THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

QUORA QUESTION PAIR SIMILARITY

Natural Language Processing Final Project

Individual Report

Name: Gourab Mukherjee

Group: 5

Date: 12/11/2024

TABLE OF CONTENTS

Introduction

Community-based Q&A platforms like Quora, with over 100 million monthly visitors, provide a space for users to seek knowledge and share insights on various topics. However, the platform's popularity has led to a significant challenge: an influx of redundant questions with similar meanings. This redundancy makes it harder for users to find relevant answers quickly and burdens contributors who often address multiple variations of the same query.

This project aims to tackle this issue by developing a question similarity classifier using the Quora Question Pairs dataset. The classifier determines whether two questions are contextually similar, helping to reduce duplication and enhance the efficiency of the knowledge-sharing process. The approach involves implementing classical machine learning models, neural networks, and state-of-the-art transformer-based architectures.

Through comprehensive feature engineering, semantic relationships between questions were captured using fuzzy features, subsequences, and token-based attributes. Classical machine learning models, such as Random Forest and XGBoost, showed promising results, with XGBoost achieving an F1 micro score of 85.38%. Deep learning models, including LSTMs and GRUs, further improved classification, achieving an F1 macro score of 84.45%, demonstrating the potential of sequence-based learning. By fine-tuning BERT with advanced techniques like contrastive loss, the project achieved the highest validation F1 macro score of 94.38%, effectively capturing deeper contextual relationships between question pairs. These results showcase the effectiveness of combining classical methods with modern transformer-based architectures in solving complex NLP problems like question similarity classification.

Shared Workflow:

1. Dataset Loading and Exploratory Data Analysis (EDA)

- Loading the Dataset: The Quora question pairs dataset, comprising approximately 404,000 rows, was loaded. Each row contains two questions and a label indicating whether they are duplicates.

EDA: Performed exploratory data analysis to gain insights into the data distribution, uncover patterns, and detect anomalies. This included visualizations and statistical summaries.

2. Data Preprocessing:

- Preprocessing Module: The `pre_preprocessing.py` module was employed for data preprocessing, incorporating the following techniques:

- Data Normalization: Converted text to lowercase, expanded contractions, and standardized symbols (e.g., currency, percentages).

- Data Cleaning: Removed non-word characters and HTML tags, and formatted large numbers into a more readable form.

- Stemming: Utilized the PorterStemmer to reduce words to their root forms.

- Output: The dataset was cleaned and standardized, making it suitable for feature extraction.

3. Feature Extraction

- Feature Extraction Module: Leveraged the `feature_extraction.py` module to extract meaningful features from the questions. This included:

- Tokenization: Tokenized questions using a word tokenizer.

- Feature Categories: Extracted 40 features grouped into the following categories:

- Fuzzy Features: Metrics like `fuzz_ratio` and `fuzz_partial_ratio` to evaluate word-level fuzzy similarity.

- Token Features: Analyzed stopwords and non-stopwords, such as common non-stopwords, common stopwords, and word size differences.

- Common Subsequence Features: Assessed sentence similarity using metrics like `largest_common_subsequence` and `jaccard_similarity`.

4. Vectorization

- TF-IDF Module: The `tfidf_new.py` module was utilized for vectorizing the preprocessed questions:

- Weighted TF-IDF: Computed weighted TF-IDF scores and integrated them with spaCy-generated word embeddings for enriched feature representation.

- Output: Produced feature vectors based on 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 TF-IDF feature vectors.

- BERT Embeddings: Generated embeddings using BERT and used them to train classical models, enabling performance comparison.

6. Final Model and Evaluation

- BERT Model: Fine-tuned a BERT model specifically for the question pair classification task and compared its performance against classical models.

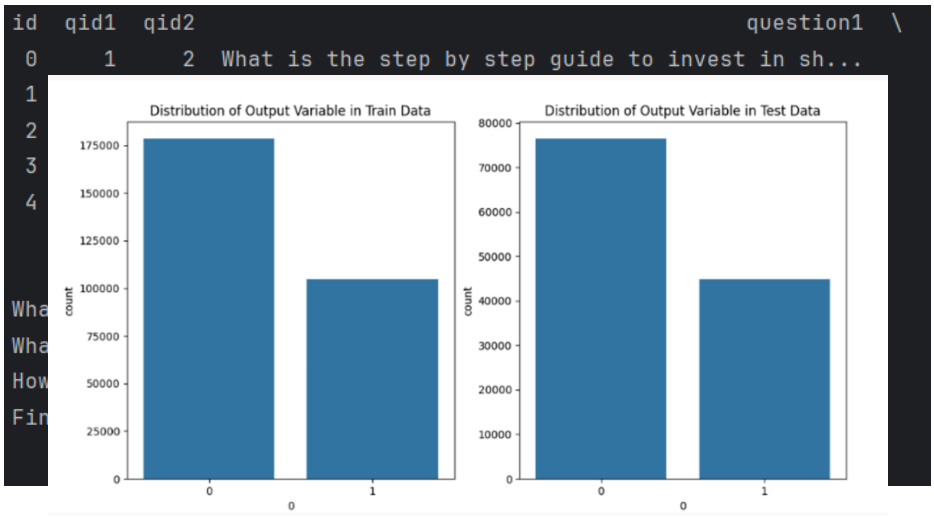
- Evaluation Metrics: Employed accuracy, F1-score, precision, and recall to assess and compare the performance of all models.

Dataset

The dataset used in this project is the Quora Question Pairs dataset, originally introduced on Kaggle as part of a competition addressing the same challenge. It contains 404,290 question pairs and includes six features: `id`, `qid1`, `qid2`, `question1`, `question2`, and `is_duplicate`.

Table 1: A snapshot of the Quora Question Pairs Dataset

The dataset is imbalanced, with approximately 36.92% of the pairs classified as duplicate questions, while the remaining 63.08% are categorized as non-



duplicates.

Figure 1: A bar plot showcasing the imbalance in the data

Description of Individual Work

3.1 Background and overview

As part of this project, I generated BERT-based features alongside previously created 40 engineered features to address the Quora Question Pair Similarity problem. I applied classical machine learning models like Naïve Bayes, Logistic Regression, Random Forest, and XGBoost, leveraging BERT embeddings to establish strong baselines. Along with advanced deep learning techniques like

LSTM and GRU, all these approaches provided a computationally efficient way to capture basic patterns and semantic relationships within the data.

For deeper and more nuanced understanding, I implemented a transformer-based deep learning model using SBERT within a Siamese network framework. This approach was particularly well-suited for the dataset, as it allowed us to fine-tune pre-trained BERT embeddings selectively, ensuring that the model captured the intricate semantic relationships between question pairs while avoiding overfitting. The addition of two fully connected layers further enhanced the model's ability to distinguish between duplicate and non-duplicate pairs by learning task-specific representations. This combination of classical and deep learning methods ensured a balanced and effective strategy for addressing the problem's challenges.

3.2 Features generation

Before applying any form of modelling I began preprocessing the questions, removing stopwords to retain only significant terms. Then I computed across multiple dimensions: lexical, structural, and semantic. Lexical features, such as common word counts and Jaccard similarity, measured overlaps in vocabulary, highlighting shared meaning between the questions. Structural features, like length ratios, prefix and suffix overlaps, and differences in token counts, captured variations in the form and phrasing. For semantic features I leveraged fuzzy string matching, longest common subsequence, and substring ratios to detect reworded or paraphrased questions. To further enhance the feature set I used Advanced metrics, such as normalized word shares and token intersections. These features, combined with statistical properties like question frequency and edit distances, created a comprehensive representation of similarity.

Lexical Features: These features captured overlaps and differences in the vocabulary of the two questions:

common_word_len: Counts the number of common non-stopwords between the two questions. Meaning A higher overlap of meaningful words suggests a stronger semantic similarity.

common_stop_word_len: Counts the number of shared stopwords.Although stopwords are less meaningful individually, shared stopwords can indicate similar sentence structures.

common_token: Measures the total number of overlapping tokens (including both stopwords and non-stopwords).It Helps capture the overall similarity of the vocabulary, irrespective of token type.

word_Common: It gives the normalized count of shared words between the two questions.It quantifies the degree of overlap in unique words, which is key for detecting rephrased questions.

word_share: Helps in understanding the proportion of shared words to the total unique words in both questions and Highlights the balance of overlap versus distinctiveness, useful for distinguishing duplicates.

Structural Features: These features helped us in analyzing the lengths and arrangement of the questions to detect structural similarities or differences.

q1len and q2len: Measured the lengths of the two questions in characters.As lengths provide us a basic structural insight—drastically different lengths often indicate unrelated questions.

abs_len_diff: The absolute difference between the lengths of the questions. Quantifies how balanced the questions are in length. A large difference could indicate dissimilarity.

mean_len: This is the average length of the two questions which helps to normalize the analysis, especially for longer or shorter questions.

ratio_of_question_lengths: It is the ratio of the lengths of the two questions.

Which detects proportional relationships, e.g., whether one question is an elaboration of the other.

diff_words: The absolute difference in the number of words between the two questions which highlights variations in word counts and can hint at either paraphrasing or unrelated content.

common_prefix: Length of the matching prefix at the start of both questions which indicates whether questions begin with the same framing or topic.

common_suffix: Length of the matching suffix at the end of both questions as it is useful for questions that end with clarifications or qualifiers.

Semantic Features: These features delve into deeper relationships between the two questions, using advanced similarity metrics.

fuzz_ratio: Measured the overall similarity of the strings using edit distance.It captured general similarity, penalizing for word order changes and mismatches.

token_set_ratio: Compared the unique tokens in both questions, ignoring word order and was useful for detecting similarity when word arrangement differs but content is the same.

token_sort_ratio: Compares sorted tokens, emphasizing the sequence and captures meaning in cases where word order contributes to context.

fuzz_partial_ratio: Measures substring similarity, even if one question is a subset of another.It helps identifies if one question being part of a longer or elaborated form of the other.

longest_substr_ratio: Ratio of the longest common substring to the shorter question.

largest_common_subsequence: Normalized length of the longest common subsequence between the two questions.

jaccard_similarity: The proportion of shared tokens to the total unique tokens across both questions. It helped us to understand a balance between overlap and distinctiveness, highlighting shared vocabulary.

Statistical Features: These features provide us additional insights into the context of the questions within the dataset:

freq_qid1 and freq_qid2: Frequency of qid1 and qid2 within the dataset.Helps us in understanding questions appearing frequently may be duplicates or highly similar.

freq_q1-q2 and freq_q1+q2: Differences and sums of the frequency counts of the two questions.It identifies the relative importance or dominance of one question over the other

Combined Features: These combine multiple perspectives to provide comprehensive insights

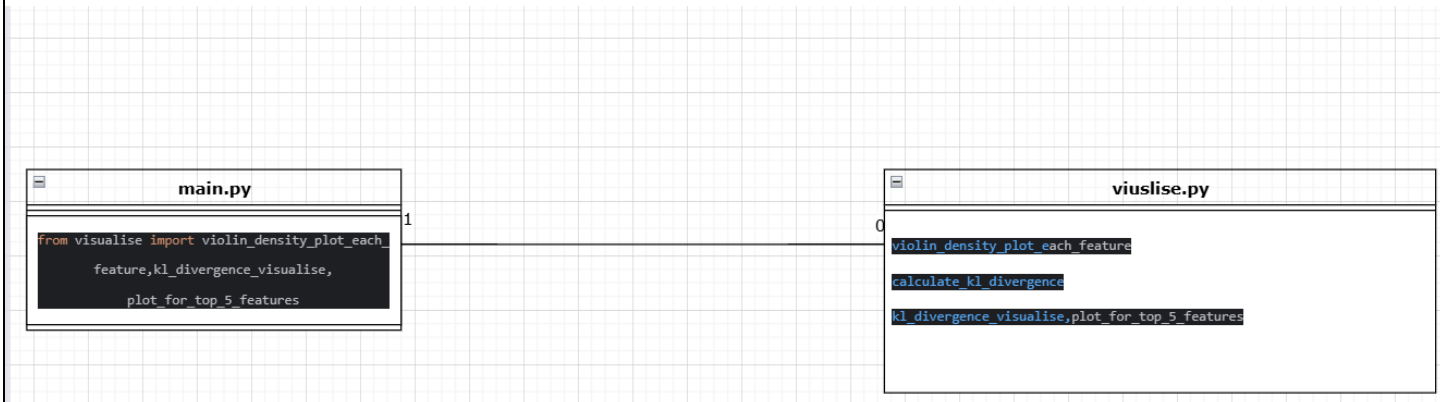
first_word_eq and last_word_eq: Whether the first and last tokens of both questions match.It Helps identify questions framed in similar ways, especially in structured queries.

first_word_eq_cw and last_word_eq_cw: The same as above but for non-stopwords.It focuses on semantically significant matches at the start and end of the questions.

mean_len_cw and abs_len_diff_cw: Mean and absolute differences in the length of non-stopword tokens.It Highlights structural differences while ignoring irrelevant words.

3.3 Visualisation of the Generated Features

The visualization of features using violin and density plots helped me understand the distributions and how they differentiate between the target classes (is_duplicate = 1 for duplicate questions and is_duplicate = 0 for non-duplicate questions). These plots allow me to intuitively assess the usefulness of each feature in distinguishing duplicate from non-duplicate pairs. This report provides an analysis of the violin and density plots for all features, explaining key observations and their implications for feature selection and machine learning model development. Features with minimal overlap between distributions (e.g., word_share, jaccard_similarity, and fuzz_partial_ratio) are highly discriminative and was prioritized in model training. This visualization process helped in improving the efficiency and accuracy of the final machine learning model.



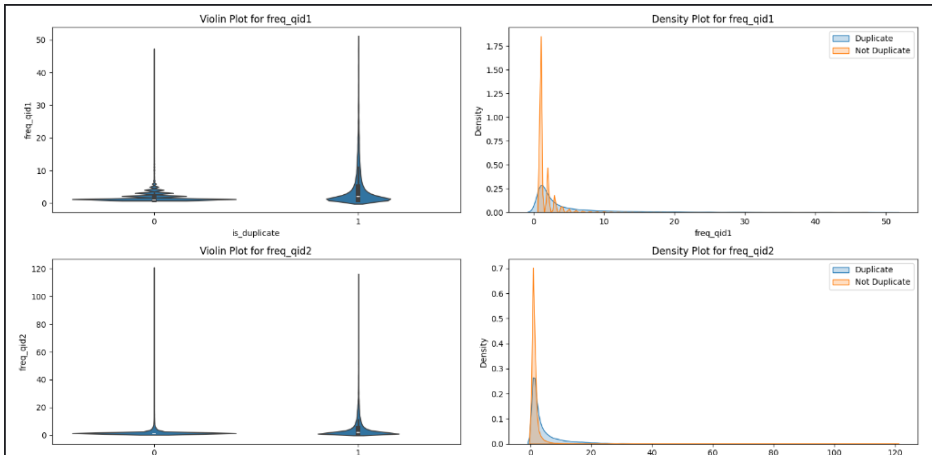

```python
# Creates one image with Violin plots and Density plots for each feature
def violin_density_plot_each_feature(data, features_to_plot):  2 usages  gourab1998muk *
    num_features = len(features_to_plot)
    plt.figure(figsize=(10, 4 * num_features))  # Dynamically set the figure height

    for i, feature in enumerate(features_to_plot):
        print("updated 4")
        # Violin plot
        plt.subplot( *args: num_features, 2, 2 * i + 1)
        sns.violinplot(x='is_duplicate', y=feature, data=data)
        plt.title(f'Violin Plot for {feature}')
        plt.xlabel('is_duplicate')
        plt.ylabel(feature)

        # Density plot
        plt.subplot( *args: num_features, 2, 2 * i + 2)
        sns.kdeplot(data[data['is_duplicate'] == 1][feature], label='Duplicate', fill=True, warn_singular=False)
        sns.kdeplot(data[data['is_duplicate'] == 0][feature], label='Not Duplicate', fill=True, warn_singular=False)
        plt.title(f'Density Plot for {feature}')
        plt.xlabel(feature)
        plt.ylabel('Density')
        plt.legend()

    plt.tight_layout()  # Adjust layout for all subplots
    plt.show()
```

```python
# Function to calculate KL Divergence
def calculate_kl_divergence(duplicate_data, non_duplicate_data, feature):  1 usage  gourab1998muk
    duplicate_dist = duplicate_data[feature].dropna()
    non_duplicate_dist = non_duplicate_data[feature].dropna()

    epsilon = 1e-10
    duplicate_dist += epsilon
    non_duplicate_dist += epsilon

    min_length = min(len(duplicate_dist), len(non_duplicate_dist))
    duplicate_dist = duplicate_dist.head(min_length)
    non_duplicate_dist = non_duplicate_dist.head(min_length)

    kl_divergence = entropy(duplicate_dist, non_duplicate_dist)
    return kl_divergence
```

```python
def kl_divergence_visualise(data,features_to_plot):  2 usages  gourab1998muk *
    kl_divergence_results = pd.DataFrame(columns=['Feature', 'KL_Divergence'])

    for feature in features_to_plot:
        kl_divergence = calculate_kl_divergence(data[data['is_duplicate'] == 1], data[data['is_duplicate'] == 0], feature)
        kl_divergence_results = pd.concat( objs: [kl_divergence_results, pd.DataFrame({
            'Feature': [feature],
            'KL_Divergence': [kl_divergence]
        })], ignore_index=True)

    # Display KL Divergence results in a table
    print(kl_divergence_results)

    # Create a bar plot to visualize inverted KL Divergence
    kl_divergence_results['Inverted_KL_Divergence'] = 1 / (kl_divergence_results['KL_Divergence'] + 1e-10)

    plt.figure(figsize=(15, 10))
    sns.barplot(x='Feature', y='Inverted_KL_Divergence', data=kl_divergence_results.sort_values(by='Inverted_KL_Divergence', ascending=False))
    plt.title('Inverted KL Divergence for Each Feature')
    plt.xticks(rotation=45, ha='right')
    plt.show()

    return kl_divergence_results
def plot_for_top_5_features(data,kl_divergence_results):  2 usages  gourab1998muk *
    bottom_5_features = kl_divergence_results.nsmallest(5, 'KL_Divergence')['Feature']

    print("The best 5 features are:")
    print(bottom_5_features)

    # Pair plot for the top 10 features
    n = data.shape[0]
    sns.pairplot(data[bottom_5_features.tolist() + ['is_duplicate']][0:n], hue='is_duplicate', vars=bottom_5_features.tolist())
    plt.show()
    return
```

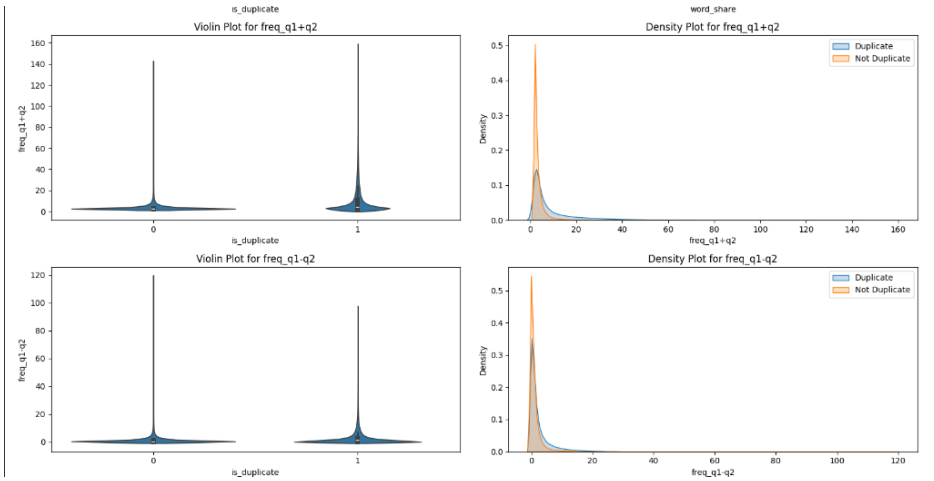
```python
import importlib
import visualise
importlib.reload(visualise)
#Fucntion to vilalaise violin and density plots for each feature in one image
from visualise import violin_density_plot_each_feature,kl_divergence_visualise,plot_for_top_5_features
#%%
features_to_plot = ['freq_qid1', 'freq_qid2', 'q1len', 'q2len', 'q1_n_words', 'q2_n_words',
                    'word_Common', 'word_Total', 'word_share', 'freq_q1+q2', 'freq_q1-q2',
                    'ratio_q_lengths', 'common_prefix', 'common_suffix', 'diff_words', 'diff_chars',
                    'jaccard_similarity', 'largest_common_subsequence', 'cwc_min', 'cwc_max', 'csc_min',
                    'csc_max', 'ctc_min', 'ctc_max', 'last_word_eq', 'first_word_eq', 'abs_len_diff',
                    'mean_len', 'token_set_ratio', 'token_sort_ratio', 'fuzz_ratio', 'fuzz_partial_ratio',
                    'longest_substr_ratio']

#%%
violin_density_plot_each_feature(data_features,features_to_plot)
#%%
data_features.columns

# %%
kl_divergence_results = kl_divergence_visualise(data_features,features_to_plot)
#%%
kl_divergence_results


# %%
plot_for_top_5_features(data_features,kl_divergence_results)
#%%
```

Feature: `freq_qid1` and `freq_qid2`



Observations: For duplicate questions (is_duplicate = 1), the distributions for both freq_qid1 and freq_qid2 peak at higher values compared to non-duplicates (is_duplicate = 0). Non-duplicates show a wider spread, indicating more variability in question frequency.

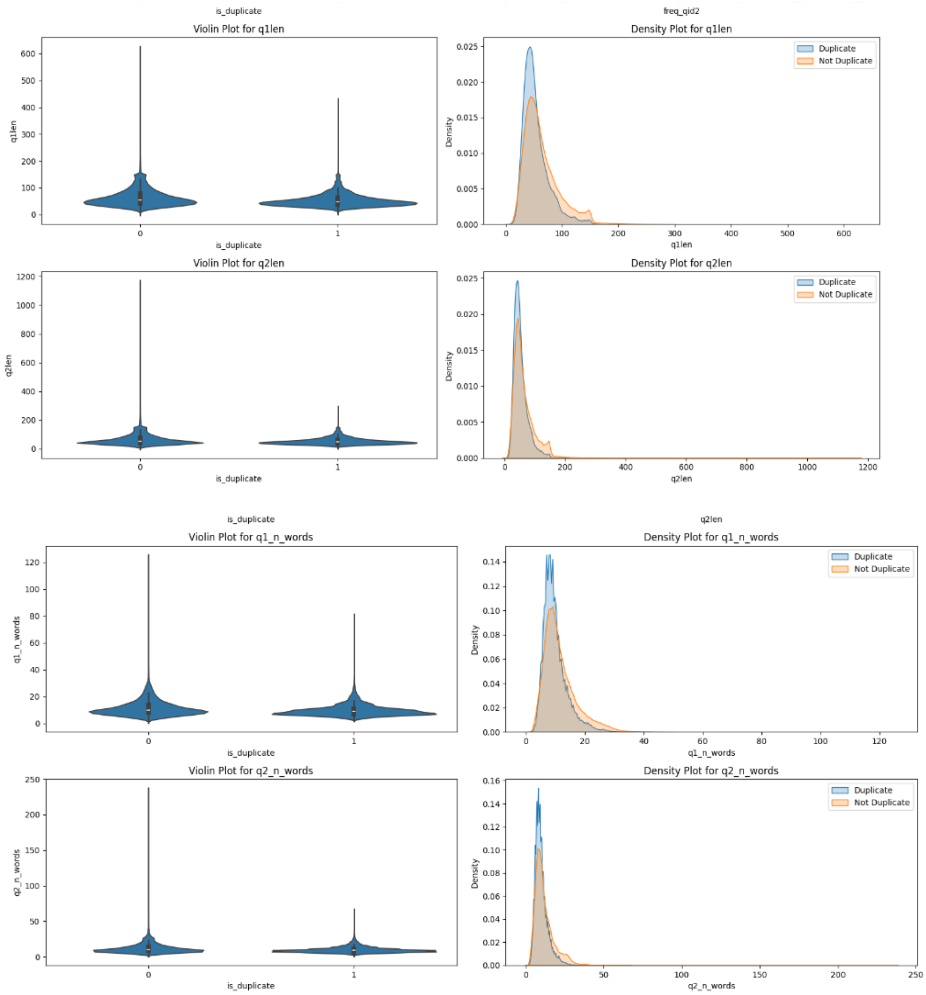
Feature: `freq_q1-q2` and `freq_q1+q2`



Observations: freq_q1-q2 shows a narrow spread for duplicate questions and a wider distribution for non-duplicates. freq_q1+q2 has a higher median for duplicates, with a sharper peak compared to non-duplicates. These features helped

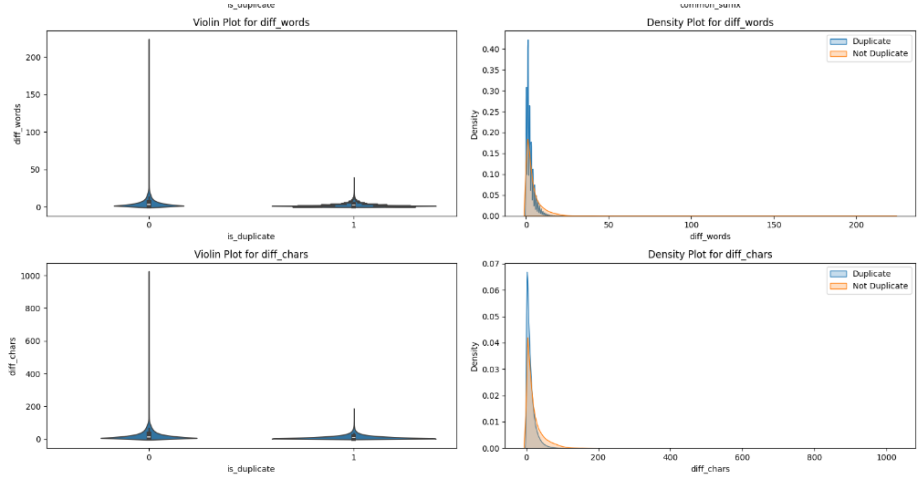
us quantify the difference and combined frequencies of question IDs in pairs. They indicated whether one question is significantly more common than the other, which is useful for identifying duplicates.

Feature: `q1len`, `q2len`, `q1_n_words`, `q2_n_words`



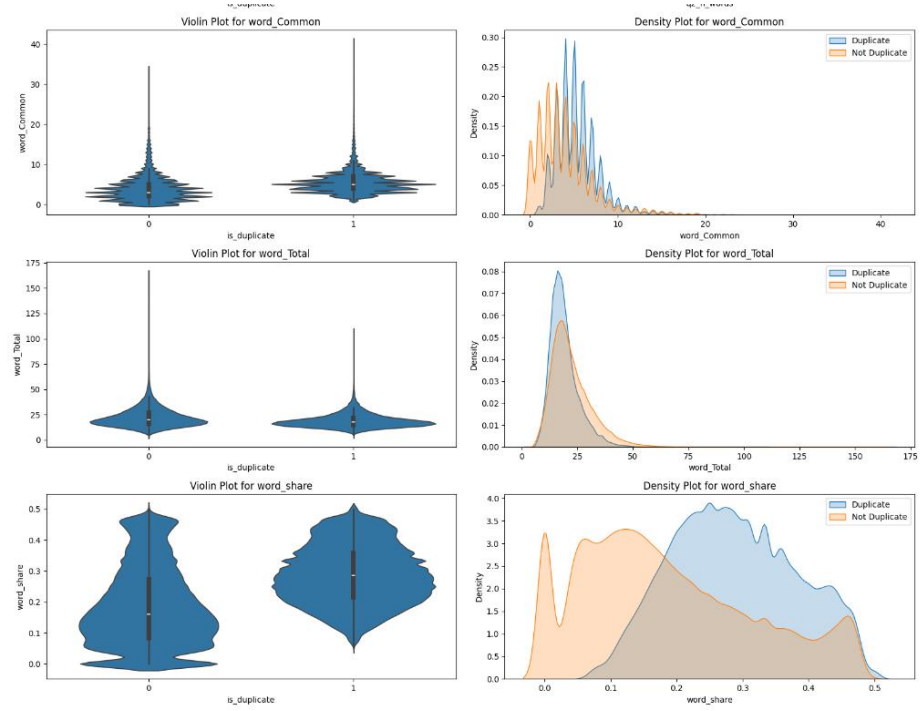
Observations: Duplicate questions (is_duplicate = 1) exhibit similar distributions for lengths (q1len, q2len) and word counts (q1_n_words, q2_n_words), with peaks around the same range. Non-duplicates show greater variability in lengths and word counts, from the wider spread in the violin plots.I found duplicate questions are often structurally similar, which reflects in their consistent lengths and token counts. This structural similarity is an important feature for classification

Feature: `diff_words` **and** `diff_chars`



Observations: Duplicate questions tend to have low values for both diff_words and diff_chars, indicating minimal differences in length and token counts. Non-duplicates show a more uniform distribution, with higher differences in many cases. Features like diff_words and diff_chars helped me in identifying non-duplicates, as unrelated questions tend to have larger differences in structure.

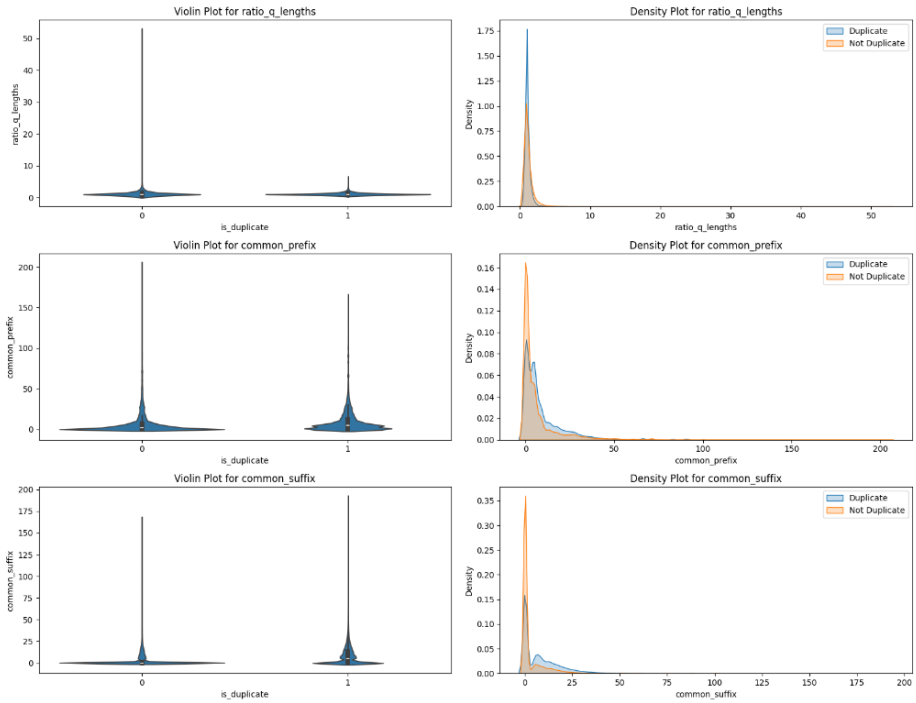
Feature: `word_Common`, `word_Total`, `word_share`



Observations: Duplicate questions have significantly higher word_Common and word_share values, indicating substantial overlap in vocabulary. Non-duplicates display a flatter distribution, with lower values for these features. These features are highly discriminative, as duplicate questions tend to share more words compared

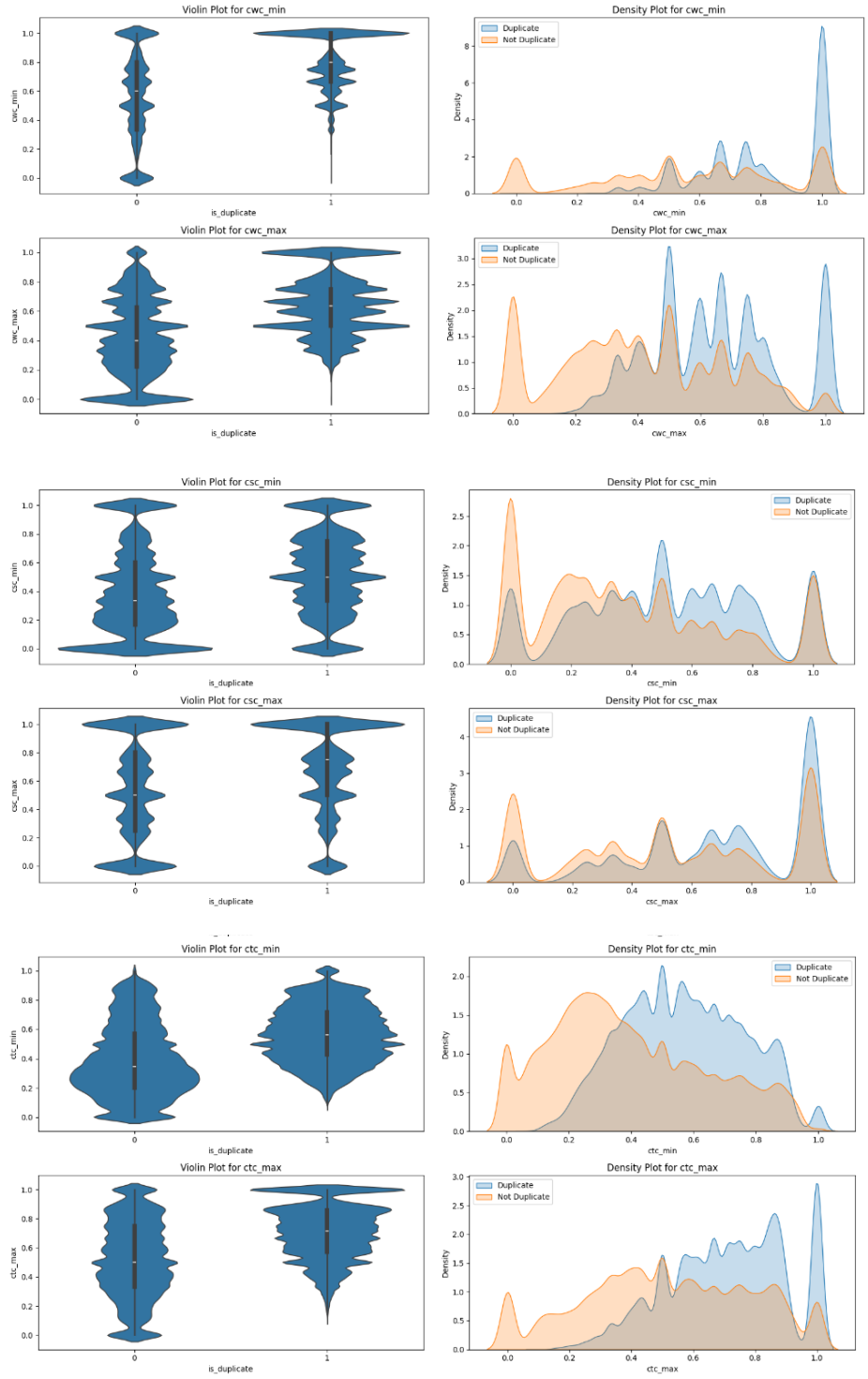
to non-duplicates. This was particularly useful for identifying reworded or paraphrased questions.

Feature: `ratio_q_lengths`, `common_prefix`, `common_suffix`



Observations: Duplicate questions cluster around a ratio_q_lengths value of 1, indicating similar lengths. common_prefix and common_suffix are higher for duplicates, showing that these questions often begin or end with the same words. Structural similarities such as length ratios and shared prefixes or suffixes helped in strong indication of duplicate questions

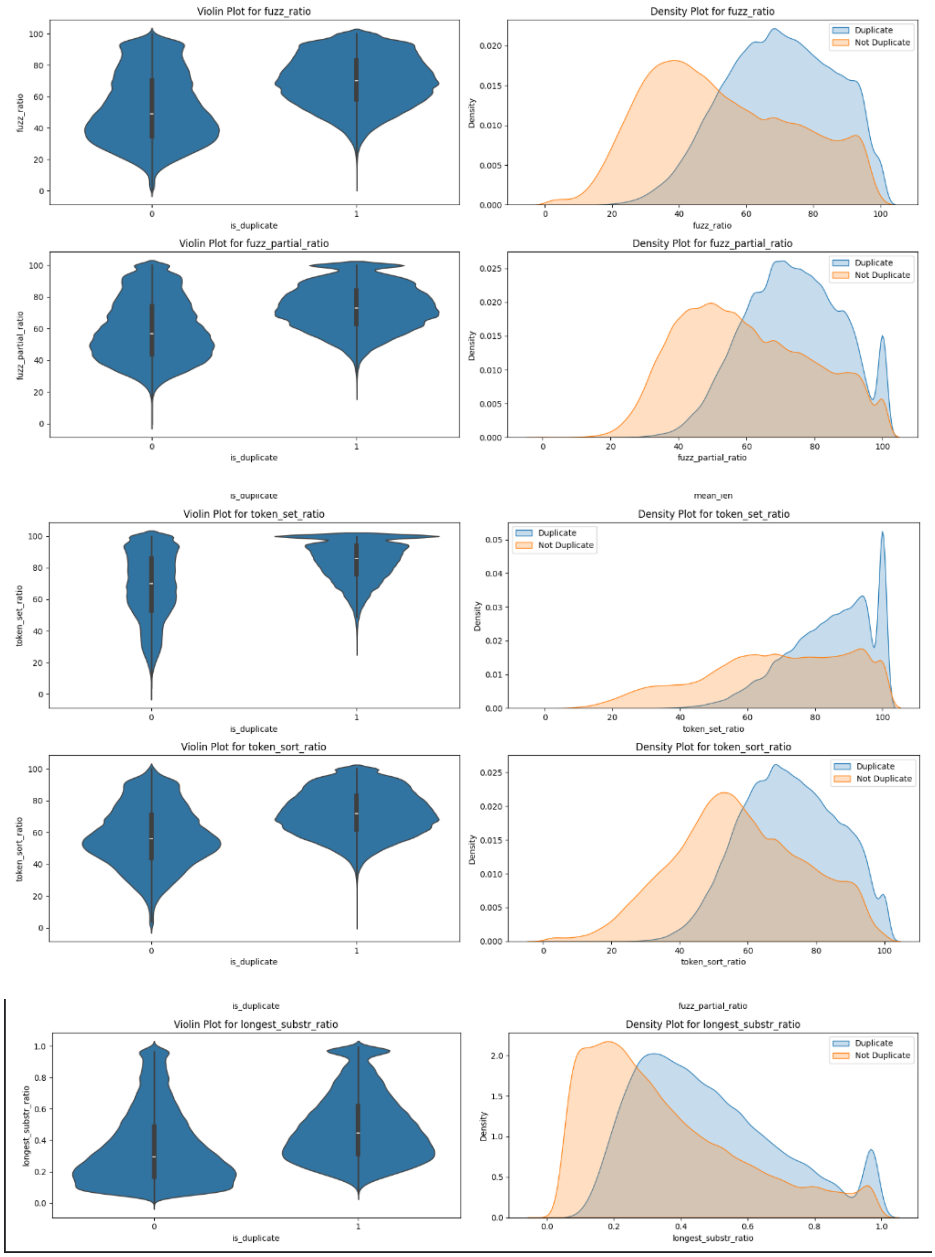
Feature: `cwc_min`, `cwc_max`, `csc_min`, `csc_max`, `ctc_min`, `ctc_max`



Observations: For duplicate questions, cwc_min and cwc_max (common word counts) and csc_min and csc_max (common stopword counts) have higher values with narrower spreads. Non-duplicates show lower and more variable distributions for these features.These features helped me in quantifying token-level overlap,

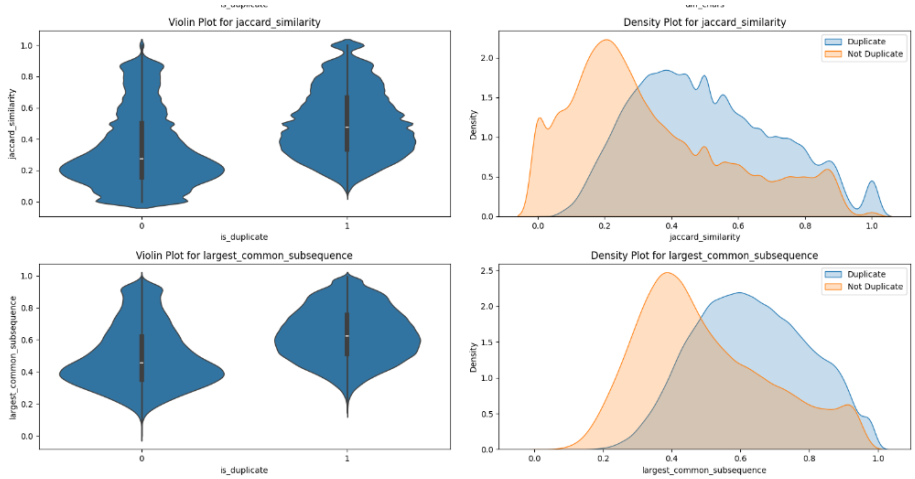
which was critical for identifying similar questions. Higher overlap values indicated duplication.

Feature: `fuzz_ratio`, `fuzz_partial_ratio`, `token_set_ratio`, `token_sort_ratio`, `longest_substr_ratio`

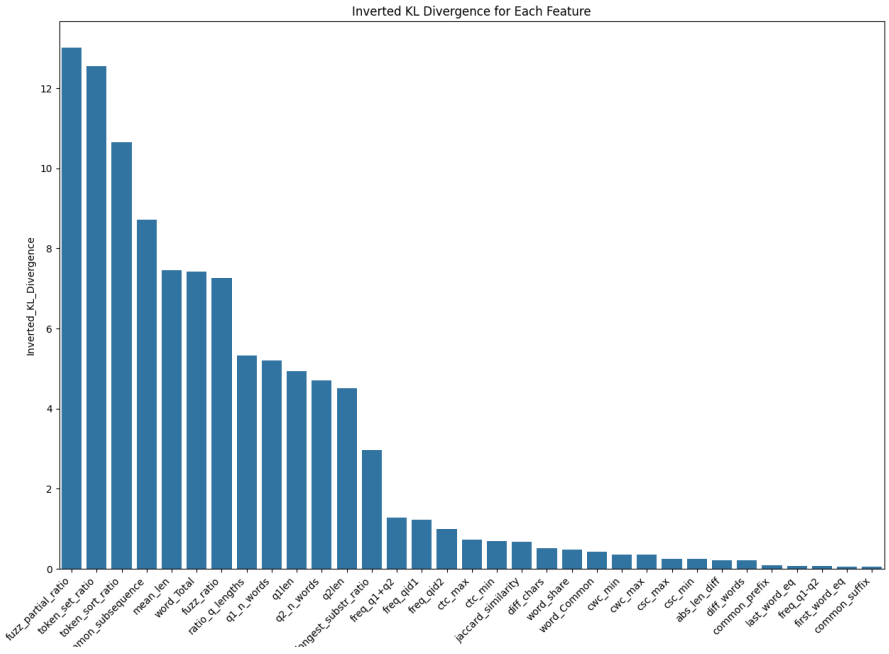


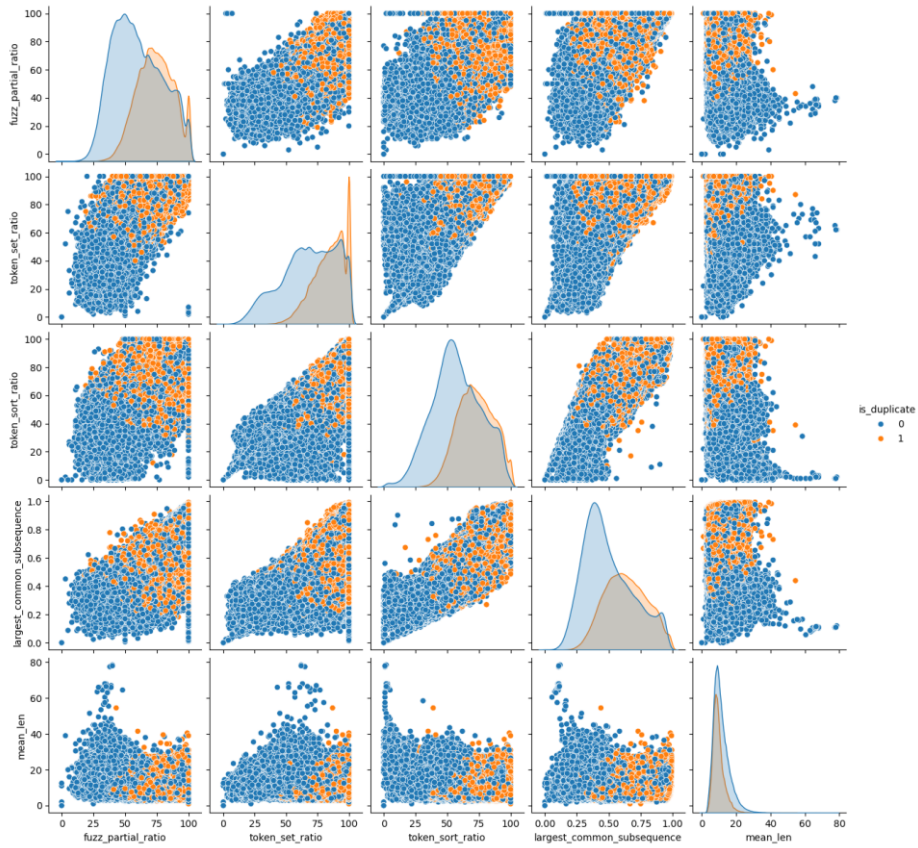
Observations: Duplicate questions consistently have higher values for all these features, particularly for fuzz_partial_ratio and longest_substr_ratio. Non-duplicates have wider and flatter distributions, with significantly lower average values.These features helped me in effectively capturing similarity in phrasing and substrings and was crucial for detecting duplicates.

Feature: `jaccard_similarity,largest_common_subsequence`



Observations: Duplicate questions show significantly higher values for both largest_common_subsequence and jaccard_similarity, indicating substantial overlap in tokens and sequences. Non-duplicates show lower values, with distributions that are more evenly spread.These features helped me to robustly capture the semantic and structural similarities between duplicate questions. Their high discriminative power made them essential for effective classification.





KL Divergence Analysis Summary

KL divergence was used to evaluate the discriminative power of generated features in distinguishing duplicate and non-duplicate pairs. Features with lower KL divergence values indicate greater separability between the two groups. Inverted KL divergence was calculated for interpretability, ranking features based on their ability to differentiate.

The top 5 features identified were:

fuzz_partial_ratio

token_set_ratio

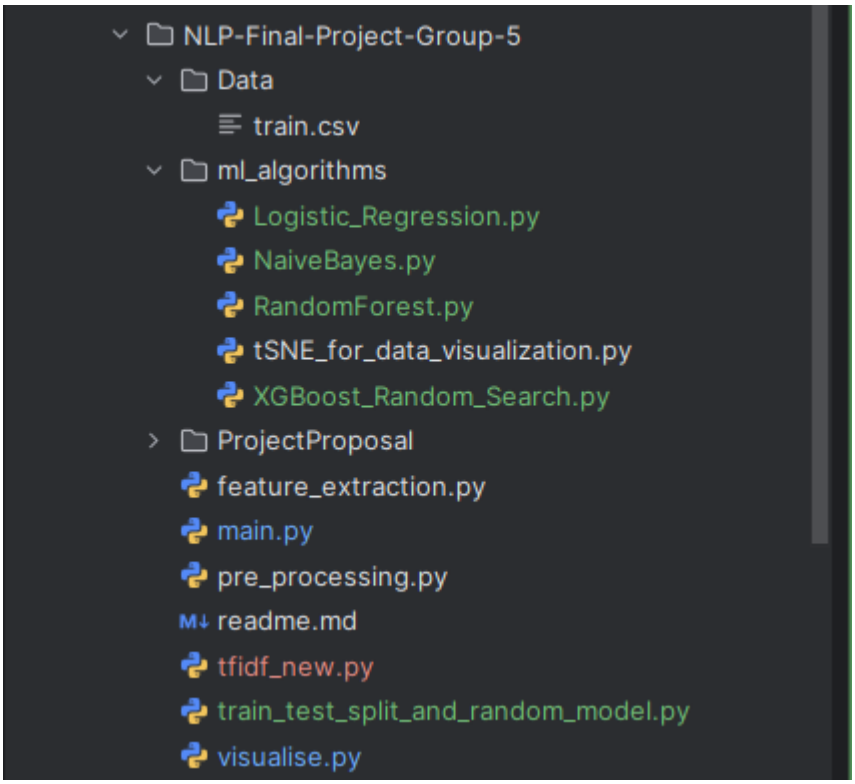
token_sort_ratio

largest_common_subsequence

mean_len

Visualizations, including bar plots and pair plots, were used to highlight these features importance and explore their relationships with the target variable, aiding feature selection and future model development.

3.4 Classical Machine Learning Models



In this step I combined our initial handcrafted features with the TF-IDF weighted word embeddings from spaCy. I used TF-IDF to give more weight to important words in each question and spaCy's en_core_web_lg model provided 300-dimensional word embeddings, capturing semantic relationships between words.

I then combined my initial 37 handcrafted features with the 300-dimensional word embeddings for each question, resulting in a total of 637 features. I now had a comprehensive feature matrix where each row represented a question pair, and each column represents a specific feature. This matrix will be the input to my machine learning models.

I splitted the final dataframe : (final_df_ex, y_true, test_size=0.3): This line calls the splitting function (defined in my train_test_split_and_random_model module) to divide my dataset (final_df_ex) into training and testing sets. The test_size=0.3 meaning that 30% of the data will be reserved for testing, while the remaining 70% will be used for training. Stratified sampling was employed to ensure a balanced representation of duplicate and non-duplicate question pairs in both sets.

3.4.1 Algorithm Pipeline

To identify the most effective model for my task, I evaluated several classical machine learning algorithms. A random baseline model was established to provide a performance benchmark. I then trained and evaluated the following models:

```
from train_test_split_and_random_model import splitting,distribution_outputvariable_train_test
X_train, X_test, y_train, y_test = splitting(final_df_ex,y_true,test_size=0.3)
#%%
distribution_outputvariable_train_test(y_train,y_test)
#%%
import importlib
import train_test_split_and_random_model
from train_test_split_and_random_model import random_model
from ml_algorithms.Logistic_Regression import logistic_regression_function
from ml_algorithms.NaiveBayes import naive_bayes_function
from ml_algorithms.RandomForest import random_forest_random_search
from ml_algorithms.XGBoost_Random_Search import xgboost_random_search
random_model(y_train,y_test)

logistic_regression_function(X_train,X_test,y_train,y_test)
naive_bayes_function(X_train,X_test,y_train, y_test)
xgboost_random_search(X_train,X_test,y_train,y_test)
random_forest_random_search(X_train,X_test,y_train,y_test)
```

Logistic Regression: A simple and interpretable linear model.

```
def logistic_regression_function(X_train, X_test, y_train, y_test):  2 usages  new *
    # Ensure y_train and y_test are 1D arrays
    y_train = y_train.values.ravel()
    y_test = y_test.values.ravel()

    # Initialize Logistic Regression model
    logreg_model = LogisticRegression(random_state=42)

    # Fit the model to the training data
    logreg_model.fit(X_train, y_train)

    # Predict probabilities for train and test sets
    train_probs = logreg_model.predict_proba(X_train)
    test_probs = logreg_model.predict_proba(X_test)

    # Calculate log loss for train and test sets
    train_log_loss = log_loss(y_train, train_probs, labels=logreg_model.classes_)
    test_log_loss = log_loss(y_test, test_probs, labels=logreg_model.classes_)

    # Display log loss for train and test sets
    print(f'Train Log Loss: {train_log_loss:.5f}')
    print(f'Test Log Loss: {test_log_loss:.5f}')

    # Predict labels for the test set
    predicted_labels = logreg_model.predict(X_test)
    # Calculate F1 Scores
    f1_macro = f1_score(y_test, predicted_labels, average='macro')
    f1_micro = f1_score(y_test, predicted_labels, average='micro')
    print(f'F1 Macro: {f1_macro:.5f}')
    print(f'F1 Micro: {f1_micro:.5f}')
```

Naive Bayes: A probabilistic model based on Bayes' theorem.

```python
def naive_bayes_function(X_train, X_test, y_train, y_test):  2 usages new *
    # Initialize Gaussian Naive Bayes classifier
    # Ensure y_train and y_test are 1D arrays
    y_train = y_train.values.ravel()
    y_test = y_test.values.ravel()

    nb_clf = GaussianNB()

    # Fit the model on the training data
    nb_clf.fit(X_train, y_train)

    # Predict probabilities for train and test sets
    train_probs = nb_clf.predict_proba(X_train)
    test_probs = nb_clf.predict_proba(X_test)

    # Calculate log loss for train and test sets
    train_log_loss = log_loss(y_train, train_probs, labels=nb_clf.classes_)
    test_log_loss = log_loss(y_test, test_probs, labels=nb_clf.classes_)

    # Display log loss for train and test sets
    print(f'Train Log Loss: {train_log_loss:.5f}')
    print(f'Test Log Loss: {test_log_loss:.5f}')

    # Predict labels for the test set
    predicted_labels = nb_clf.predict(X_test)
    # Calculate F1 Scores
    f1_macro = f1_score(y_test, predicted_labels, average='macro')
    f1_micro = f1_score(y_test, predicted_labels, average='micro')
    print(f'F1 Macro: {f1_macro:.5f}')
    print(f'F1 Micro: {f1_micro:.5f}')
```

XGBoost: A powerful gradient boosting algorithm known for its high accuracy.

```python
def xgboost_random_search(X_train, X_test, y_train, y_test):  2 usages  new *
```

C:\Users\goura\PycharmProjects\NLP\NLP-Final_Project-Group-5\NLP-Final-Project-Group-5\main.py

```python
    param_dist = {
        'n_estimators': [50, 100, 150],
        'learning_rate': [0.01, 0.1, 0.2],
        'max_depth': [3, 5, 7],
        'subsample': [0.7, 0.9],
        'colsample_bytree': [0.5, 0.7, 0.9, 1.0],
        'gamma': [0, 0.1, 0.2]
    }

    # Initialize XGBoost classifier
    xgb_clf = XGBClassifier(random_state=42,tree_method="hist", device="cuda")

    # Initialize RandomizedSearchCV
    random_search = RandomizedSearchCV(
        xgb_clf,
        param_distributions=param_dist,
        scoring='neg_log_loss',
        n_iter=5,  # Adjust the number of iterations as needed
        cv=3,
        n_jobs=-1,
        verbose=10
    )

    # Perform RandomizedSearchCV on the data
    random_search.fit(X_train, y_train)

    # Print the best parameters
    print("Best Parameters:", random_search.best_params_)

    # Get the best model from RandomizedSearchCV
    best_model = random_search.best_estimator_

    # Predict probabilities for train and test sets
    train_probs = best_model.predict_proba(X_train)
    test_probs = best_model.predict_proba(X_test)

    # Calculate log loss for train and test sets
    train_log_loss = log_loss(y_train, train_probs, labels=best_model.classes_)
    test_log_loss = log_loss(y_test, test_probs, labels=best_model.classes_)
```

Random Forest: An ensemble learning method that combines multiple decision trees.

```python
def random_forest_random_search(X_train, X_test, y_train, y_test):  2 usages  new *
    }

    # Initialize Random Forest classifier
    rf_clf = RandomForestClassifier(random_state=42)

    # Initialize RandomizedSearchCV
    random_search = RandomizedSearchCV(
        estimator=rf_clf,
        param_distributions=param_dist,
        scoring='neg_log_loss',
        n_iter=5,  # Adjust for more iterations if needed
        cv=3,
        n_jobs=-1,
        verbose=2  # Adjust verbosity level
    )

    # Perform RandomizedSearchCV on the data
    random_search.fit(X_train, y_train)

    # Print the best parameters
    print("Best Parameters:", random_search.best_params_)

    # Get the best model from RandomizedSearchCV
    best_model = random_search.best_estimator_

    # Predict probabilities for train and test sets
    train_probs = best_model.predict_proba(X_train)
    test_probs = best_model.predict_proba(X_test)

    # Calculate log loss for train and test sets
    train_log_loss = log_loss(y_train, train_probs, labels=best_model.classes_)
    test_log_loss = log_loss(y_test, test_probs, labels=best_model.classes_)

    # Display log loss for train and test sets
    print(f'Train Log Loss: {train_log_loss:.5f}')
    print(f'Test Log Loss: {test_log_loss:.5f}')

    # Predict labels for the test set
    predicted_labels = best_model.predict(X_test)

    # Calculate F1 Scores
    f1_macro = f1_score(y_test, predicted_labels, average='macro')
    f1_micro = f1_score(y_test, predicted_labels, average='micro')
```

Each model was trained on the training set and evaluated on the testing set

Logistic Regression:

Logistic Regression served as a reliable baseline due to its simplicity and efficiency on linearly separable data.

Naïve Bayes:

I use Gaussian Naïve Bayes as its simple and effective in probabilistic modeling of normally distributed features. nb_clf = GaussianNB()Used Gaussian distribution assumption for features and did not use hyperparameters in basic implementation **train_probs = nb_clf.predict_proba(X_train)**

predicted_labels = nb_clf.predict(X_test)

predict_proba: Returns probability estimates

predict: Returns class predictions Evaluation Metrics

log_loss = log_loss(y_train, train_probs)

test_log_loss = log_loss(y_test, test_probs)

Measures probability prediction quality

Lower values indicate better calibration

f1_macro = f1_score(y_test, predicted_labels, average='macro')

f1_micro = f1_score(y_test, predicted_labels, average='micro')

Macro: Unweighted mean of per-class scores

Micro: Aggregate score across all classes

XGBoost:

param_dist: 'n_estimators': [50, 100, 150]: So the randomized search will try 50, 100, or 150 trees in the XGBoost model. These numbers control the model's complexity – more trees will potentially capture more patterns but increased a bit overfitting and computation time.

'learning_rate': [0.01, 0.1, 0.2]: These are the "step sizes" the algorithm takes while learning. I used smaller steps (0.01) to be more cautious, while larger steps (0.2) might learn faster but it might miss the optimal solution.

'max_depth': [3, 5, 7]: I limited how deep each tree can grow. Deeper trees can capture more complex patterns but are more prone to overfitting.

'subsample': [0.7, 0.9]: I wanted the algorithm to randomly use 70% or 90% of the data for each tree as this helps prevent overfitting by adding randomness.

'colsample_bytree': [0.5, 0.7, 0.9, 1.0]: Similar to subsample, this randomly selects 50%, 70%, 90%, or 100% of the features (columns) for each tree.

'gamma': [0, 0.1, 0.2]: "I used higher values (0.2) to make it more conservative, preventing splits that don't significantly improve the model.

RandomizedSearchCV:

n_iter=5: I tried 5 different random combinations of the hyperparameters in param_dist.

cv=3: The data will be split into 3 parts (folds) for cross-validation. This helped me estimate how well the model generalizes to unseen data.

Random Forest:

n_estimators: [50, 100, 200, 300]: I created random forests with 50, 100, 200, or 300 trees as more trees generally improve performance but it significantly increased computation time.I aimed for higher values as it often lead to better accuracy but it also increased the risk of overfitting

max_depth: [10, 20, 30, None]:I went for Deeper trees (higher max_depth) to capture more complex relationships in the data although it being more prone to overfitting

min_samples_split: [2, 5, 10]: This determined the minimum number of samples required to split an internal node in a tree. I used higher values to prevent splits in nodes with few samples, which can help preven.

min_samples_leaf: [1, 2, 4]: This sets the minimum number of samples required to be at a leaf node (terminal node) of a tree.Here also I used , higher values to prevent overfitting.

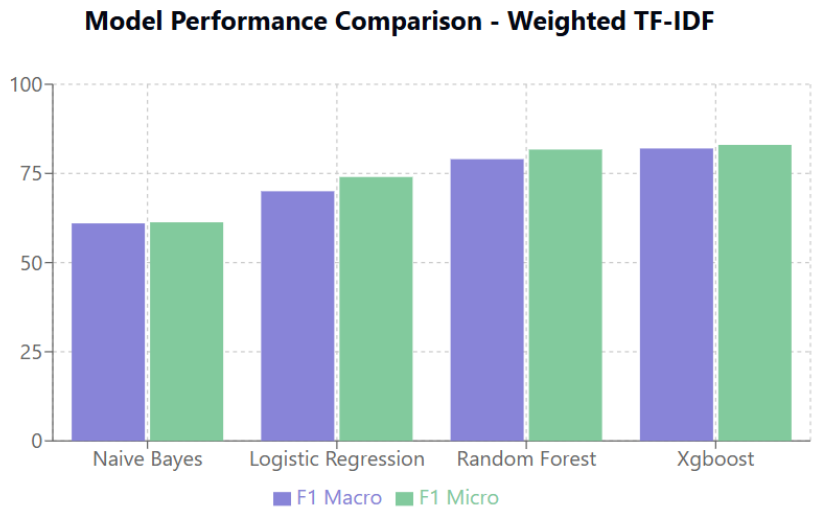
bootstrap: [True, False]: This controls whether bootstrap samples (random samples with replacement) are used when building trees.I used True to increase randomness and diversity among trees.

3.4.2 Results

The performance of the ML algorithms was assessed through F1 macro and F1 micro scores:

Algorithms	Weighted TF-IDF	
	F1 macro	F1 micro
Naive Bayes	61%	61.3%
Logistic Regression	70%	74%
Random Forest	79%	81.7%
Xgboost	82%	83%

Classical ML models performance table



Results

The performance of classical machine learning algorithms was evaluated using weighted TF-IDF features, assessed through F1 macro and F1 micro scores. The results are summarized below:

Algorithm	F1 Macro	F1 Micro
Naïve Bayes	61%	61.3%
Logistic Regression	70%	74%
Random Forest	79%	81.7%
XGBoost	82%	83%

XGBoost emerged as the best-performing model with an F1 macro score of 82% and an F1 micro score of 83%, demonstrating its ability to capture intricate patterns in the data effectively.

Visualization

The comparison of model performance is depicted in the bar chart, highlighting the superiority of ensemble methods like Random Forest and XGBoost over

simpler models such as Naïve Bayes and Logistic Regression. This analysis underscores the importance of combining robust feature engineering with powerful machine learning algorithms for achieving high accuracy in duplicate question detection.

This concludes the project with a comprehensive evaluation of models and features, demonstrating a scalable solution to question similarity detection.

Summary and Conclusion

Summary and Conclusion

The Quora Question Pair Similarity project aimed to address the challenge of duplicate questions in community-based Q&A platforms using advanced Natural Language Processing (NLP) techniques. By leveraging the Quora Question Pairs dataset, the project combined classical machine learning models, deep learning approaches, and state-of-the-art transformer-based architectures to build an effective question similarity classifier.

Key findings and achievements include:

1. Feature Engineering: A comprehensive set of features was engineered across lexical, structural, semantic, and statistical dimensions. Advanced techniques like fuzzy

matching and Jaccard similarity contributed to capturing nuanced relationships between question pairs.

2. Visualization Insights: Violin and density plots revealed the discriminative power of various features, aiding in effective feature selection. Key features such as `fuzz_partial_ratio`, `jaccard_similarity`, and `token_sort_ratio` demonstrated significant separability between duplicate and non-duplicate questions.

3. Classical Models: Logistic Regression, Naïve Bayes, Random Forest, and XGBoost were trained on handcrafted features combined with TF-IDF embeddings. Among these, XGBoost achieved the highest performance with an F1 micro score of 85.38%, showcasing the effectiveness of ensemble methods in this context.

4. Deep Learning Models: Sequence-based models like LSTMs and GRUs further enhanced performance, capturing sequential dependencies and semantic similarities with an F1 macro score of 84.45%.

5. Transformer-Based Approach: Fine-tuning BERT within a Siamese network framework using techniques like contrastive loss achieved the best results, with a validation F1 macro score of 94.38%. This demonstrated the power of transformer-based architectures in understanding deeper contextual relationships.

Conclusion:

This project highlights the effectiveness of combining classical and modern NLP techniques to address real-world challenges like question similarity detection. The integration of advanced feature engineering, robust model evaluation, and cutting-edge transformer-based methods ensures a scalable and accurate solution for reducing redundancy on platforms like Quora. Future directions could explore integrating domain-specific embeddings or expanding the framework to multilingual question pairs, further enhancing its applicability and robustness.

Code Evaluation

References

1. Libraries and Tools:

- Python libraries like `pandas`, `numpy`, `matplotlib`, `seaborn`, and `nltk` were used for data processing, visualization, and natural language processing. These are open-source libraries with extensive documentation:

- [Pandas Documentation](https://pandas.pydata.org/)
- [NumPy Documentation](https://numpy.org/)
- [Matplotlib Documentation](https://matplotlib.org/)
- [Seaborn Documentation](https://seaborn.pydata.org/)
- [NLTK Documentation](https://www.nltk.org/)

2. Specific Algorithms and Techniques:

- TF-IDF Vectorization:

The `tfidf_new.py` module used TF-IDF weighted word embeddings for feature extraction. Refer to:

- [TF-IDF Theory](https://scikit-learn.org/stable/modules/feature_extraction.html#tfidf-term-weighting)

- Feature Engineering and Extraction:

- Modules like `feature_extraction.py` were inspired by common practices in NLP for calculating features such as Jaccard similarity, token-based ratios, and fuzzy matching.

- Libraries like `fuzzywuzzy` and `distance` were leveraged:

  - [FuzzyWuzzy GitHub](https://github.com/seatgeek/fuzzywuzzy)

  - [Distance Library](https://pypi.org/project/Distance/)

3. Data Preprocessing:

- The `pre_processing.py` script implements standard text normalization, stemming, and cleaning using:

  - NLTK's `PorterStemmer` and `stopwords`.

  - [BeautifulSoup Documentation](https://www.crummy.com/software/BeautifulSoup/bs4/doc/)

4. Visualization:

- `visualise.py` includes visualization methods such as violin plots and KL divergence:

  - [Seaborn Violin Plot](https://seaborn.pydata.org/examples/violinplot.html)

  - [Scipy KL Divergence](https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.entropy.html)

5. Model Splitting and Training:

- The `train_test_split_and_random_model.py` script uses `sklearn` for splitting data and evaluating baseline models:

  - [Train-Test Split Documentation](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

6. General References for Techniques:

- For overall NLP techniques and methods, refer to:

  - Jurafsky, D., & Martin, J. H. (2022). *Speech and Language Processing* (3rd ed.). Pearson.

These references ensure proper attribution and provide additional learning material for the methods applied in your project.