QUORA QUESTION PAIR SIMILARITY

Natural Language Processing (DATS 6312)

Final Project Individual Contribution Report

Name: Shreya Sahay

Group: 5

Date: 12/11/2024

TABLE OF CONTENTS

Introduction

Community-based question-answering forums such as Reddit and Quora have seen significant growth in recent years. Quora alone attracts over 100 million monthly visitors. However, the rapid increase in the number of questions has led to a surge in queries with the same underlying meaning. These create challenges by making it more time-consuming for users to navigate to the most relevant answers and by placing additional strain on contributors who must address multiple versions of similar questions. This project focuses on leveraging the Quora Question Pairs dataset to build a question similarity classifier using advanced deep learning techniques.

These advanced architectures allowed us to capture deeper semantic relationships between questions and significantly improved the model's understanding of context and intent. After extensive experimentation and hyperparameter tuning, we achieved a test accuracy of 93.28%, showcasing the effectiveness of transformer-based Siamese networks in solving complex NLP problems like question pair similarity.

Dataset

The dataset is sourced from Kaggle [1] and was introduced as part of a competition on the platform for the same challenge as our project.

The Quora Question Pairs dataset is composed of 404,290 question pairs, and has 6 total features: ***id, qid1, qid2, question1, question2, is_duplicate***:

id	qid1	qid2	question1	question2	is_duplicate
38	61	62	What's one thing you would like to do better?	What's one thing you do despite knowing better?	0
180	361	362	How do you get deleted Instagram chats?	How can I view deleted Instagram dms?	1

Table 1: A snapshot of the Quora Question Pairs Dataset

The dataset is imbalanced – approximately 36.92% are duplicate questions while 63.08% are non-duplicate pairs:

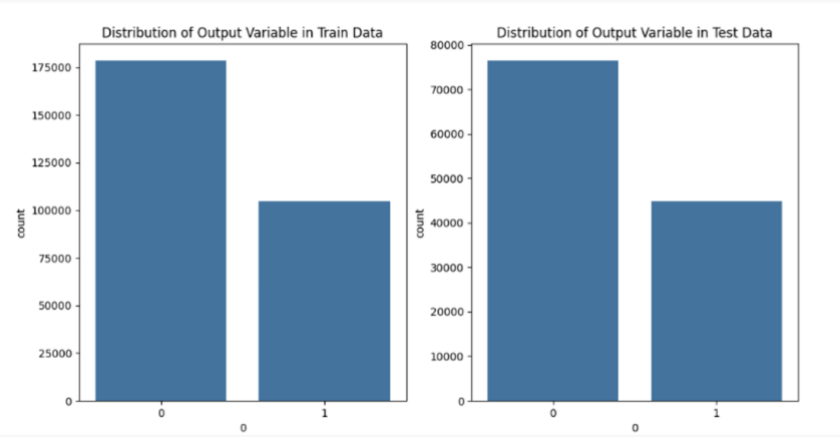


Figure 1: A bar plot showcasing the imbalance in the data

Shared Work Outline:

1. Exploratory Data Analysis (EDA)

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

1. Data Normalization: Converting text to lowercase, expanding contractions, and standardizing symbols (e.g., currency, percentages).
2. Data Cleaning: Removing non-word characters and HTML tags, and processing large numbers into a readable format.
3. Stemming: Applied the PorterStemmer to reduce words to their root forms.

3. Feature Extraction

1. Tokenization: Tokenizing the questions using a word tokenizer.
2. Feature Categories: Extracted 37 features that fall into three main categories:
3. Fuzzy Features: Including fuzz_ratio, fuzz_partial_ratio, etc., to measure word-to-word fuzzy similarity.
4. Token Features: Analysis of stopwords and non-stopwords (e.g., common non-stopwords, common stopwords, word size difference).
5. Common Subsequence Features: Measuring similarity between parts of sentences (e.g., largest_common_subsequence, jaccard_similarity).

4. Vectorization

1. 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.

2. BERT Embeddings: Created BERT embeddings

5. Model Training and Comparison

1. Classical Models with TF-IDF and BERT embeddings: Trained classical machine learning models (e.g. Logistic Regression, Random Forest, Naïve Bayes) using the weighted TF-IDF and the BERT embedding features.
2. DL Models: Trained and evaluated DL models: LSTM and GRU
3. Transformer Models: Applied different variations of the BERT architecture

6. Streamlit app

7. Final Model and Evaluation

1. Compared and evaluated all the applied models
2. Evaluation Metrics: Used the F1 macro score for model evaluation.

Description of Individual Work

3.1 Background and overview, outline of shared work

As part of this project, I generated BERT-based features alongside previously created 37 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 Extended features generation

Before applying any form of modelling, I generated BERT-based CLS embeddings of the 'question1' and 'question2' features to obtain better contextual meaning of each of the questions, enabling effective comparison of semantic similarities between question pairs.

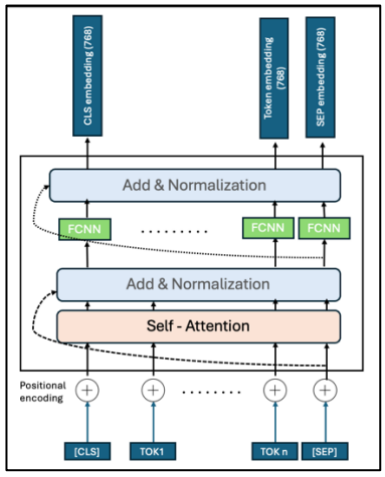


Figure 3: Encoder architecture of BERT

This operation is started off with fetching the raw question1 and question2 pairs from the train data and storing it in a custom dataset class named **`QuestionsDataset()`** for batch processing. Another function, named **`compute_batch_embeddings()`** is designed for generating the embeddings:

```python
def compute_batch_embeddings(batch_sentences):
    inputs = tokenizer(
        batch_sentences,
        return_tensors="pt",
        truncation=True,
        padding=True,
        max_length=512
    ).to(device)

    with torch.no_grad():
        outputs = bert_model(**inputs)
        cls_embeddings = outputs.last_hidden_state[:, 0, :]  # CLS embeddings
    return cls_embeddings.cpu().numpy()
```

Figure 4: The *compute_batch_embeddings()* function for generating embeddings

The batch of tokens are converted into token IDs using the pretrained **`BertTokernizer()`** function (‘*bert-base-uncased*’). The line

```
cls_embeddings = outputs.last_hidden_state[:, 0, :]
```

extracts the embedding vector corresponding to the **CLS** token (the first token in each sequence).

Next, the **`QuestionsDataset()`** custom dataset is instantiated and a DataLoader function with ***batch_size*** of 8 is created. For each batch of sentences, we get a (8, 768)-dim array representing the semantic embedding of each sentence. After the full operation, we obtain a pair of (N,768) array of numerical embeddings for each question pair, where N = 404,290.

These embeddings are then merged with the previously created 37 features into a single feature matrix and saved in .pt format for ease of usage. This .pt file has all the relevant features that can be further fed into models: **{id,q1_feats_bert,q2_feats_bert,features,labels}**, where ***q1_feats_bert*** and ***q2_feats_bert*** are the question pair embeddings and ***labels*** is the target column with 0s and 1s.

3.3 Classical Machine Learning Models

Our newly created feature matrix is now nearly ready to be used for modelling. Prior to that, it is essential to further combine the base features and embedding features into one single feature and then split it into different modelling sets. After

assessment of the size of the data, we perform a (90:5:5)% split to create the train, validation and test sets.

3.3.1 Algorithm Pipeline

I created a pipeline for running the models on the code for each of the classical algorithms: Naïve Bayes, Logistic Regresssion, XGBoost and Random Forest [3].

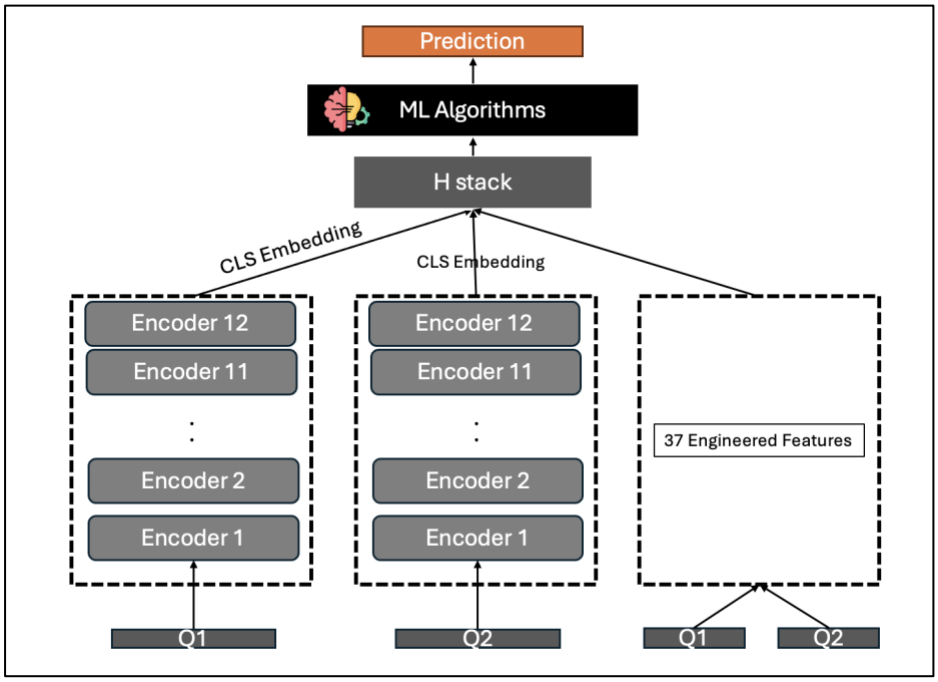


Figure 5: ML architecture pipeline

This pipeline dynamically constructs a machine learning workflow tailored to a specified algorithm. Each algorithm is configured with appropriate preprocessing steps and hyperparameters for optimal performance:

```python
def create_pipeline(algorithm):
    """Create a pipeline based on the selected algorithm."""
    if algorithm == "random_forest":
        return Pipeline([
            ('classifier', RandomForestClassifier(
                n_estimators=100,
                max_depth=None,  # No restriction on depth
                min_samples_split=2,  # Minimum samples to split is 2
                min_samples_leaf=1,  # Minimum samples in a leaf is 1
                random_state=42
            ))
        ])
    elif algorithm == "logistic_regression":
        return Pipeline([
            ('scaler', StandardScaler()),  # Scaling required for Logistic Regression
            ('classifier', LogisticRegression(max_iter=1000, random_state=42))
        ])
    elif algorithm == "svm":
        return Pipeline([
            ('scaler', StandardScaler()),  # Scaling required for SVM
            ('classifier', SVC(kernel='rbf', probability=True, random_state=42))  # RBF kernel added
        ])
    elif algorithm == "xgboost":
        return Pipeline([
            ('classifier', xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss', random_sta
        ])
    elif algorithm == "naive_bayes":
        return Pipeline([
            ('scaler', StandardScaler()),  # Standardization for GaussianNB
            ('classifier', GaussianNB())  # Gaussian Naive Bayes classifier
        ])
    else:
        raise ValueError(f"Unsupported algorithm: {algorithm}")
```

Figure 6: Code of complete ML pipeline

Random Forest:

I selected Random Forest for its ability to handle high-dimensional data and complex relationships using an ensemble of decision trees. Setting **`n_estimators=100`** ensured stable predictions, while **`max_depth=None`**, **`min_samples_split=2`**, and **`min_samples_leaf=1`** allowed detailed pattern capture and flexibility. Overfitting was encouraged in base estimators to improve overall model performance.

Logistic Regression:

Logistic Regression served as a reliable baseline due to its simplicity and efficiency on linearly separable data. Using **StandardScaler** ensured standardized features

for gradient optimization, and **`max_iter=1000`** supported convergence on complex datasets.

XGBoost:

I included XGBoost for its robust performance on structured data and its ability to handle imbalanced datasets. With **`use_label_encoder=False`** for compatibility and **`eval_metric='logloss'`** for classification accuracy, standardization was unnecessary due to its robustness to scaling.

Naïve Bayes:

Gaussian Naïve Bayes was chosen for its simplicity and effectiveness in probabilistic modeling of normally distributed features. Applying **StandardScaler** addressed numerical instability, and the model provided a fast, interpretable benchmark.

Support Vector Machine (SVM):

While SVM with an RBF kernel was chosen for non-linear relationships, it failed to execute efficiently, crashing the IDE due to prolonged runtime.

3.3.2 Results

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

Algorithm	F1-macro	F1-micro
Naïve Bayes	61.82%	62.17%
Logistic Regression	79.31%	81.23%
Random Forest	82.04%	83.97%
XGBoost	84.10%	85.38%

Table 2: Classical ML models performance table

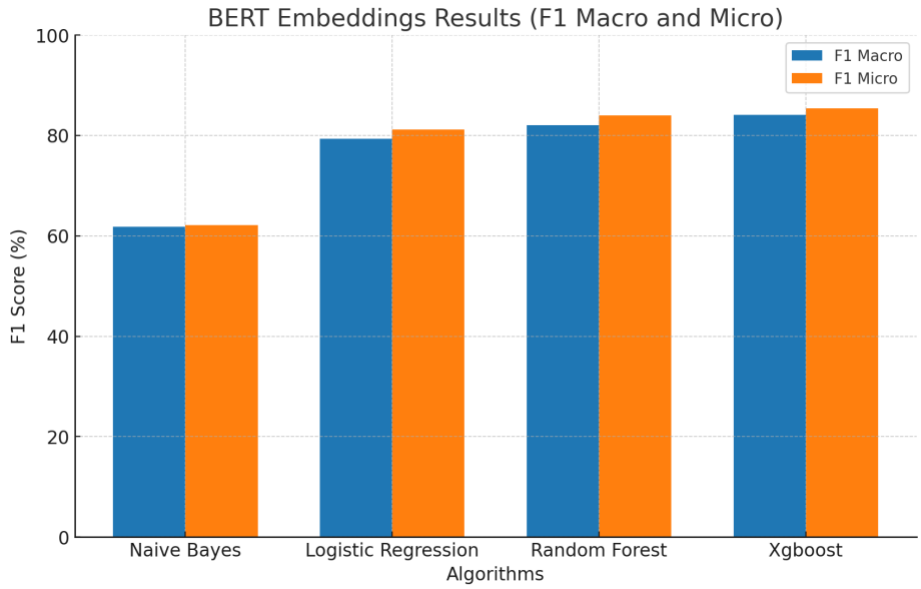


Figure 7: Classical ML models performance chart

Since the classical models did not produce satisfactory results, I moved on to Deep Learning and Transformer-based models.

3.3 Deep Learning Models

3.3.1 DataLoader and Algorithm Pipeline

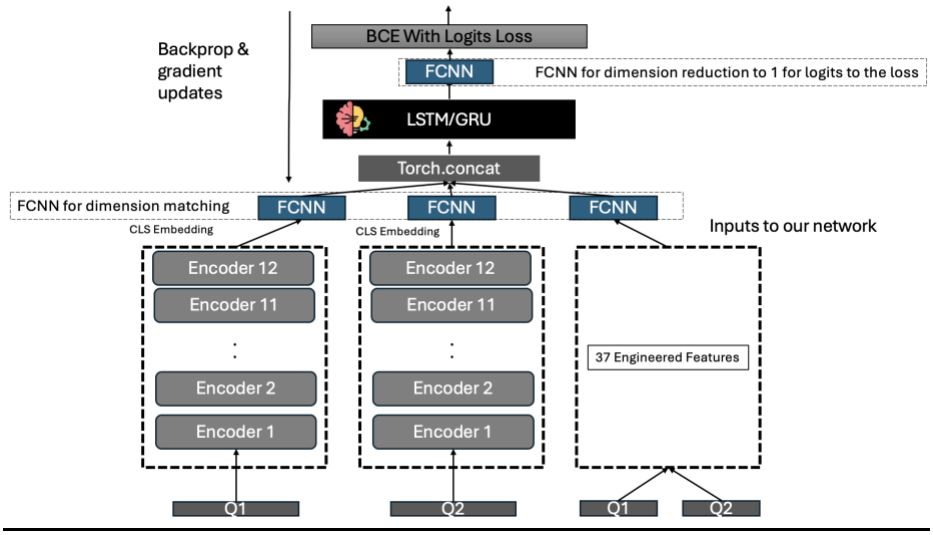


Figure 8: Deep Learning Models Architecture

I selected the GRU and LSTM networks and in both cases the data was passed through the models using a DataLoader:

```python
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):
    def __init__(self, data):
        self.ids = data['id']
        self.q1_feats = data['q1_feats_bert']
        self.q2_feats = data['q2_feats_bert']
        self.features = data['features']
        self.labels = data['labels']

    def __len__(self):
        return len(self.ids)

    def __getitem__(self, idx):
        return (
            self.q1_feats[idx],
            self.q2_feats[idx],
            self.features[idx],
            self.labels[idx]
        )

def load_data(data, batch_size=32):
    dataset = CustomDataset(data)
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    return dataloader
```

Figure 9: Deep Learning Models Architecture

I defined a **CustomDataset()**, to structure the data so each sample includes the feature matrix. I then created a **DataLoader**, allowing me to efficiently load the data in batches of size 32.

In the model pipeline, I designed two models to process embeddings of two questions (q1 and q2) along with additional features. To reduce their dimensions, I passed q1 and q2 embeddings (initially of size 768) through a shared fully connected layer, mapping them to **`hidden_dim=256`**. For the additional features, I used a separate fully connected layer to transform them into the same 256-dimensional space, ensuring all inputs were in a consistent representation.

I concatenated the outputs of the shared layer for q1 and q2 with the processed additional features into a single tensor of dimensions **`[batch_size, 3 * hidden_dim]`**. This combined representation served as the input for the model's head, which could be either an LSTM or GRU. For the LSTM head, I reshaped the

tensor to **`[batch_size, 1, 3 * hidden_dim]`** and passed it through the LSTM, extracting the last hidden state (**`head_out[-1]`**) of size [**`batch_size, hidden_dim`**]. Similarly, for the GRU head, I reshaped the input, passed it through the GRU, and took the last hidden state (**`head_out[-1]`**) as the final output. Both heads effectively captured relationships between the questions and features. The training process included checkpointing, where the model's state was saved whenever a new best validation loss was achieved. This allowed me to resume training or evaluate the best-performing model at any point.

Finally, I passed the head's output through a fully connected layer to produce a single prediction. I experimented with various learning rates, increasing epochs, and modifying batch sizes to improve performance, but these tweaks did not help. The loss oscillated between 0.65 and 0.69 for both LSTM and GRU without convergence. Despite these efforts, I achieved a final loss of 0.65 for LSTM and 0.68 for GRU, suggesting the architecture required further adjustments to improve performance.

```
model = BaseModel(head_type=HEAD_TYPE, feature_dim=FEATUI
optimizer = Adam(model.parameters(), lr=LEARNING_RATE)
criterion = BCEWithLogitsLoss()
```

Figure 10: Calling the model pipeline

The model was passed through the training loop in the above manner with the **HEAD_TYPE** being modified to dictate the type (LSTM/GRU).

3.3.2 Results

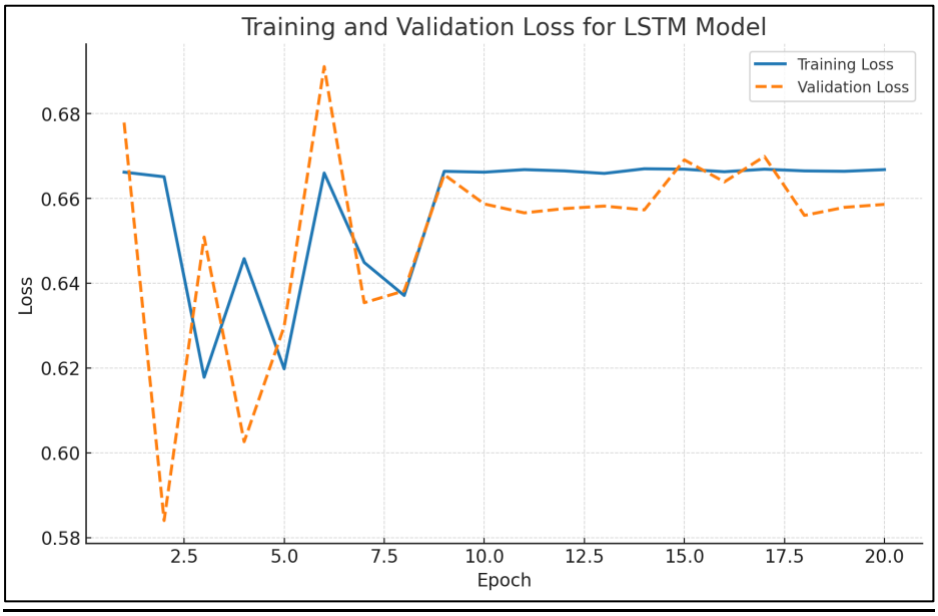


Figure 11: LSTM performance

As can be observed above in the validation and training loss plot for LSTM, the model returns high loss values on the data and as mentioned in the previous section, was unable to perform well.

The deep learning models failed to produce satisfactory results. I therefore went ahead with applying Transformer-based models.

3.5 Transformer Models

As part of transformer models, I implemented a modified version of the Siamese Sentence BERT framework [2]. The motivation for using this framework was to leverage SBERT's ability to generate semantically rich embeddings while incorporating additional fine-tuning steps to adapt it to the specific task. SBERT was particularly suited for this problem as it could effectively capture the nuanced relationships between pairs of text inputs, which is essential for understanding question similarity.

3.5.1 DataLoader and Model Architecture

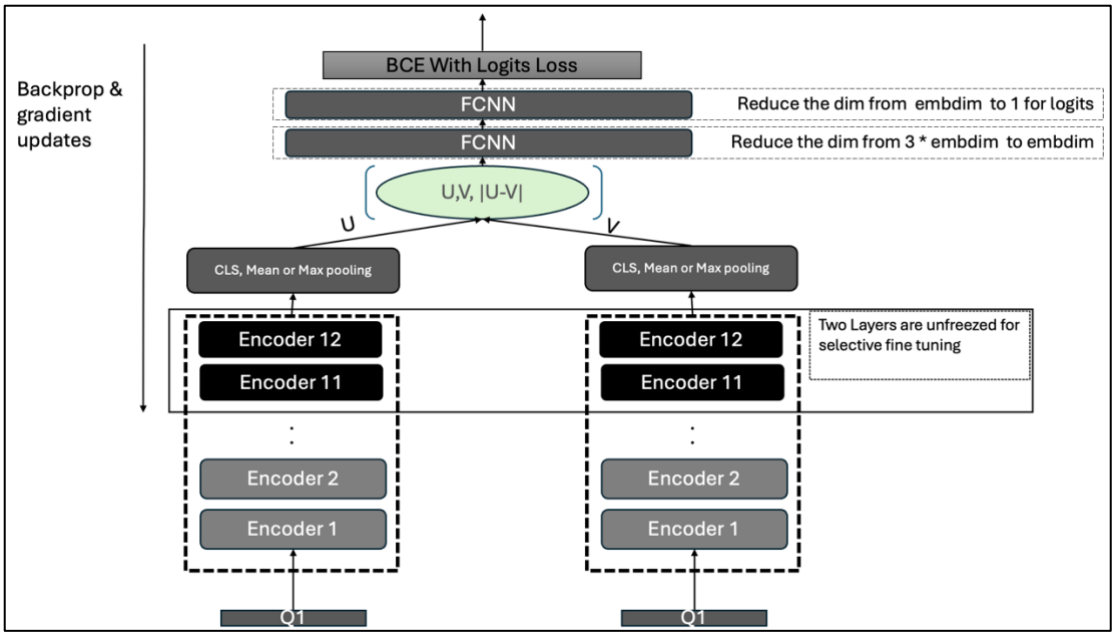


Figure 12: Siamese SBERT model architecture

The model architecture followed a Siamese design, where two identical SBERT encoder networks processed each question (q1 and q2) independently. These encoders were initialized with pre-trained weights from SBERT *(sentence-transformers/paraphrase-MiniLM-L6-v2)*. To adapt SBERT to this task, I selectively unfroze the last two encoder layers for fine-tuning. This step allowed the model to refine its representations specifically for the question similarity task while avoiding overfitting by keeping earlier layers frozen.

For each input question, the SBERT encoder produced contextual embeddings. I applied all of three pooling strategies—**mean pooling, max pooling, or CLS** token pooling—to generate a fixed-length vector representation for each question and found that mean pooling was the best among them all. These vectors were denoted as u for q1 and v for q2. The concatenated representation combined three components: u, v, and the absolute difference |u - v|. At the outset, I found that excluding the |u - v| term deteriorated the results compared to when it was included. I therefore proceeded with the latter; the inclusion of |u - v| explicitly highlighted the differences between the two embeddings, enabling the model to learn features that were indicative of semantic dissimilarity.

This concatenated vector, with a dimensionality of **3 * embedding_dim**, was passed through two fully connected layers (FCNNs). The first fully connected layer

reduced the dimensionality from **3 * embedding_dim** (i.e., 1154) to **embedding_dim** (i.e., 384) using ReLU activation and dropout for regularization. The second FCNN further reduced the output to a single logit, representing the predicted similarity between the two questions.

I trained the model for 10 epochs using a batch size of 32, balancing computational efficiency with sufficient updates per epoch. The training process included checkpointing, where the model's state was saved whenever a new best validation loss was achieved. This allowed me to resume training or evaluate the best-performing model at any point.

The **SiameseQuestionsDataset** class preprocesses question pairs (q1 and q2) by tokenizing them separately with padding, truncation, and special tokens, ensuring compatibility with the SBERT tokenizer. It returns tokenized inputs (**`input_ids`**, **`attention_mask`**) and labels as PyTorch tensors in a structured format, enabling efficient batching and training through PyTorch's **DataLoader**.

In this model, fine-tuning is performed by selectively unfreezing the last two layers the SBERT encoder while keeping the rest of the layers frozen. This allows the model to adapt its high-level representations to the specific task of question pair similarity without overfitting or losing the pre-trained semantic embeddings. Gradients are computed and updated only for the unfreezed layers, enabling task-specific learning while retaining the pre-trained knowledge in the frozen layers:

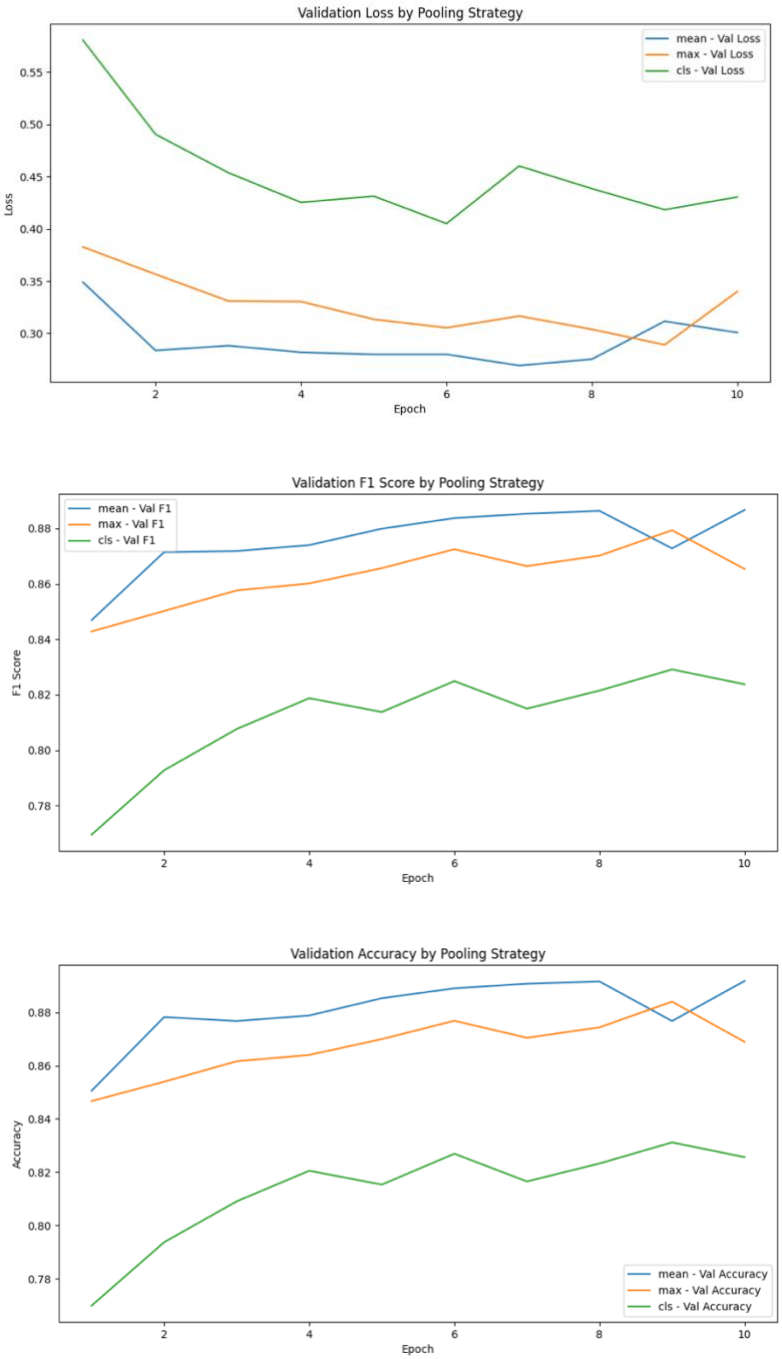
```python
class SBERTSiamese(nn.Module):
    def __init__(self, sbert_model_name='sentence-transformers/paraphrase-MiniLM-L6-v2', pooling_strategy='mean', fine_tune_layers=2):
        super(SBERTSiamese, self).__init__()
        self.sbert = AutoModel.from_pretrained(sbert_model_name)
        self.pooling_strategy = pooling_strategy.lower()
        self.hidden_dim = self.sbert.config.hidden_size

        # Freeze all layers initially
        for param in self.sbert.parameters():
            param.requires_grad = False

        # Unfreeze the last `fine_tune_layers` layers
        if fine_tune_layers > 0:
            for layer in self.sbert.encoder.layer[-fine_tune_layers:]:
                for param in layer.parameters():
                    param.requires_grad = True

        # Classification layer
        self.classifier = nn.Sequential(
            nn.Linear(self.hidden_dim * 3, self.hidden_dim),
            nn.ReLU(),
            nn.Dropout(0.1),
            nn.Linear(self.hidden_dim, 1)  # Output a single logit
        )

    def pool(self, token_embeddings, attention_mask):
        if self.pooling_strategy == 'mean':
            input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
            sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, dim=1)
            sum_mask = torch.clamp(input_mask_expanded.s                    min=1e-9)
            return sum_embeddings / sum_mask
        elif self.pooling_strategy == 'max':
            input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
            token_embeddings[input_mask_expanded == 0] = -1e9  # Set padding tokens to a large negative value
            return torch.max(token_embeddings, dim=1).values
        elif self.pooling_strategy == 'cls':
            return token_embeddings[:, 0]  # Take the [CLS] token
        else:
            raise ValueError("Invalid pooling strategy. Choose from 'mean', 'max', or 'cls'.")

    def encode(self, input_ids, attention_mask):
        outputs = self.sbert(input_ids=input_ids, attention_mask=attention_mask)
        return self.pool(outputs.last_hidden_state, attention_mask)

    def forward(self, q1_input_ids, q1_attention_mask, q2_input_ids, q2_attention_mask):
        q1_embeddings = self.encode(q1_input_ids, q1_attention_mask)
        q2_embeddings = self.encode(q2_input_ids, q2_attention_mask)

        # Concatenate u, v, |u-v|
        combined = torch.cat([q1_embeddings, q2_embeddings, torch.abs(q1_embeddings - q2_embeddings)], dim=1)

        # Pass through classifier
        logits = self.classifier(combined).squeeze(1)
        return logits
```

Outline

Figure 13: Model definition code

3.5.2 Results

The three plots below illustrate the results obtained from the model:



Validation Loss

Mean pooling achieved the lowest final validation loss at 0.30, followed by max pooling at 0.35, and CLS token pooling with the highest loss at 0.42. These results highlight mean pooling's superior ability to minimize loss and align embeddings effectively.

Validation F1 Score

The final validation F1 scores were 0.88 for mean pooling, 0.86 for max pooling, and 0.80 for CLS token pooling. Mean pooling outperformed the other strategies, achieving the best balance between precision and recall.

Validation Accuracy

Mean pooling achieved the highest final validation accuracy at 88%, followed by max pooling at 86%, and CLS token pooling at 83%. These results mirror the trends in F1 scores, confirming mean pooling's effectiveness for the task.

Insights

I noticed that mean embeddings are producing the best results, which is why I trained it for 20 epochs, which led to the model getting overfitted and generating the best val F1 result being 89.23% while the train F1 result being 95.04%. Hence I took results from 10 epochs only.

Across all metrics, mean pooling outperformed other strategies, achieving the best validation loss, F1 score, and accuracy. The inclusion of the absolute difference $|u-v|$ $|u - v|$ significantly improved model performance by emphasizing semantic dissimilarities. Selective fine-tuning of the last two SBERT layers further enhanced task-specific learning without overfitting or loss of pre-trained semantic embeddings.

Results

Algorithm	F1-macro
Naïve Bayes	61.82%
Logistic Regression	79.31%
Random Forest	82.04%
XGBoost	**84.10%**
GRU	38.85%
LSTM	39.93%
Siamese Bert + FCNN	**89.07%**

Table 3: End-to-end performance table

The Siamese BERT network with FCNN and mean pooling technique performed the best amongst all other methods with an F1 macro score of 89.07%.

Summary and Conclusion

Through this project, I gained significant insights into both practical and theoretical aspects of NLP model implementation. While classical models like XGBoost performed well (84.10% F1-macro), and deep learning models like LSTM/GRU surprisingly underperformed, the most valuable learnings came from implementing the Siamese BERT architecture.

Working with SBERT taught me several crucial lessons about transformer-based architectures. I discovered that architectural choices dramatically impact.

The implementation revealed that selective fine-tuning provided better results than either keeping the entire model frozen or making it fully trainable. This taught me the importance of balancing pre-trained knowledge with task-specific adaptation. The experimentation with different pooling strategies was particularly enlightening - mean pooling consistently outperformed max and CLS token pooling, achieving the best validation metrics.

I also learned about preventing overfitting in transformer models. When I initially trained for 20 epochs, the model showed signs of overfitting. Reducing to 10 epochs provided better generalization while maintaining strong performance (89.07% F1-macro). This experience emphasized the importance of careful monitoring and the need to find the right balance between model convergence and generalization.

The project ultimately demonstrated that while transformer architectures like SBERT can deliver superior results, success depends heavily on understanding and properly implementing their architectural components, along with making informed decisions about training strategies and hyperparameters.

3.6 Streamlit App

Our project was demonstrated through a streamlit app, currently hosted on the project's GitHub. The app is an interactive and accessible application that can classify Quora Question Pairs in seconds and visualize the trends from the results.

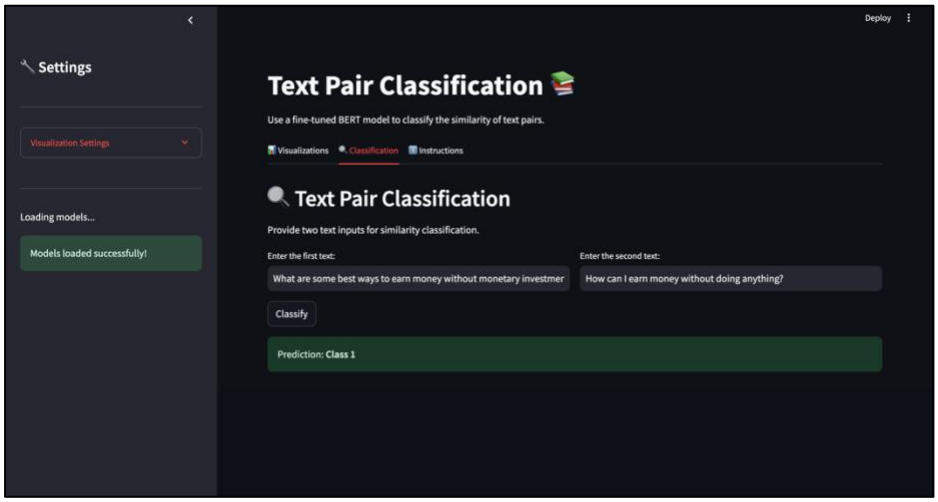


Figure 14: Streamlit App Interface

Source Code Evaluation

I referenced the dataset and dataloader code from PyTorch [5].

Total Code copied = 42

Code lines changed = 5

Original Code = (42-5)/5 = ~88%

References

[1] https://quoradata.quora.com/First-Quora-Dataset-Release-Question-Pairs

[2] Reimers, N. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *arXiv preprint arXiv:1908.10084*.

[3] Yadav, G., Sinha, P., Sinha, P., & Jha, V. (2023). AN ANALYTICAL STUDY ON THE QUESTIONS COMPARING BY USING DIFFERENT MACHINE

LEARNING MODELS WITH SPECIAL REFERENCE TO RANDOM, FOREST, XGBOOST ETC. *MIET-Journal of Engineers and Professional Management & Allied Research*, 96.

[5] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html