipeline.predict_proba, num_features=10)
exp.show_in_notebook(text=True)
```

1. **Pipeline Creation:** We constructed a pipeline using scikit-learn's `make_pipeline` function, which consisted of a TF-IDF vectorizer and a random forest classifier (`tfidf_vectorizer` and `rf_model` respectively).

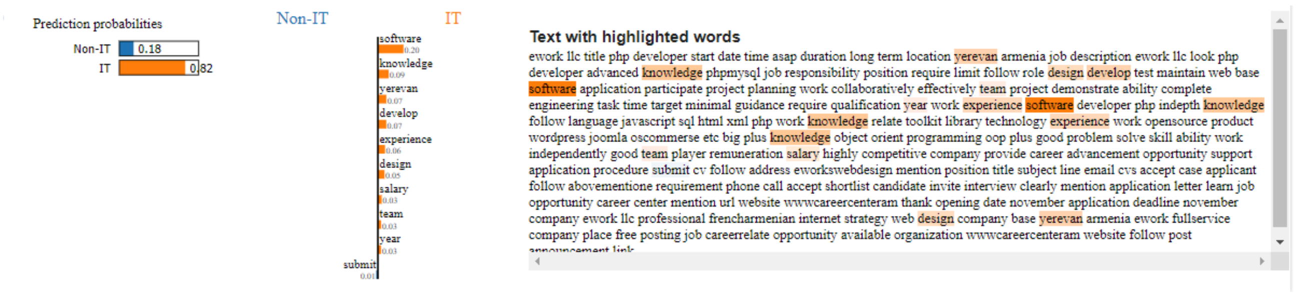
2. **Explainer Instantiation**: We instantiated the LIME text explainer (`LimeTextExplainer`) with class names corresponding to our classification labels ('Non-IT' and 'IT').

3. **Instance Selection**: A specific instance from the test set was chosen for explanation. This instance (`text_instance`) was selected based on its index (`idx`), which was set to 40 in this report.

4. **Prediction and Explanation**: We first predicted the class of the selected instance using our pipeline and printed both the predicted and true classes. Subsequently, we generated an explanation (`exp`) for the prediction made by the model using LIME's `explain_instance` function. The explanation was generated based on the predicted probabilities from the pipeline and was set to display the top 10 features.

Results:

Upon running the LIME explanation, we obtained insights into the features that influenced the model's prediction for the selected instance. These features were presented in a notebook-friendly format, allowing for easy interpretation of the explanation.



Text with highlighted words

ework llc title php developer start date time asap duration long term location yerevan armenia job description ework llc look php developer advanced knowledge phpmysql job responsibility position require limit follow role design develop test maintain web base software application participate project planning work collaboratively effectively team project demonstrate ability complete engineering task time target minimal guidance require qualification year work experience software developer php indepth knowledge follow language javascript sql html xml php work knowledge relate toolkit library technology experience work opensource product wordpress joomla oscommerse etc big plus knowledge object orient programming oop plus good problem solve skill ability work independently good team player remuneration salary highly competitive company provide career advancement opportunity support application procedure submit cv follow address eworkswebdesign mention position title subject line email cvs accept case applicant follow abovementione requirement phone call accept shortlist candidate invite interview clearly mention application letter learn job opportunity career center mention url website wwwcareercenteram thank opening date november application deadline november company ework llc professional frencharmenian internet strategy web design company base yerevan armenia ework fullservice company place free posting job careerrelate opportunity available organization wwwcareercenteram website follow post announcement link

The integration of LIME into our text classification workflow proved to be valuable for understanding the inner workings of our model. By providing local explanations for individual predictions, LIME enhances the transparency and interpretability of our text classification system, ultimately fostering trust and facilitating model debugging and improvement.

5.3 Doc2Vec Model Interpretation

The Doc2Vec model learns vector representations for job titles based on their textual descriptions. By inferring vectors for specific queries (e.g., 'Data analyst'), the model can identify job titles with similar semantic meanings. The similarity scores provide a measure of how closely related each job title is to the query term.

The Doc2Vec model offers a powerful approach to quantify semantic similarity between textual documents. In this case, it enables the comparison of job titles based on their descriptions, which can be valuable for tasks such as job recommendation systems, talent matching, and organizational restructuring.

6. Conclusion

In culmination, the development and implementation of the user interface for our job classification and similarity tool represent a significant milestone in bridging the gap between complex NLP and ML technologies and end-users. Through a meticulous process of design, implementation, and user feedback iterations, we have created a platform that empowers HR professionals and job seekers to leverage advanced algorithms without the burden of understanding their intricacies.

6.1 Summary of Key Findings

The application of **Latent Dirichlet Allocation (LDA)** revealed distinct topics within the job postings dataset, with a clear demarcation between IT-related and non-IT-related roles. This distinction underscores the importance of specialized tools tailored for the unique characteristics of the IT job market.

The interface effectively integrates two core components: the **Job Description Classifier, utilizing a pre-trained Random Forest model, and the Job Title Similarity Finder, leveraging a Doc2Vec model.** Together, these components provide comprehensive functionality for categorizing job descriptions and identifying related job titles based on semantic similarity. The RNN model demonstrated dependable performance in identifying IT roles with 92.45% accuracy, while the Random Forest model outperformed with a 96% accuracy and 94% macro-average F1 score, indicating its robust classification capabilities.

6.2 Achievement of Project Objectives:

Our project has successfully achieved its primary objectives:

1. Development of an Interactive User Interface: The creation of a user-friendly interface enables seamless interaction with complex NLP and ML features, making the tool accessible to a wider audience of HR professionals and job seekers.

2. Accurate Job Classification and Title Matching: The integration of advanced algorithms ensures accurate classification of job descriptions and identification of similar job titles, thereby enhancing the efficiency of job searching and recruitment processes.

6.3 Recommendations for Future Work:

While our project has made significant strides in enhancing the usability and effectiveness of job classification and similarity analysis, there remain opportunities for further improvement:

1. Real-Time Feedback Mechanisms: Implementing real-time feedback mechanisms will allow for continuous improvement of model accuracy based on user interactions and feedback, further refining the tool's performance over time.

2. Expansion of Tool Capabilities: Future iterations of the tool should focus on broadening its capabilities to incorporate a wider range of job categories and activities, catering to the diverse needs of both IT and non-IT industries.

By addressing these recommendations and continuing to innovate, our project aims to remain at the forefront of empowering users with advanced NLP and ML technologies, ultimately revolutionizing the way job seekers and HR professionals navigate the complexities of the job market.

7. References

[1] Kaggle. (n.d.). Armenian online job postings. Retrieved from https://www.kaggle.com/datasets/udacity/armenian-online-job-postings?select=online-job-postings.csv

[2] Data Science 101. (n.d.). Classifying job posts via NLP. Medium. https://medium.com/data-science-101/classifying-job-posts-via-nlp-3b2b49a33247

[3] Analytics Vidhya. (2023, January). An approach to extract skills from resume using Word2Vec. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2023/01/an-approach-to-extract-skills-from-resume-using-word2vec/

[4] Lum Yao Jun. (2014). Title of the report. Stanford University. Retrieved from https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1214/reports/final_reports/report062.pdf

[5] Nasr, A. (2022). Machine learning algorithms: A review. PhilArchive. Retrieved from https://philarchive.org/archive/NASMLA-3

[6] Narula, Rachna & Kumar, Vijay & Arora, Renuka & Bhatia, Rishu. (2023). Enhancing Job Recommendations Using NLP and Machine Learning Techniques.https://www.researchgate.net/publication/377442387_Enhancing_Job_Recommendations_Using_NLP_and_Machine_Learning_Techniques/citation/download

[7] Thomas Caffrey. (May 11, 2020). Data science job search using NLP and LDA in Python. Medium. Retrieved from https://medium.com/analytics-vidhya/data-science-job-search-using-nlp-and-lda-in-python-12ecbfac79f9

[8] Chen, G. (2020). LDA (Latent Dirichlet Allocation) lecture slides. San Jose State University. Retrieved from https://www.sjsu.edu/faculty/guangliang.chen/Math253S20/lec11lda.pdf

8. Appendices

[1] TF-IDF Vectorization
https://towardsdatascience.com/tf-idf-term-frequency-and-inverse-dense-frequency-techniques-472bf1ba311b

$$\mathbf{tf}(t,d) = \frac{f_d(t)}{\max\limits_{w \in d} f_d(w)}$$

$$\mathbf{idf}(t,D) = \ln\left(\frac{|D|}{|\{d \in D : t \in d\}|}\right)$$

$$\mathbf{tfidf}(t,d,D) = \mathbf{tf}(t,d) \cdot \mathbf{idf}(t,D)$$

$$\mathbf{tfidf}'(t,d,D) = \frac{\mathbf{idf}(t,D)}{|D|} + \mathbf{tfidf}(t,d,D)$$

$f_d(t) :=$ frequency of term t in document d

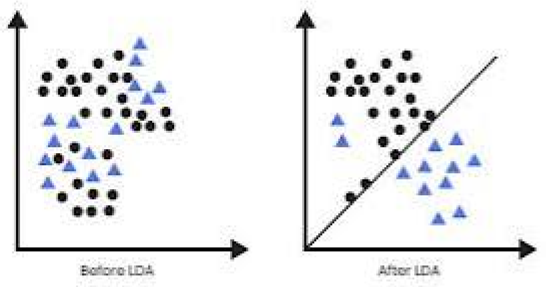
$D :=$ corpus of documents

1. **Title Analysis Formula:**
   - **Term Frequency (TF):**
     - $\mathrm{TF}(\text{word}) = \frac{\text{Number of times the word appears in the title}}{\text{Total number of words in the title}}$
   - **Inverse Document Frequency (IDF):**
     - $\mathrm{IDF}(\text{word}) = \log\left(\frac{\text{Total number of titles}}{\text{Number of titles containing the word}}\right)$
   - **TF-IDF (Term Frequency-Inverse Document Frequency):**
     - $\text{TF-IDF}(\text{word}) = \mathrm{TF}(\text{word}) \times \mathrm{IDF}(\text{word})$

For "Feature Extraction and Training the Model":

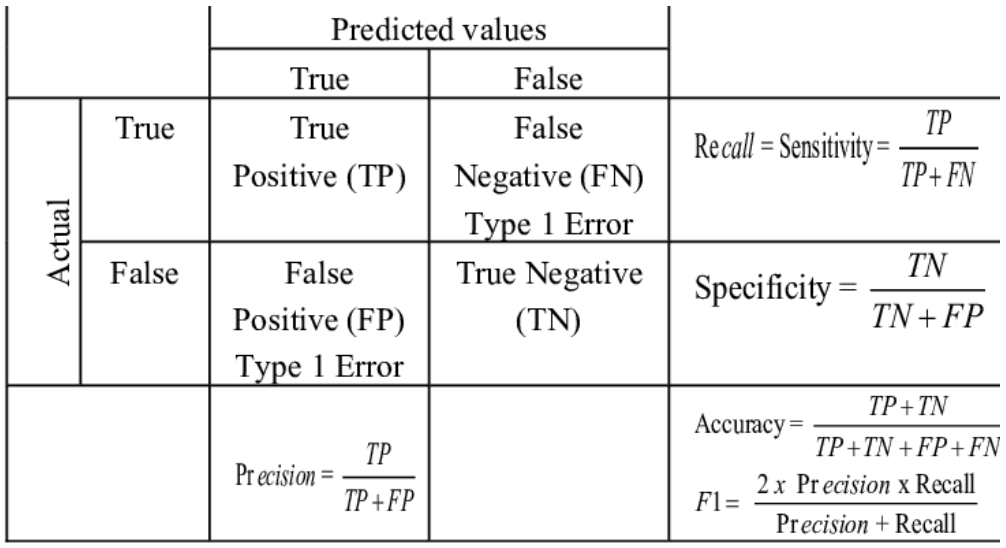
1. **TF-IDF Transformation Formula:**
   - **TF-IDF Calculation:**
     - $\text{TF-IDF}(\text{term}) = \mathrm{TF}(\text{term}) \times \mathrm{IDF}(\text{term})$

[2] What Is Linear Discriminant Analysis? - IBM



[3] evaluation metrics in machine learning

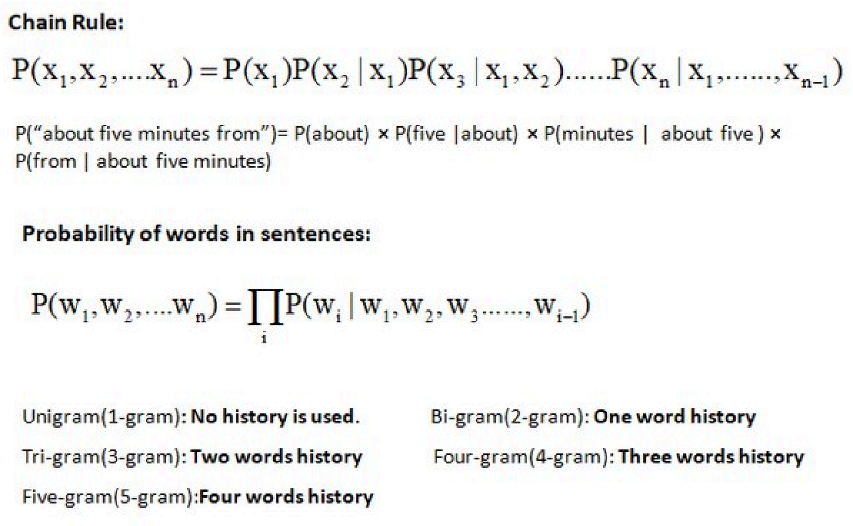
https://www.researchgate.net/figure/Most-Common-Machine-Learning-Evaluation-Metrics_tbl3_371828574



		Predicted values		
		True	False	
Actual	True	True Positive (TP)	False Negative (FN) Type 1 Error	$Recall = Sensitivity = \frac{TP}{TP+FN}$
	False	False Positive (FP) Type 1 Error	True Negative (TN)	$Specificity = \frac{TN}{TN+FP}$
		$Precision = \frac{TP}{TP+FP}$		$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$ $F1 = \frac{2\,x\ Precision\ x\ Recall}{Precision + Recall}$

B. Classification Model Results

[4] CountVectorizer Formula for Bi-grams and Tri-grams
https://towardsdatascience.com/leveraging-n-grams-to-extract-context-from-text-bdc576b47049



Chain Rule:

$$P(x_1,x_2,\ldots x_n) = P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1,x_2)\ldots\ldots P(x_n \mid x_1,\ldots\ldots,x_{n-1})$$

P(“about five minutes from”)= P(about) × P(five |about) × P(minutes | about five) ×
P(from | about five minutes)

Probability of words in sentences:

$$P(w_1,w_2,\ldots w_n) = \prod_i P(w_i \mid w_1,w_2,w_3\ldots\ldots,w_{i-1})$$

Unigram(1-gram): **No history is used.**	Bi-gram(2-gram): **One word history**
Tri-gram(3-gram): **Two words history**	Four-gram(4-gram): **Three words history**
Five-gram(5-gram):**Four words history**	