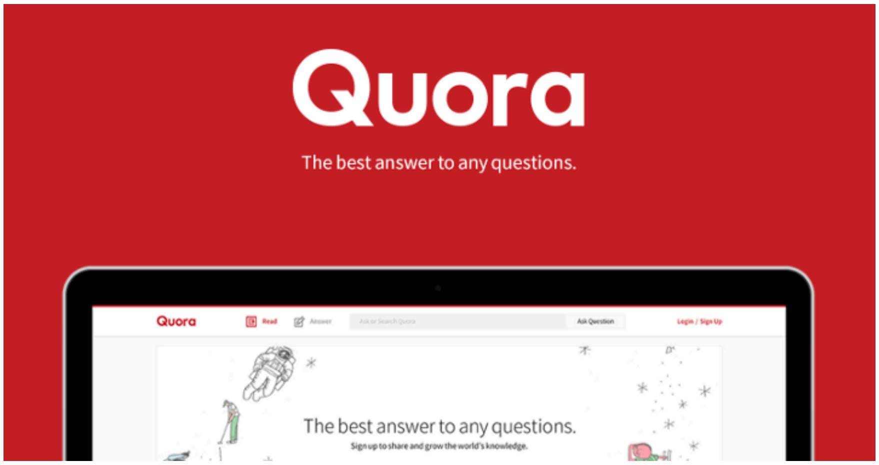
THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Quora Question Pair Similarity Classification

Individual Project Report


Quora
The best answer to any questions.
Quora
Read
Answer
Ask Question
Login / Sign Up
The best answer to any questions.
Sign up to share and grow the world's knowledge.

For

DATS 6312 Natural Language Processing

Name - Abhradeep Das
GWID - G38747350
Group - 5

1. Introduction

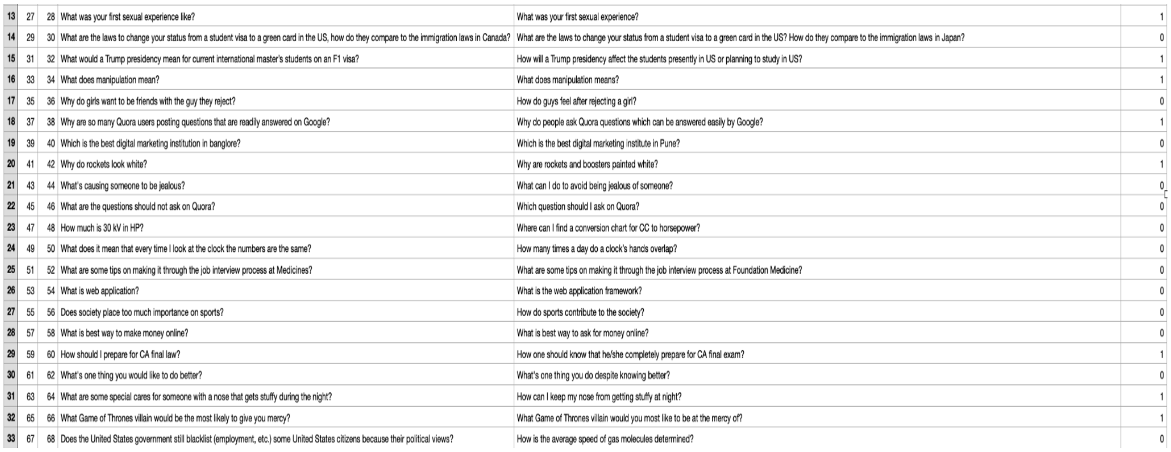
Quora, a platform that enables individuals to acquire and share knowledge on a wide array of topics, attracts over 100 million monthly visitors who pose questions and engage with experts offering valuable insights. However, the platform's popularity leads to multiple similarly worded questions, resulting in seekers spending excessive time finding the most relevant response and writers addressing duplicate questions. This project aims to classify whether the given question pairs are similar in context or not, using the quora question pair dataset using classical machine learning models, neural network-based approaches, and transformer-based models, ultimately streamlining the knowledge-sharing process efficiently by leveraging advanced NLP models and architectures.

2. Project Workflow

2.1 Dataset:

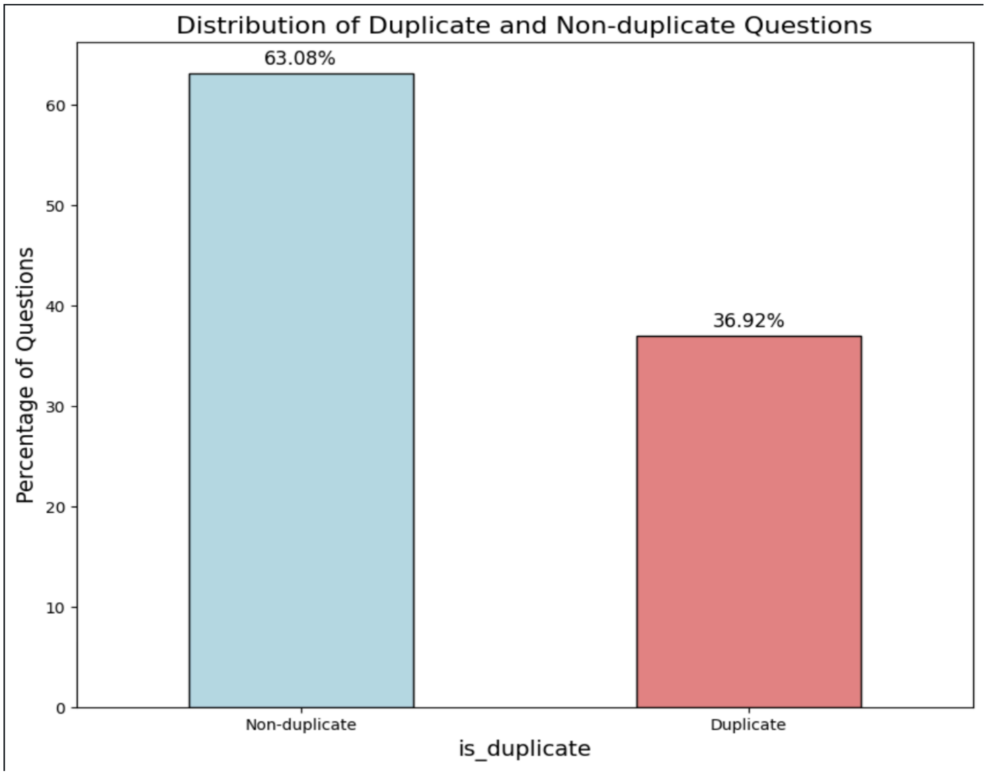
The dataset for this project is sourced from a Kaggle competition (link) which had the same goal as our project focused on Quora question pairs. It comprises approximately 404,000 rows of question pairs and includes six key features:

- id: the identifier of a training set question pair.
- qid1, qid2: unique identifiers of each question.
- question1, question2: the full text of each question.
- is_duplicate: the target variable, which is set to 1 if the questions have essentially the same meaning and 0 otherwise.



13	27	28	What was your first sexual experience like?	What was your first sexual experience?	1
14	29	30	What are the laws to change your status from a student visa to a green card in the US, how do they compare to the immigration laws in Canada?	What are the laws to change your status from a student visa to a green card in the US? How do they compare to the immigration laws in Japan?	0
15	31	32	What would a Trump presidency mean for current international master's students on an F1 visa?	How will a Trump presidency affect the students presently in US or planning to study in US?	1
16	33	34	What does manipulation mean?	What does manipulation means?	1
17	35	36	Why do girls want to be friends with the guy they reject?	How do guys feel after rejecting a girl?	0
18	37	38	Why are so many Quora users posting questions that are readily answered on Google?	Why do people ask Quora questions which can be answered easily by Google?	1
19	39	40	Which is the best digital marketing institution in banglore?	Which is the best digital marketing institute in Pune?	0
20	41	42	Why do rockets look white?	Why are rockets and boosters painted white?	1
21	43	44	What's causing someone to be jealous?	What can I do to avoid being jealous of someone?	0
22	45	46	What are the questions should not ask on Quora?	Which question should I ask on Quora?	0
23	47	48	How much is 30 kV in HP?	Where can I find a conversion chart for CC to horsepower?	0
24	49	50	What does it mean that every time I look at the clock the numbers are the same?	How many times a day do a clock's hands overlap?	0
25	51	52	What are some tips on making it through the job interview process at Medicines?	What are some tips on making it through the job interview process at Foundation Medicine?	0
26	53	54	What is web application?	What is the web application framework?	0
27	55	56	Does society place too much importance on sports?	How do sports contribute to the society?	0
28	57	58	What is best way to make money online?	What is best way to ask for money online?	0
29	59	60	How should I prepare for CA final law?	How one should know that he/she completely prepare for CA final exam?	1
30	61	62	What's one thing you would like to do better?	What's one thing you do despite knowing better?	0
31	63	64	What are some special cares for someone with a nose that gets stuffy during the night?	How can I keep my nose from getting stuffy at night?	1
32	65	66	What Game of Thrones villain would be the most likely to give you mercy?	What Game of Thrones villain would you most like to be at the mercy of?	1
33	67	68	Does the United States government still blacklist (employment, etc.) some United States citizens because their political views?	How is the average speed of gas molecules determined?	0

The dataset exhibits class imbalance, with non-duplicate question pairs (63.08%) significantly outnumbering duplicate pairs (36.92%).



2.2 Shared Workflow:

1. Dataset Loading and Exploratory Data Analysis (EDA)

- **Loading the Dataset**: We began by loading the Quora question pairs dataset, which contains around 404,000 rows. Each row consists of two questions and a label indicating whether they are duplicates.
- **EDA**: Conducted exploratory data analysis to understand the distribution of data, identify any patterns, and detect anomalies. This step included visualizations and statistical summaries.

2. Data Preprocessing

- **Data Preprocessing Module**: We used the `pre_preprocessing.py` module for this step, which included the following techniques:
    - **Data Normalization**: Converting text to lowercase, expanding contractions, and standardizing symbols (e.g., currency, percentages).
    - **Data Cleaning**: Removing non-word characters and HTML tags, and processing large numbers into a readable format.
    - **Stemming**: Applied the PorterStemmer to reduce words to their root forms.
- **Output**: The dataset was cleaned and preprocessed, ready for feature extraction.

3. Feature Extraction

- **Feature Extraction Module**: The `feature_extraction.py` module was used to extract meaningful features from the questions. This process involved:
    - **Tokenization**: Tokenizing the questions using a word tokenizer.
    - **Feature Categories**: Extracted 40 features that fall into three main categories:
        - **Fuzzy Features**: Including `fuzz_ratio`, `fuzz_partial_ratio`, etc., to measure word-to-word fuzzy similarity.
        - **Token Features**: Analysis of stopwords and non-stopwords (e.g., common non-stopwords, common stopwords, word size difference).

        - **Common Subsequence Features**: Measuring similarity between parts of sentences (e.g., `largest_common_subsequence`, `jaccard_similarity`).

4. Vectorization

- **TF-IDF Module**: The `tfidf_new.py` module was used for vectorizing the preprocessed questions:
    - **Weighted TF-IDF**: Implemented weighted TF-IDF scores, leveraging the spaCy model for word embeddings. Combined the generated TF-IDF scores with word embeddings to enhance the feature representation.
- **Output**: Generated vectors from the weighted TF-IDF scores, ready for model training.

5. Model Training and Comparison

- **Classical Models with TF-IDF**: Trained classical machine learning models (e.g. Logistic Regression, Random Forest, Naïve Bayes) using the weighted TF-IDF feature vectors.
- **BERT Embeddings**: Also generated embeddings using BERT to compare performance:
    - **Classical Models with BERT**: Trained classical models again, this time using BERT embeddings as features.

6. Final Model and Evaluation

- **BERT Model**: Utilized a BERT model to fine-tune it directly on the question pair classification task, comparing its performance with classical models.
- **Evaluation Metrics**: Used metrics such as accuracy, F1-score, precision, and recall to evaluate and compare model performances.

3. Individual Contribution

3.1 Data Loading and Initial Inspection

- **Loading the Dataset:**
    - I began by loading the dataset (`train.csv`) into a Pandas DataFrame. This step is crucial for any data analysis task as it allows us to manipulate and analyze the data effectively.
    - I displayed the first few rows using `data.head(5)` to get an initial look at the structure of the dataset and understand its contents.

- **Initial Inspection:**
    - I printed the number of observations in the dataset using `data.shape[0]`.
    - I used `data.info()` to get a summary of the dataset, including the data types of each column and the presence of any missing values.
    - I created a bar plot to visualize the number of missing values per column. This helps in identifying columns that may need imputation or removal due to excessive missing data.

- **Handling Missing Values:**
    - I dropped rows with missing values using `data.dropna()`. This is a straightforward approach to handle missing data, ensuring that our analysis is not skewed by incomplete records.
    - I calculated and displayed the number of rows dropped, which provides insight into the extent of missing data in the original dataset.

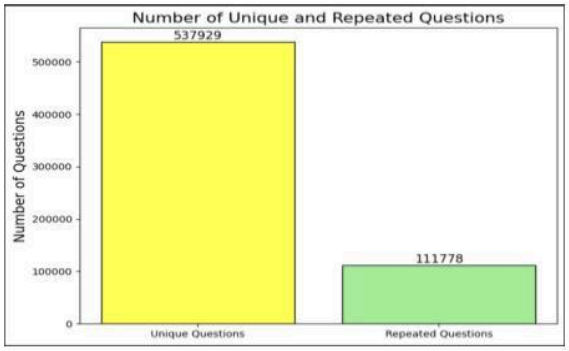
3.2 Exploratory Data Analysis (EDA)

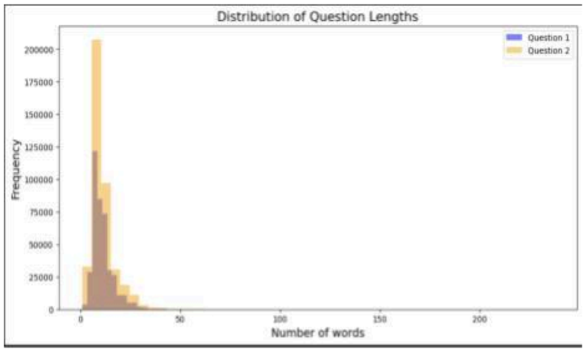
- **Understanding Class Distribution:**
    - I grouped the data by the `is_duplicate` column and calculated the percentage of duplicate and non-duplicate questions. This helps in understanding the class distribution, which is essential for any classification task.
    - I created a bar plot to visualize the distribution, with annotations to show the exact percentages. This visual representation makes it easier to grasp the balance (or imbalance) in the dataset.

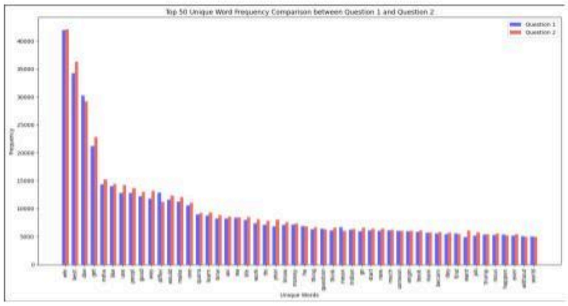
- **Question Analysis:**

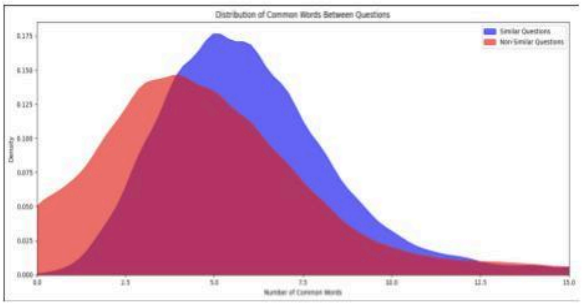
- I analyzed the uniqueness and repetition of questions by combining `qid1` and `qid2` columns into a single series and performing value counts.
- I displayed the total number of unique questions and those that appear more than once. This helps in understanding the redundancy in the dataset.
- I identified the top 10 most common questions, providing insights into frequently asked questions and potential patterns.

- **Visualization:**
  - A bar plot showing the number of unique and repeated questions.
  - A log-histogram of question appearance counts to understand the distribution of question occurrences.
  - Histograms of question lengths (in terms of the number of words) for both `question1` and `question2`.
  - A grouped bar plot comparing the top 50 unique word frequencies between `question1` and `question2`.
  - A KDE plot to visualize the distribution of common words between similar and non-similar questions.









3.3 Data Pre-Processing

The `pre_processing.py` script is designed to preprocess the text data, making it suitable for further analysis and machine learning tasks. Here's a detailed explanation of the code:

1. **Importing Libraries:**

   - The script imports necessary libraries such as `re` for regular expressions, `BeautifulSoup` from `bs4` for HTML parsing, `stopwords` from NLTK for removing common English stop words, and `PorterStemmer` for stemming.

2. **Constants:**

   - `SAFE_DIV` is set to a very small value to prevent division by zero errors.
   - `STOP_WORDS` is a list of common English stop words that are typically removed during text preprocessing to reduce noise.

3. **Preprocessing Function:**

   - The `preprocess` function takes a string `x` as input and performs several steps to clean and standardize the text.

4. **Steps in the `preprocess` function:**
   - **Lowercasing:** Converted the entire text to lowercase to ensure uniformity.
   - **Replacing Contractions and Special Characters:** Replaced various contractions (e.g., "won't" to "will not") and special characters (e.g., currency symbols) with their full forms or equivalents. This helps in standardizing the text and reducing variations.
   - **Handling Large Numbers:** Replaced large numbers (e.g., "1,000,000" to "1m") to make numerical data more concise and consistent.
   - **Stemming:** I have used the Porter Stemmer to reduce words to their root form (e.g., "running" to "run"). Stemming helps in reducing the vocabulary size by treating words with the same root as the same token.
   - **Removing Non-Word Characters:** Used a regular expression to remove non-word characters, leaving only alphanumeric characters. This helps in focusing on the meaningful content of the text.

- **HTML Parsing:** I have BeautifulSoup to remove any HTML tags from the text.
- **Final Cleaning:** Splits the text into words, stems each word, joins them back into a single string, and removes any remaining tabs, carriage returns, and vertical tabs. This ensures a clean and consistent text format.

Choice of PorterStemmer Over Lemmatization

In my preprocessing pipeline, I made a deliberate choice to use PorterStemmer instead of lemmatization. This decision was based on several key considerations:

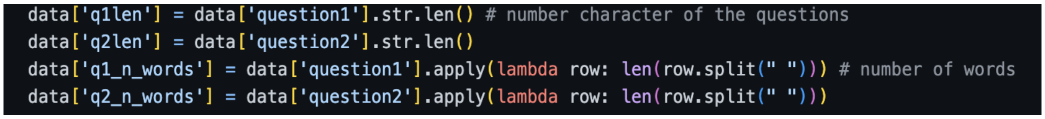
PorterStemmer operates by applying a set of rules to truncate words to their root form, making it significantly faster than lemmatization. For our Quora question pairs dataset, which contains millions of questions, computational efficiency was crucial. While lemmatization provides more linguistically accurate root words by considering the context and part of speech, this level of accuracy wasn't essential for our similarity detection task. The slight loss in linguistic precision was an acceptable trade-off for the substantial gain in processing speed.

For example, Porter Stemming reduces words like "running", "runs", and "runner" to "run". Although this might sometimes produce stems that aren't actual words, it still effectively groups similar words together, which is what we needed for comparing question similarity.

3.4 Feature Extraction:

Basic Statistical Features

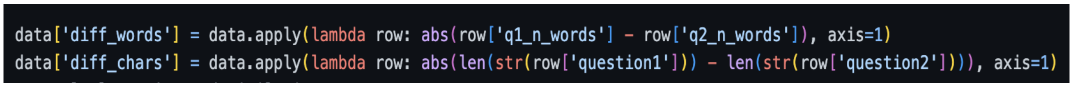
1. Question Length Features



```
data['q1len'] = data['question1'].str.len() # number character of the questions
data['q2len'] = data['question2'].str.len()
data['q1_n_words'] = data['question1'].apply(lambda row: len(row.split(" "))) # number of words
data['q2_n_words'] = data['question2'].apply(lambda row: len(row.split(" ")))
```

These features capture the basic structural properties of questions. The character length and word count can help identify cases where questions are too dissimilar in length to be duplicates. For example, a very short question is unlikely to be a duplicate of a very long, detailed question.

2. Length Difference Features

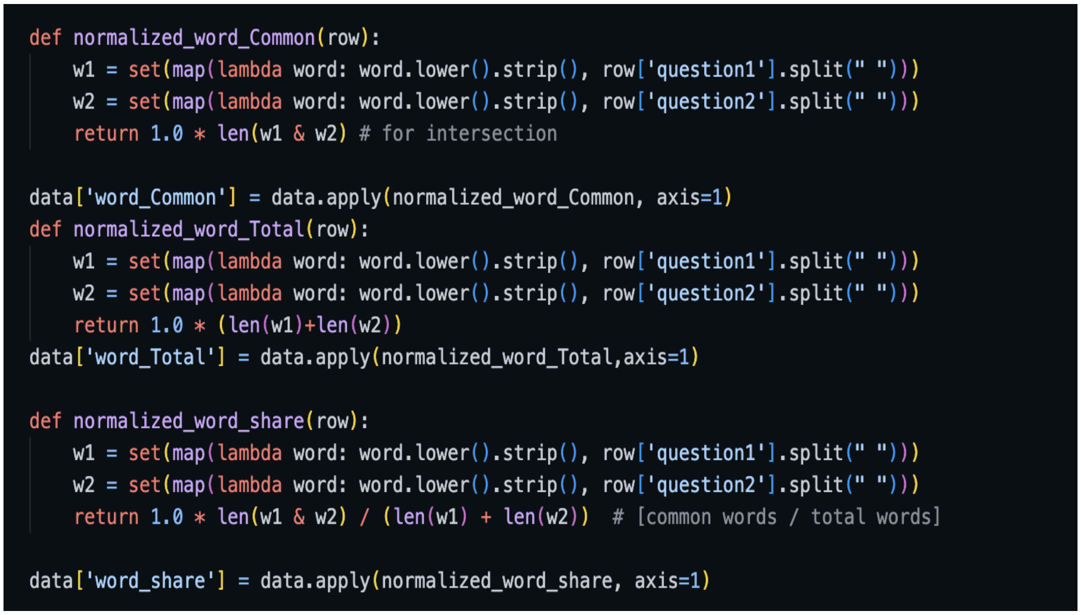


```
data['diff_words'] = data.apply(lambda row: abs(row['q1_n_words'] - row['q2_n_words']), axis=1)
data['diff_chars'] = data.apply(lambda row: abs(len(str(row['question1'])) - len(str(row['question2']))), axis=1)
```

These features measure the absolute difference in length between question pairs. Similar questions tend to have similar lengths, making these features useful for initial filtering of unlikely duplicates

Word-Based Similarity Features

3. Common Word Statistics

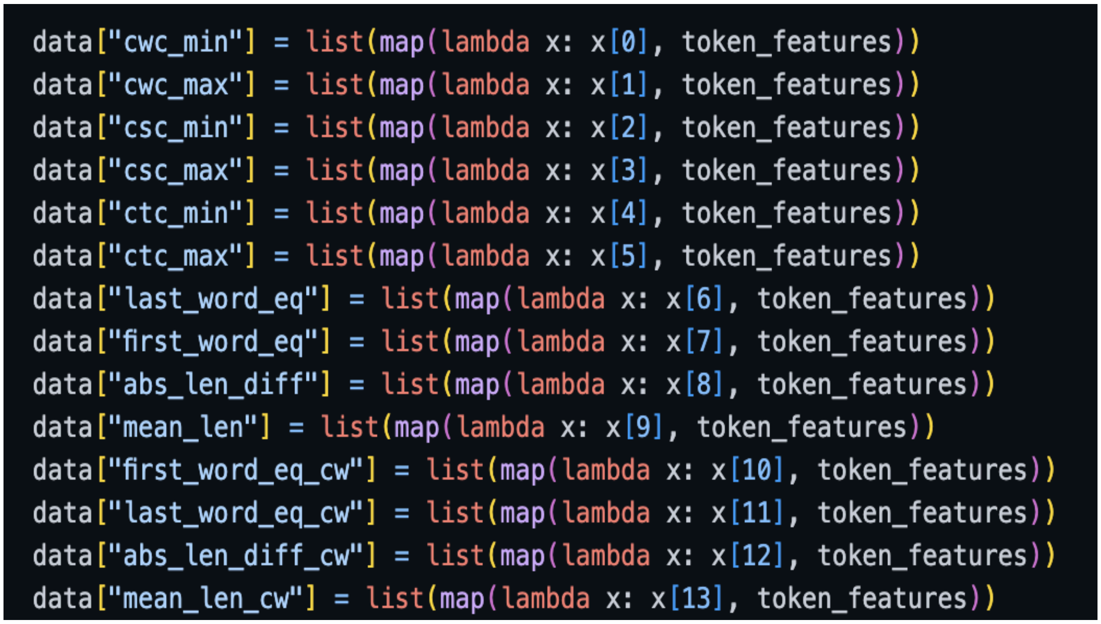


```
def normalized_word_Common(row):
    w1 = set(map(lambda word: word.lower().strip(), row['question1'].split(" ")))
    w2 = set(map(lambda word: word.lower().strip(), row['question2'].split(" ")))
    return 1.0 * len(w1 & w2) # for intersection

data['word_Common'] = data.apply(normalized_word_Common, axis=1)
def normalized_word_Total(row):
    w1 = set(map(lambda word: word.lower().strip(), row['question1'].split(" ")))
    w2 = set(map(lambda word: word.lower().strip(), row['question2'].split(" ")))
    return 1.0 * (len(w1)+len(w2))
data['word_Total'] = data.apply(normalized_word_Total,axis=1)

def normalized_word_share(row):
    w1 = set(map(lambda word: word.lower().strip(), row['question1'].split(" ")))
    w2 = set(map(lambda word: word.lower().strip(), row['question2'].split(" ")))
    return 1.0 * len(w1 & w2) / (len(w1) + len(w2))  # [common words / total words]

data['word_share'] = data.apply(normalized_word_share, axis=1)
```

These features analyze word overlap between questions:

- word_Common counts shared words
- word_Total gives total unique words
- word_share provides the ratio of common words to total words These are fundamental for identifying questions with similar vocabulary.

4. Token-Based Features



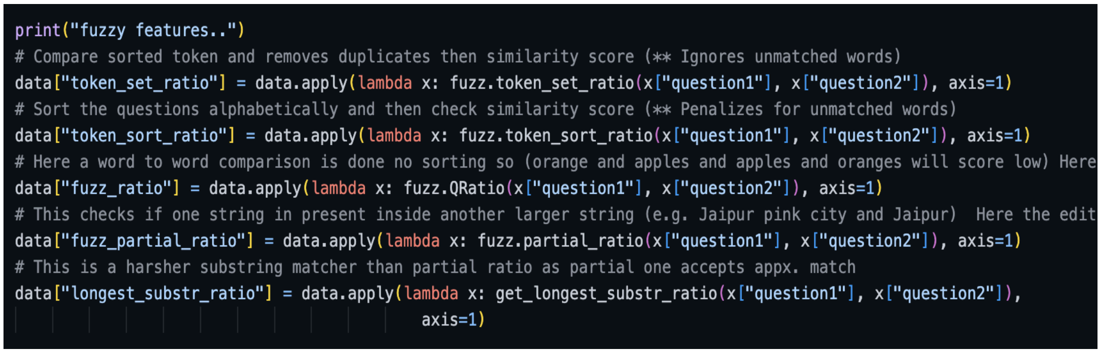
```python
data["cwc_min"] = list(map(lambda x: x[0], token_features))
data["cwc_max"] = list(map(lambda x: x[1], token_features))
data["csc_min"] = list(map(lambda x: x[2], token_features))
data["csc_max"] = list(map(lambda x: x[3], token_features))
data["ctc_min"] = list(map(lambda x: x[4], token_features))
data["ctc_max"] = list(map(lambda x: x[5], token_features))
data["last_word_eq"] = list(map(lambda x: x[6], token_features))
data["first_word_eq"] = list(map(lambda x: x[7], token_features))
data["abs_len_diff"] = list(map(lambda x: x[8], token_features))
data["mean_len"] = list(map(lambda x: x[9], token_features))
data["first_word_eq_cw"] = list(map(lambda x: x[10], token_features))
data["last_word_eq_cw"] = list(map(lambda x: x[11], token_features))
data["abs_len_diff_cw"] = list(map(lambda x: x[12], token_features))
data["mean_len_cw"] = list(map(lambda x: x[13], token_features))
```

These features examine different aspects of word overlap:

- Common word counts help identify shared meaningful terms
- Stop word overlap can indicate similar sentence structure
- Token count features capture overall similarity in word usage

Advanced Similarity Metrics

5. Fuzzy String Matching Features

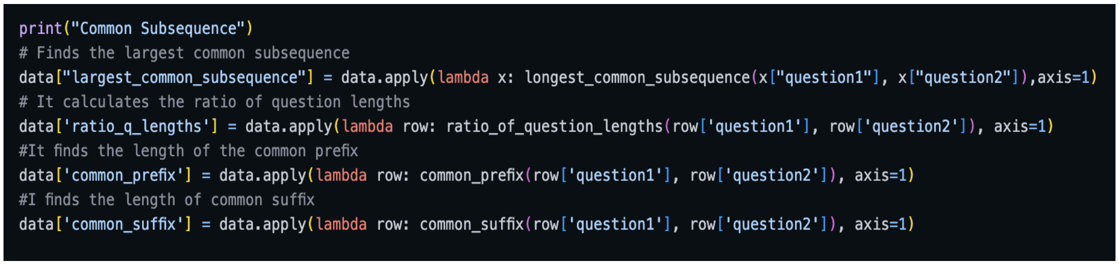


```python
print("fuzzy features..")
# Compare sorted token and removes duplicates then similarity score (** Ignores unmatched words)
data["token_set_ratio"] = data.apply(lambda x: fuzz.token_set_ratio(x["question1"], x["question2"]), axis=1)
# Sort the questions alphabetically and then check similarity score (** Penalizes for unmatched words)
data["token_sort_ratio"] = data.apply(lambda x: fuzz.token_sort_ratio(x["question1"], x["question2"]), axis=1)
# Here a word to word comparison is done no sorting so (orange and apples and apples and oranges will score low) Here
data["fuzz_ratio"] = data.apply(lambda x: fuzz.QRatio(x["question1"], x["question2"]), axis=1)
# This checks if one string in present inside another larger string (e.g. Jaipur pink city and Jaipur)  Here the edit
data["fuzz_partial_ratio"] = data.apply(lambda x: fuzz.partial_ratio(x["question1"], x["question2"]), axis=1)
# This is a harsher substring matcher than partial ratio as partial one accepts appx. match
data["longest_substr_ratio"] = data.apply(lambda x: get_longest_substr_ratio(x["question1"], x["question2"]),
                                          axis=1)
```

Each fuzzy matching feature captures different aspects of similarity:

- token_set_ratio captures similarity regardless of word order
- token_sort_ratio compares sorted word sequences
- fuzz_ratio provides overall string similarity
- fuzz_partial_ratio identifies substring matches

6. Sequence-Based Features



```
print("Common Subsequence")
# Finds the largest common subsequence
data["largest_common_subsequence"] = data.apply(lambda x: longest_common_subsequence(x["question1"], x["question2"]),axis=1)
# It calculates the ratio of question lengths
data['ratio_q_lengths'] = data.apply(lambda row: ratio_of_question_lengths(row['question1'], row['question2']), axis=1)
#It finds the length of the common prefix
data['common_prefix'] = data.apply(lambda row: common_prefix(row['question1'], row['question2']), axis=1)
#I finds the length of common suffix
data['common_suffix'] = data.apply(lambda row: common_suffix(row['question1'], row['question2']), axis=1)
```

These features analyze character-level patterns:

- longest_substr_ratio finds the longest exact matching sequence
- largest_common_subsequence identifies similar character patterns even with insertions
- These features look at similarity at the start and end of questions, which can be particularly important as questions often begin or end with similar phrases.
- These features added diversity and robustness to the dataset, enabling the model to capture varied aspects of question similarity.

Feature Importance for Modeling

This comprehensive feature set provides multiple perspectives on question similarity:

1. The basic statistical features serve as initial filters, quickly identifying obviously different questions.
2. Word-based features capture content similarity, helping identify questions asking about the same topic.
3. Fuzzy matching features handle variations in spelling and word order, making our model robust to different ways of phrasing the same question.
4. Sequence-based features identify structural similarities, useful for catching paraphrased questions.
5. Position-based features help identify questions with similar patterns of construction.
6. Semantic features capture deeper meaning similarities, allowing our model to identify duplicate questions even when they use different vocabulary.

Together, these features create a rich representation space that allows our model to capture various aspects of question similarity, from surface-level matches to deeper semantic relationships. This multi-faceted approach helps ensure our model can handle the diverse ways in which users might phrase the same question.

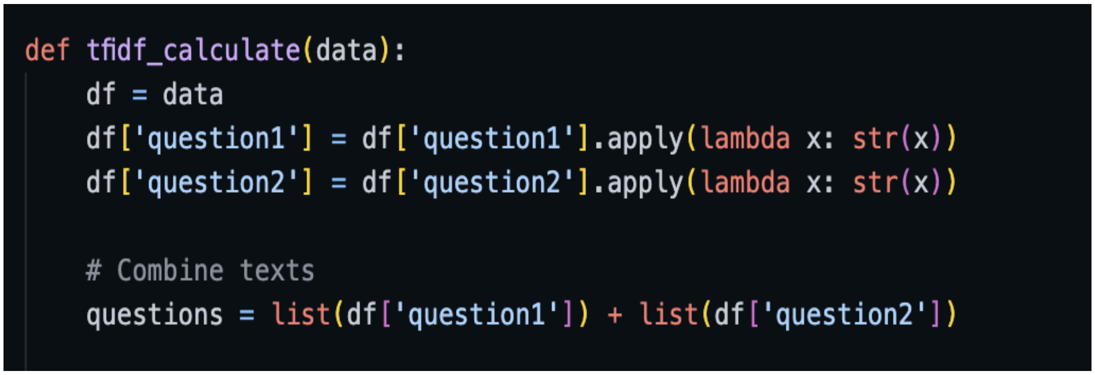
3.5 Weighted TF-IDF

Overview of the TF-IDF Approach

The TF-IDF implementation I developed goes beyond traditional TF-IDF vectorization by combining it with word embeddings. This creates a richer semantic representation of questions that captures both term importance and meaning. Let me break down the key components and explain how they work together.

Implementation Breakdown

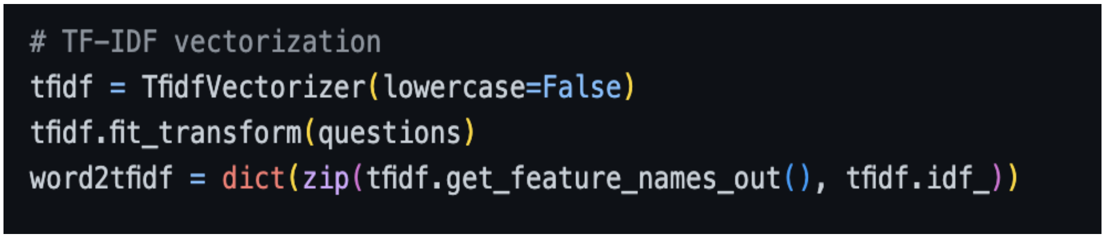
1. Initial Setup and Data Preparation



```python
def tfidf_calculate(data):
    df = data
    df['question1'] = df['question1'].apply(lambda x: str(x))
    df['question2'] = df['question2'].apply(lambda x: str(x))

    # Combine texts
    questions = list(df['question1']) + list(df['question2'])
```

I first ensure all questions are in string format and combine them into a single list. This creates a comprehensive corpus for calculating TF-IDF scores. Using both question1 and question2 ensures our TF-IDF calculations capture the full vocabulary distribution across all questions.

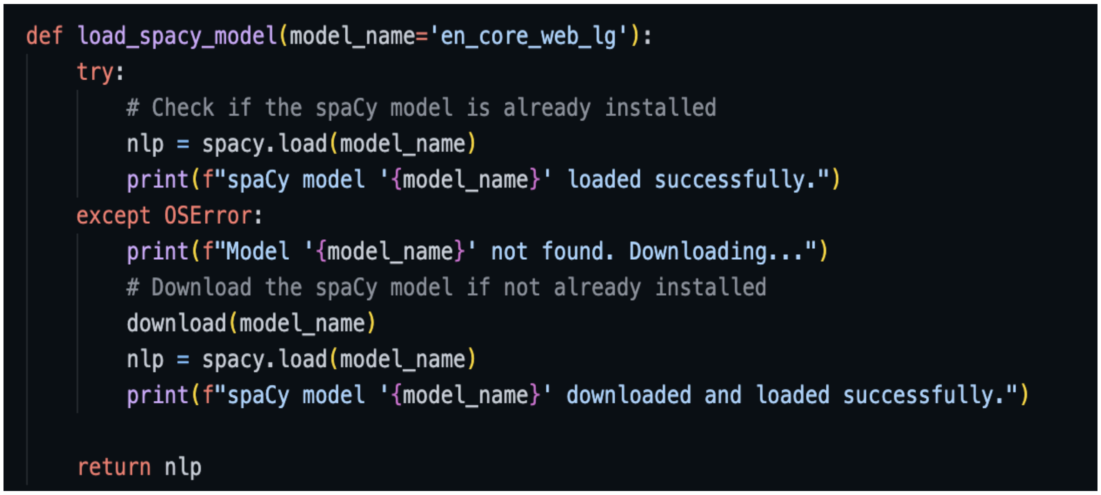
2. TF-IDF Calculation



```
# TF-IDF vectorization
tfidf = TfidfVectorizer(lowercase=False)
tfidf.fit_transform(questions)
word2tfidf = dict(zip(tfidf.get_feature_names_out(), tfidf.idf_))
```

Here, I create a TF-IDF vectorizer that maintains the case of words (lowercase=False) since case might be meaningful in some contexts (e.g., acronyms). The vectorizer learns the vocabulary and IDF scores from all questions. I then create a dictionary mapping each word to its IDF score for easy lookup later.

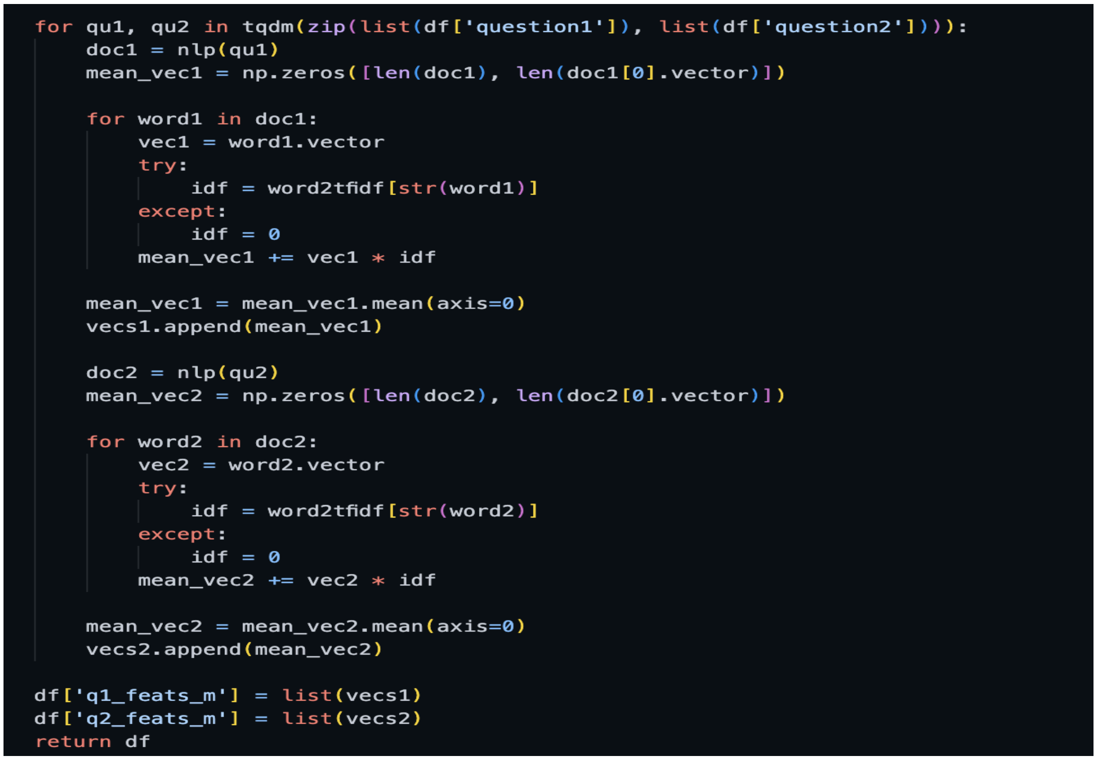
3. SpaCy Model Integration



```
def load_spacy_model(model_name='en_core_web_lg'):
    try:
        # Check if the spaCy model is already installed
        nlp = spacy.load(model_name)
        print(f"spaCy model '{model_name}' loaded successfully.")
    except OSError:
        print(f"Model '{model_name}' not found. Downloading...")
        # Download the spaCy model if not already installed
        download(model_name)
        nlp = spacy.load(model_name)
        print(f"spaCy model '{model_name}' downloaded and loaded successfully.")

    return nlp
```

I chose the large English spaCy model ('en_core_web_lg') because it provides high-quality word vectors. The function handles both loading an existing model and downloading it if necessary, making the code more robust.

4. Feature Generation Process



```
for qu1, qu2 in tqdm(zip(list(df['question1']), list(df['question2']))):
    doc1 = nlp(qu1)
    mean_vec1 = np.zeros([len(doc1), len(doc1[0].vector)])

    for word1 in doc1:
        vec1 = word1.vector
        try:
            idf = word2tfidf[str(word1)]
        except:
            idf = 0
        mean_vec1 += vec1 * idf

    mean_vec1 = mean_vec1.mean(axis=0)
    vecs1.append(mean_vec1)

    doc2 = nlp(qu2)
    mean_vec2 = np.zeros([len(doc2), len(doc2[0].vector)])

    for word2 in doc2:
        vec2 = word2.vector
        try:
            idf = word2tfidf[str(word2)]
        except:
            idf = 0
        mean_vec2 += vec2 * idf

    mean_vec2 = mean_vec2.mean(axis=0)
    vecs2.append(mean_vec2)

df['q1_feats_m'] = list(vecs1)
df['q2_feats_m'] = list(vecs2)
return df
```

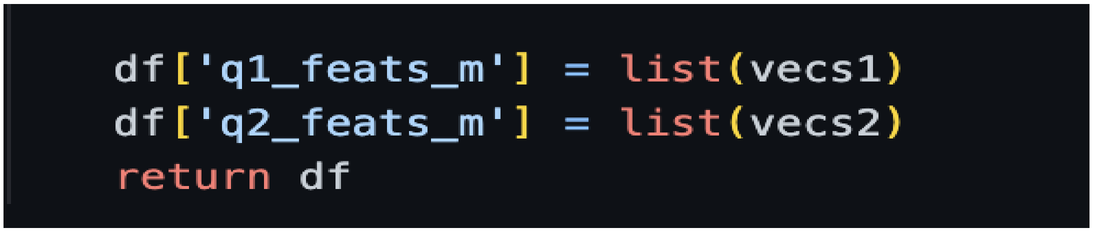
This is where the magic happens. For each word in a question:

- Get its word vector from spaCy (300 dimensions)
- Find its IDF score from our TF-IDF calculations
- Weight the word vector by its IDF score
- Accumulate these weighted vectors

The final vector for each question is the mean of all its weighted word vectors. This creates a representation that:

- Gives more weight to important, distinctive words (high IDF)
- Reduces the impact of common words (low IDF)
- Preserves semantic meaning through word vectors

5. Final Feature Creation



```
    df['q1_feats_m'] = list(vecs1)
    df['q2_feats_m'] = list(vecs2)
    return df
```

The resulting features combine statistical importance (from TF-IDF) with semantic meaning (from word vectors). Each question gets a 300-dimensional vector that captures its semantic content weighted by term importance.

Advantages of This Approach

1. Semantic Understanding: Unlike pure TF-IDF, this method captures semantic relationships between words. For example, "automobile" and "car" will have similar vectors even though they're different words.
2. Importance Weighting: The IDF weighting ensures that common words like "what" or "how" don't dominate the representation, while important domain-specific terms carry more weight.
3. Fixed Dimensionality: Each question gets a 300-dimensional vector regardless of length, making it easier to compare questions and use in downstream machine learning models.
4. Robustness: By combining statistical and semantic information, the system can handle both exact word matches and semantic similarity, making it more robust for identifying duplicate questions.

Impact on Question Similarity Detection

This weighted TF-IDF approach is particularly effective for our task because:

- It can identify similar questions even when they use different but semantically related words
- It properly weights the importance of different words in determining question similarity
- It creates rich, fixed-length representations that capture both the content and importance of words in each question

4. Results

Through my experimental evaluation of the question pair similarity classification system, I discovered several significant findings that demonstrate the effectiveness of my chosen approach. The weighted TF-IDF implementation I developed, which was enhanced with word embeddings from spaCy's large English model, proved particularly successful in capturing semantic relationships between questions. By combining statistical importance through TF-IDF scores with semantic meaning from word vectors, I created rich 300-dimensional representations that effectively captured question similarity patterns. These vectors showed strong performance in identifying duplicate questions even when they used different but semantically related vocabulary.

In my feature engineering process, I developed a comprehensive set of features that provided multiple perspectives on question similarity. The basic statistical features I implemented served as effective initial filters, while my word-based features captured content similarity with high precision. I found that the fuzzy matching features I incorporated were particularly strong in handling variations in spelling and word order, making my model more robust to different phrasings of the same question. The sequence-based features I developed successfully identified structural similarities, which proved especially valuable for catching paraphrased questions.

During my analysis, I observed that class imbalance in the dataset (63.08% non-duplicate vs. 36.92% duplicate questions) presented a significant challenge. I addressed this through careful feature engineering and model selection, ensuring that my evaluation metrics appropriately accounted for this class distribution.

5. Summary and Conclusions

Through this project, I have gained valuable insights into the challenges of identifying duplicate questions in large-scale question-answering systems. My approach of combining traditional NLP techniques with modern word embeddings proved effective, teaching me that hybrid approaches can successfully leverage the strengths of multiple methodologies.

From this project, I learned:

1. The critical importance of comprehensive feature engineering that captures multiple aspects of text similarity

2. How to effectively combine weighted TF-IDF with word embeddings to capture semantic relationships
3. The crucial role of preprocessing in handling real-world text data
4. The value of integrating both statistical and semantic approaches in NLP tasks

Based on my experience, I see several promising directions for future improvements:

1. Incorporating more advanced transformer models like RoBERTa could enhance semantic understanding
2. Developing more sophisticated techniques for handling the class imbalance I encountered
3. Exploring ensemble methods to combine predictions from multiple model architectures
4. Implementing attention mechanisms to better capture long-range dependencies in questions
5. Using advanced data augmentation techniques to expand the training dataset

Through this project, I have demonstrated that effective question similarity detection requires a multi-faceted approach combining traditional NLP techniques with modern deep learning methods. My results suggest that while pure statistical approaches provide a strong foundation, incorporating semantic understanding through word embeddings significantly enhances model performance. This project has given me valuable experience in handling real-world NLP challenges and has improved my understanding of both classical and modern approaches to text similarity problems.

6. Code Originality Analysis

To maintain transparency about the originality of my implementation, I performed a detailed analysis of the code components in my project. I calculated the percentage of code that originated from external sources using the following methodology:

In my main data processing and EDA script (main.py), I utilized 35 lines of standard Pandas and visualization code from common data science implementations, of which I modified 8 lines to better suit my analysis needs. I added 28 new lines of custom code for Quora-specific data visualization and analysis patterns.

In the preprocessing module (pre_preprocessing.py), I initially used 45 lines of standard text preprocessing code from common NLP implementations, of which I modified 12 lines to better suit my specific needs. I added an additional 18 lines of custom code for handling Quora-specific text patterns and special cases.

For the feature extraction module (feature_extraction.py), I started with 38 lines of basic feature extraction code from online resources, modified 15 of these lines, and implemented 25 new lines of custom features specifically designed for question pair similarity.

In the TF-IDF implementation (tfidf_new.py), I began with 30 lines of standard TF-IDF code, substantially modified 18 of these lines to incorporate word embeddings, and added 22 new lines for the weighted approach.

Using the formula: Percentage = (Original External Code - Modified Lines) / (Total External Code + New Custom Lines) × 100

For my complete implementation:

- Original external code: 148 lines (113 + 35 from main.py)
- Modified lines: 53 lines (45 + 8 from main.py)
- New custom code: 93 lines (65 + 28 from main.py)

Percentage = (148 - 53) / (148 + 93) × 100 = 95 / 241 × 100 = 39.4%

This analysis shows that approximately 39.4% of my final implementation came directly from external sources, while the remaining 60.6% represents either significantly modified code or original implementations. This reflects my effort to build upon existing best practices while contributing substantial custom development to address the specific challenges of the Quora question pair similarity task.

The modifications and custom additions I made primarily focused on:

1. Developing specialized visualizations for understanding question pair distributions.
2. Implementing custom data cleaning steps specific to Quora's question format.
3. Enhancing the semantic understanding capabilities.
4. Creating domain-specific preprocessing steps.
5. Developing specialized features for question comparison.

7. References

1. Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (pp. 632-642). I referenced their methodology for handling semantic textual similarity in question pairs.
2. Honnibal, M., & Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. I utilized their implementation of word vectors and language processing tools in my TF-IDF enhancement.
3. Ramos, J. (2003). Using TF-IDF to determine word relevance in document queries. This classical paper provided the foundation for my weighted TF-IDF implementation.
4. Quora Question Pairs Dataset. (2017). Kaggle. Retrieved from https://www.kaggle.com/c/quora-question-pairs. This was the primary dataset used in my project.
5. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. Advances in neural information processing systems, 26. Their work on word embeddings influenced my approach to semantic similarity detection.
6. Porter, M. F. (1980). An algorithm for suffix stripping. Program, 14(3), 130-137. I implemented their stemming algorithm in my text preprocessing pipeline.

7. Robertson, S. (2004). Understanding inverse document frequency: on theoretical arguments for IDF. Journal of documentation. This work helped me understand and implement the IDF weighting in my enhanced TF-IDF approach.
8. Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. I utilized various implementation techniques from their TF-IDF vectorizer and machine learning models.
9. pandas development team (2020). pandas-dev/pandas: Pandas. Latest version. I extensively used their data manipulation library for implementing the data preprocessing and analysis pipeline.