Job Description Similarity Analysis

Introduction

In this project, we aim to use Natural Language Processing (NLP) techniques to identify similarities between different job descriptions. This can improve the hiring process by matching jobs with suitable candidates more efficiently and identifying redundant or duplicated job postings. The project will involve data preprocessing, applying various algorithms, and analyzing the results.

Dataset

Chosen Dataset

I have selected a dataset from Kaggle containing job descriptions. The dataset includes fields such as job title, company, location, job description, and skills required.

Data Organization

The data has been organized into the following main parts:

- Job Title
- Company
- Location
- Job Description
- Skills Required

Data Preprocessing Techniques

To prepare the data for analysis, I applied the following preprocessing techniques:

1. **Data Cleaning**: Removing HTML tags, punctuation, and special characters from the job descriptions.
2. **Tokenization**: Splitting text into individual words or tokens.
3. **Lowercasing**: Converting all text to lowercase to maintain consistency.
4. **Stop Words Removal**: Eliminating common words that do not contribute to the meaning (e.g., "and", "the").
5. **Lemmatization**: Converting words to their base forms (e.g., "running" to "run").

Considered Algorithms and Implementations

We considered and implemented the following algorithms to analyze job description similarities:

1. **TF-IDF Vectorization**: Transforming text data into numerical features using Term Frequency-Inverse Document Frequency.

2. **Cosine Similarity**: Measuring the cosine of the angle between two non-zero vectors, providing a similarity score between job descriptions.
3. **Word Embeddings (Word2Vec)**: Representing words in a continuous vector space where semantically similar words are closer together.
4. **Doc2Vec**: Extending Word2Vec to capture the context of the whole document (job description).

Scalability of the Proposed Solution

Our solution is designed to scale with larger datasets by:

1. Using efficient vectorization techniques (TF-IDF, Word2Vec).
2. Implementing parallel processing where applicable.
3. Utilizing cloud-based solutions such as Google Colab for computationally intensive tasks.

Experiments

Description of Experiments

We conducted the following experiments:

1. **Baseline TF-IDF Similarity**: Calculated similarity scores using TF-IDF and cosine similarity.
2. **Word2Vec Similarity**: Trained a Word2Vec model on the job descriptions and calculated similarity scores.
3. **Doc2Vec Similarity**: Used Doc2Vec for document-level embeddings and calculated similarity scores.

Experimental Results

- **TF-IDF Similarity**: Provided a good baseline with clear distinctions between similar and dissimilar job descriptions.
- **Word2Vec Similarity**: Improved the similarity measurement by capturing semantic meanings.
- **Doc2Vec Similarity**: Offered the best results by considering the context of the entire job description.

Discussion

Our experiments show that using advanced NLP techniques like Word2Vec and Doc2Vec can significantly enhance the similarity measurement between job descriptions. These methods outperform traditional TF-IDF in capturing the contextual and semantic relationships.

Replicability

The experiments are replicable using the provided Jupyter notebook, which includes all necessary code and detailed explanations.

Scalability

The use of scalable NLP techniques ensures that our solution can handle larger datasets efficiently.

Declaration

“We declare that this material, which we now submit for assessment, is entirely our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of our work, and including any code produced using generative AI systems. We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should we engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by us or any other person for assessment on this or any other course of study.”

Submission Details

- **Names and Student IDs**: Mehrad MosanenMozafari
- **Enrollment**: Master in Data Science for Economics
- **GitHub Link**: https://github.com/MahradMozafari

Dataset and Organization

Data Preprocessing

To clean up the job descriptions, I applied several preprocessing techniques using various methods to remove noise, convert text to lowercase, remove stop words, and lemmatize words.

Noise Removal

1. **Regex**: Removed digits and punctuation as well as normalized whitespace.
2. **HTML Tags**: Used BeautifulSoup to remove HTML tags and Unicode data to normalize text.

Convert to Lowercase

1. **Basic**: Converted text directly to lowercase.
2. **Spacy**: Used Spacy tokens to ensure consistent lowercase conversion.

Remove Stop Words

1. **NLTK**: Removed stop words using NLTK's list of English stop words.
2. **Spacy**: Removed stop words using Spacy's built-in stopword list.

Lemmatization

1. **NLTK**: Used NLTK's WordNetLemmatizer.

2. **Spacy**: Used Spacy lemmatizer.
3. **TextBlob**: Used TextBlob's vocabulary capabilities.

Stemming

1. **Porter**: Applied Porter's stemming algorithm.
2. **Snowball**: Used the Snowball stemmer.
3. **Lancaster**: Implemented Lancaster's original algorithm.

Feature Extraction

I explored several methods for feature extraction from text data.

1. **TF-IDF**: TF-IDF features were extracted using basic and advanced Spacy methods.
2. **Bag of Words**: Created a bag of words model.

Similarity Detection

To detect similarities between job descriptions, I considered the following methods:

1. **Cosine Similarity**: Calculated between TF-IDF vectors.
2. **Jaccard Similarity**: Measured between sets of words.
3. **Edit Distance**: Calculated using NLTK.

Evaluation of Methods

I used K-fold cross-validation with 5 folds to evaluate the performance of various preprocessing, feature extraction, and similarity detection methods, measuring performance using F1 score, precision, and recall.

Selected Methods

Based on our review, I selected the best-performing methods for each step:

- **Preprocessing**: Best methods for noise removal, lowercase conversion, stop word removal, and lemmatization.
- **Feature Extraction**: TF-IDF.
- **Similarity Detection**: Cosine similarity.

Word2Vec Embeddings

I also explored Word2Vec embeddings for job descriptions. I trained the Word2Vec model on cleaned job descriptions, calculated word embeddings, and similarity scores.

Scalability

The proposed solution scales well to larger datasets. The preprocessing and feature extraction methods are efficient, and the use of cosine similarity ensures computational feasibility for large datasets.

Experiments and Results

I conducted experiments using the best-selected methods. The job descriptions were preprocessed, features were extracted, and similarities were calculated. Performance was evaluated using F1 score, precision, and recall. The results show that the selected methods perform well in identifying similar job descriptions, with TF-IDF and cosine similarity providing accurate and scalable similarity detection.

Conclusion

The combination of TF-IDF and cosine similarity methods allows for effective and scalable similarity detection in job descriptions.

Code Explanation

Import Libraries and Set Environment Variables

```
import os
import pandas as pd
from zipfile import ZipFile

os.environ['KAGGLE_USERNAME'] = "your_kaggle_username"
os.environ['KAGGLE_KEY'] = "your_kaggle_api_key"
```

These lines define the environment variables for your Kaggle username and API key. Replace placeholders with your actual Kaggle credentials to authenticate using the Kaggle API.

Unzip the Dataset

```
with ZipFile('dataset_name.zip', 'r') as zip_ref:
    zip_ref.extractall('data')
```

This block extracts the contents of the downloaded ZIP file into a folder named `data`.

Load Data into a Pandas DataFrame

```
df = pd.read_csv('data/job_summary.csv')
```

This line reads a CSV file named `job_summary.csv` from the `data` directory and loads it into a pandas DataFrame named `df`.

Define a Method Dictionary for Each Step

The method dictionary organizes different preprocessing, feature extraction, and similarity detection methods into categories.

```
methods = {
    "preprocessing": {
        "remove_noise": [remove_noise_regex, remove_noise_html,
remove_numbers, remove_punctuation],
        "lowercase": [to_lowercase, to_lowercase_spacy],
        "remove_stopwords": [remove_stopwords_nltk, remove_stopwords_spacy],
        "lemmatize": [lemmatize_nltk, lemmatize_spacy, lemmatize_textblob],
        "stem": [stem_porter, stem_snowball, stem_lancaster]
    },
    "feature_extraction": {
        "tf_idf": [tf_idf_feature_extraction, tf_idf_spacy],
        "bag_of_words": [bag_of_words_feature_extraction]
    },
    "similarity_detection": {
        "cosine_similarity": [cosine_similarity],
        "jaccard_similarity": [jaccard_similarity],
        "edit_distance": [edit_distance]
    }
}
```

Measure Time Function

```
def measure_time(func, text_series):
    start_time = time.time()
    text_series.apply(func)
    return time.time() - start_time
```

This function measures the time taken to apply a function to a pandas series containing text data.

Preprocessing Functions Explanation

This section contains various text preprocessing functions. These functions clean and normalize text data.

Remove Noise Using Regex

```
def remove_noise_regex(text):
    text = re.sub(r'\d+', '', text)
    text = text.translate(str.maketrans('', '', string.punctuation))
    text = re.sub(r'\s+', ' ', text)
    return text
```

Remove Noise Using HTML Parsing

```
def remove_noise_html(text):
    soup = BeautifulSoup(text, "html.parser")
    text = soup.get_text()
    text = unicodedata.normalize("NFKD", text)
    text = re.sub(r'\s+', ' ', text)
    return text
```

Remove Numbers

```
def remove_numbers(text):
    return re.sub(r'\d+', '', text)
```

Remove Punctuation

```
def remove_punctuation(text):
    return text.translate(str.maketrans('', '', string.punctuation))
```

Convert to Lowercase

```
def to_lowercase(text):
    return text.lower()
```

Convert to Lowercase Using Spacy

```
def to_lowercase_spacy(text):
    doc = nlp(text)
    return ' '.join([token.text.lower() for token in doc])
```

Remove Stopwords Using NLTK

```
def remove_stopwords_nltk(text):
    words = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_words = [word for word in words if word not in stop_words]
    return ' '.join(filtered_words)
```

Remove Stopwords Using Spacy

```
def remove_stopwords_spacy(text):
    doc = nlp(text)
    filtered_words = [token.text for token in doc if not token.is_stop]
    return ' '.join(filtered_words)
```

Lemmatize Using NLTK

```
def lemmatize_nltk(text):
    words = word_tokenize(text)
    lemmatizer = WordNetLemmatizer()
    lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
    return ' '.join(lemmatized_words)
```

Lemmatize Using Spacy

```
def lemmatize_spacy(text):
    doc = nlp(text)
    lemmatized_words = [token.lemma_ for token in doc]
    return ' '.join(lemmatized_words)
```

Lemmatize Using TextBlob

```
def lemmatize_textblob(text):
    words = word_tokenize(text)
    lemmatized_words = [Word(word).lemmatize() for word in words]
    return ' '.join(lemmatized_words)
```

Stem Using Porter Stemmer

```
def stem_porter(text):
    words = word_tokenize(text)
    porter_stemmer = PorterStemmer()
    stemmed_words = [porter_stemmer.stem(word) for word in words]
    return ' '.join(stemmed_words)
```

Stem Using Snowball Stemmer

```
def stem_snowball(text):
    words = word_tokenize(text)
    snowball_stemmer = SnowballStemmer(language='english')
    stemmed_words = [snowball_stemmer.stem(word) for word in words]
    return ' '.join(stemmed_words)
```

Stem Using Lancaster Stemmer

```
def stem_lancaster(text):
    words = word_tokenize(text)
    lancaster_stemmer = LancasterStemmer()
    stemmed_words = [lancaster_stemmer.stem(word) for word in words]
    return ' '.join(stemmed_words)
```

Feature Extraction Functions Explanation

This section contains functions for extracting features from the cleaned text data.

TF-IDF Feature Extraction

```
def tf_idf_feature_extraction(corpus):
    vectorizer = TfidfVectorizer()
    X = vectorizer.fit_transform(corpus)
    return X
```

Bag of Words Feature Extraction

```
def bag_of_words_feature_extraction(corpus):
    vectorizer = CountVectorizer()
    X = vectorizer.fit_transform(corpus)
    return X
```

Similarity Detection Functions Explanation

This section contains functions for detecting similarities between job descriptions.

Cosine Similarity

```
def cosine_similarity(X):
    return cosine_similarity(X)
```

Jaccard Similarity

```
def jaccard_similarity(str1, str2):
    a = set(str1.split())
    b = set(str2.split())
    return float(len(a.intersection(b)) / len(a.union(b)))
```

Edit Distance

```
def edit_distance(str1, str2):
    return distance.edit_distance(str1, str2)
```

Training and Evaluating Word2Vec Embeddings

```
from gensim.models import Word2Vec

# Create sentences from job descriptions
sentences = df['job_description'].apply(lambda x: word_tokenize(x.lower()))

# Train Word2Vec model
model = Word2Vec(sentences, vector_size=100, window=5, min_count=1,
workers=4)

# Get Word2Vec embeddings
embeddings = df['job_description'].apply(lambda x:
model.wv[word_tokenize(x.lower())])
```

This block trains a Word2Vec model on job descriptions and extracts embeddings.

Combining Methods for Final Processing

This section combines the selected methods for each preprocessing step, feature extraction, and similarity detection, applying them to the dataset.

```
# Preprocess text data
df['cleaned_description'] =
df['job_description'].apply(remove_noise_html).apply(to_lowercase).apply(remo
ve_stopwords_spacy).apply(lemmatize_spacy)

# Extract features
tf_idf_matrix = tf_idf_feature_extraction(df['cleaned_description'])
```

```
# Calculate similarities
similarities = cosine_similarity(tf_idf_matrix)
```

Conclusion

The combination of TF-IDF and cosine similarity methods allows for effective and scalable similarity detection in job descriptions. This approach ensures accurate identification of similar job descriptions, aiding job seekers and recruiters.