Statistical Methods in Artificial Intelligence

Team 2

Natural Language Understanding with the Quora Question Pairs Dataset

December 2022

Final Report

Summary

1 Introduction

The Quora dataset consists of many duplicate questions. The aim is to identify which pairs of questions are duplicates from the dataset. The experts find it highly tedious to respond to the same question twice, thus they are not inclined to do so. Users are also prevented from seeing high-quality answers due to the duplicate questions.

This problem requires Natural Language Understanding (NLU) and NLU, ultimately, helps in other Natural Language Processing (NLP) tasks. The model must be able to understand the morphological and grammatical meanings of the given sentences.

An extensive set of models is used consisting of Linear Models, Tree-based models and a range of deep neural networks. The Linear models include Logistic Regression, Support Vector Machine (SVM) using n-gram features and SVM using sentence embedding feature vectors. The Tree-based models comprise of Decision Tree, Random Forest and Gradient Boosted Classifier. The features used for tree-based models will be discussed in detail later. The neural-network models consist of a simple Continuos Bag of Words neural network, Long short-term memory (LSTM), bidirectional LSTM (BiLSTM), and an LSTM model with the idea of Attention.

2 Dataset Analysis

The dataset is downloaded from the kaggle competition, Quora Question Pairs. The Quora Question Pairs dataset consists of around 404 thousand question pairs, out of which a few have null values. Each sample row of the dataset has the following fields:

- id: ID of the pair of questions
- qid1: ID of question 1
- qid2: ID of question 2
- question1: Full text of question 1

- question2: Full text of question 2
- is_duplicate: a value indicating if the pair of questions are duplicates are not. 1 indicates duplicate questions and 0 indicate not duplicate.

Some of the texts of these questions had non-ASCII characters and many of them were repeated across multiple pairs. The ratio of duplicate question pairs to non-duplicate question pairs is 0.3692.

The dataset was split into 70:20:10 for training, validation and test sets respectively. Although the paper mentions two types of splits: Blind and Disjoint, all the models have been fitted, validated and tested on Blind split. This is because the Blind dataset split serves more of a paradigmatic to real-world applications as a question is more likely to be asked more than once and the vocabularies of the sets in the Disjoint set would be completely different. The ratio of duplicate to non-duplicate question pairs (0.3692) is maintained across all the training, validation and test sets.

For preprocessing, the text is converted to its lowercase form, non-ASCII characters are removed. Punctuation are removed for the tree-based models and stop-words are also removed after tokenization. Stemming is also done in linear models.

3 Linear Models

3.1 Logistic Regression

3.1.1 Preprocessing

Preprocessing is done by converting to lowercase form. Non-ASCII characters are removed. Tokenization is done using `nltk.word_tokenize`. Stemming is done using `nltk.stem.porter.PorterStemmer` after removing the stopwords in `nltk.corpus.stopwords.words('english')`.

```
stemmer = PorterStemmer()
def tokenize(text: str) -> list[str]:
    tokens = nltk.word_tokenize(re.sub(r'[^\x00-\x7F]+',' ', text))
    tokens = [stemmer.stem(w) for w in tokens if stemmer.stem(w) not in stopwords]
    return tokens
```

3.1.2 Features

Unigram, Bigram and Trigram features are extracted after the preprocessing and tokenization step using `sklearn.feature_extraction.text.CountVectorizer`. The `tokenize` function is passed as an argument in the `tokenizer` parameter of the `CountVectorizer` constructor. The `ngram_range` parameter specifies which n_gram features are to be extracted.

```
unigramVectorizer = CountVectorizer(
                        analyzer='word',
                        ngram_range=(1,1),
                        lowercase=True,
                        tokenizer=tokenize
                    )
bigramVectorizer = CountVectorizer(
                        analyzer='word',
                        ngram_range=(1,2),
                        lowercase=True,
                        tokenizer=tokenize
                    )
trigramVectorizer = CountVectorizer(
                        analyzer='word',
                        ngram_range=(1,3),
                        lowercase=True,
                        tokenizer=tokenize
                    )
```

3.1.3 Model performance

The `sklearn.linear_model.SGDClassifier` with the parameter `loss='log_loss'` is used as a Logistic Regression classifier to test three models with the following set of features:

- Unigram features
- Unigram and Bigram features

- Unigram, Bigram and Trigram features

L2 penalty is used as the regularizer and optimal learning rate is used, which is, $\eta_{(t)} = \frac{1}{(\alpha(t+t_0))}$. The value of α used is 0.00001.

```
unigramLogisticRegressionModel = SGDClassifier(
                                loss='log_loss',
                                penalty='l2',
                                alpha=0.00001,
                                max_iter=1000,
                                n_iter_no_change=20,
                                learning_rate='optimal',
                                n_jobs=-1,
                                random_state=42)
unigramLogisticRegressionModel.fit(X_train_unigram, y_train)
y_pred_unigram_logistic = unigramLogisticRegressionModel.predict(X_test_unigram)
```

The above process is done for the set of unigram and bigram features, and the set of unigram, bigram and trigram features. Maximum accuracy is observed in the trigram feature Logistic Regression model.

Feature set	**Accuracy**	**F1-score**
Unigram	74.18	63.11
Unigram and Bigram	79.62	70.66
Unigram, Bigram and Trigram	81.15	71.48

The Trigram model was tuned further to identify the set of hyperparameters which would provide the best performance. `sklearn.model_selection.GridSearchCV` and `sklearn.model_selection.StratifiedKFold` was used to do an exhaustive search over the parameters of `alpha` and `n_iter_no_change` (the number of iterations without improvement to wait before stopping fitting).

```
parameters = dict({
                'alpha':[0.01, 0.001, 0.0001, 0.00001, 0.000001],
                'n_iter_no_change': [5, 10, 15, 20]
            })
```

```
cv_stratified_splitter = StratifiedKFold(n_splits=5)
grid_search = GridSearchCV(trigramLogisticRegressor,
                           parameters,
                           cv=cv_stratified_splitter,
                           scoring=['accuracy', 'f1'],
                           n_jobs=-1,
                           refit='f1')
```

The best set of hyperparameters for trigram Logistic Regression model is `alpha`= 0.000001 and `n_iter_no_change`= 20.

Logistic Regression Trigram Model, Tuned	
Accuracy	F1-Score
80.31	72.37

3.2 Support Vector Machines

Preprocessing, tokenization and feature-extraction is done as the same in Logistic Regression models. Unigram, Bigram and Trigram features are extracted and SVM models are fitted on the three sets of features.

3.2.1 Model performance

The `sklearn.svm.SVC` is used for generic kernel SVM and `sklearn.svm.LinearSVC` is used for Linear SVM. According to the paper, the default parameters to be used were $C = 1.0$ and `kernel='linear'`. Therefore, Linear SVM was fitted on the three set of features with $C = 1.0$.

Model	**Accuracy**	**F1-Score**
Linear SVM, Unigram	73.39	64.13
Linear SVM, Bigram	77.65	69.94
Linear SVM, Trigram	79.26	71.32

Linear and rbf kernels were fitted with different values of C. The models were tuned using the `sklearn.svm.SVC`. The best set of hyperparameters for trigram SVM model is `C`= 0.000001 and `kernel='linear'`

SVM Trigram Linear Model, Tuned	
Accuracy	F1-Score
80.11	71.29

3.3 Sentence Embeddings

3.3.1 Preprocessing

Preprocessing is done as the same in Logistic Regression and SVM Models. The `CountVectorizer` was calling the custom tokenize function in the above models. However, the preprocessing to find sentence embeddings had to be done separately.

```
def preprocessAndTokenizeForGlove(text: str) -> list[str]:
    text = re.sub(r'[^\x00-\x7F]+',' ', text.lower())
    text = text.translate(str.maketrans('', '', string.punctuation))
    tokens = nltk.word_tokenize(text)
    tokens = [stemmer.stem(w) for w in tokens if stemmer.stem(w) not in stopwords]
    return tokens
```

3.3.2 Features

Sentence embeddings of a question were found by simply summing the word-embeddings of the tokens that constitute the sentence. The GloVe.6B.50.txt to extract 50-dimensional feature vector for each token in the sentence. The extraction of features for the pair of questions was done in the two following ways:

1. *Plain sentence embedding*: A feature vector of 100-dimension formed by horizontally stacking the 50-dimension embeddings of both the questions.
2. *Distance measure between vectors*: A feature vector of 7 distance measures between the two embedding vectors of the questions.

The 7 distance measures were as follows:

1. Bray Curtis distance:

$$d_{BrayCurtis} = \frac{1}{2} \sum_{i=1}^{50} \frac{|q_1[i] - q_2[i]|}{|q_1[i] + q_2[i]|} \tag{1}$$

2. Canberra distance:

$$d_{Canberra} = \sum_{i=1}^{50} \frac{|q_1[i] - q_2[i]|}{|q_1[i]| + |q_2[i]|} \tag{2}$$

3. Chebyshev distance:

$$d_{Chebyshev} = max(|q_1[i] - q_2[i]|) \tag{3}$$

4. City block distance:

$$d_{Cityblock} = \sum_{i=1}^{50} |q_1[i] - q_2[i]| \tag{4}$$

5. Correlation distance:

$$d_{Correlation} = 1 - \frac{\sum_{i=1}^{50}(q_1[i] - \bar{q_1})(q_2[i] - \bar{q_2})}{\sqrt{\sum_{i=1}^{50}(q_1[i] - \bar{q_1})^2}\sqrt{\sum_{i=1}^{50}(q_2[i] - \bar{q_2})^2}} \tag{5}$$

6. Cosine distance:

$$d_{Cosine} = 1 - \frac{\sum_{i=1}^{50} q_1[i]q_2[i]}{\sqrt{\sum_{i=1}^{50} q_1[i]^2}\sqrt{\sum_{i=1}^{50} q_2[i]^2}} \tag{6}$$

7. Euclidean distance:

$$d_{Euclidean} = \sqrt{\sum_{i=1}^{50}(q_1[i] - q_2[i])^2} \tag{7}$$

All of these distance measures were calculated using `scipy.spatial.distance`.

3.3.3 Model performance

The parameters were `C=1.0` and two kernels: linear and rbf.

For the plain sentence embedding features,

Model	**Accuracy**	**F1-Score**
Linear SVM	63.85	61.93
Rbf SVM	77.38	69.89

For the distance measure features,

Model	**Accuracy**	**F1-Score**
Linear SVM	63.91	62.53
Rbf SVM	67.94	68.34

4 Tree Based Models

Tree based models often give excellent results and are frequently used in practice. In order to build a tree based models we require two necessary parameters - Feature Engineering and Hyperparameters.

4.1 Feature Engineering

We need to find individual features as models based on decision trees make splitting decisions based on individual features which are sorted to decide a split. There were mainly two issues which we faced. One - For the tree based models we generally make the splitting decisions on individual features but with sparse encoding it is likely that a majority of data points will have no value so we can't assign proper partitions to those data points. Second - Dense representation of data took up too much memory. To resolve these problems we designed a small number of features that could be extracted from the examples in the dataset. To design the final set of features (Misc), we examined the errors incurred by the model trained on the previous 6 feature sets. We take the following features :

1. (L) These are Length based features. It consists of length for question 1 (l_1), question 2 (l_2), difference of lengths ($l_1 - l_2$) and ratio of lengths (l_1/l_2)
2. (LC) It includes the number of common lowercased words, i.e. the count and count/length of longest sentence.
3. ($LCXS$)) It includes the number of common lowercased words, excluding stop-words: count, count/length of longest sentence.
4. (LW) It includes the cases having same last words.
5. (CAP) it includes the number of common capitalized words i.e. count and count/length of longest sentence.
6. (PRE) It consists of the mumber of common prefixes, and for prefixes of length 3–6: count, count/length of longest sentence.
7. (M) Misc it checks that whether questions 1, 2, and both contain "not", both contain the same digit, and number of common lowercased words after stemming.

4.2 Hyperparameters

We tune the different parameter settings for random forests and the gradient boosting trees. These parameter settings include maximum depth, minimum number of data-points in a leaf and the number of estimators. The tree models we used had the following parameter settings :

1. Decision Trees : Max Depth = 10 , min samples per leaf = 5
2. Random Forest : Max Depth = None, min samples per leaf= 5, num estimators= 50
3. Gradient Boosted Tree : Max Depth = 4, num estimators = 500

4.3 Accuracies and F-scores

Features	Num-Features	Accuracy(%)	F-score
L	4	64.32	23.28
L, LC	6	69.61	58.35
L, LC, LCXS	8	69.60	58.24
L, LC, LCXS, LW	9	72.33	62.01
L, LC, LCXS, LW, CAP	11	72.34	62.11
L, LC, LCXS, LW, CAP, PRE	19	72.54	62.93
L, LC, LCXS, LW, CAP, PRE, M	25	74.01	65.13

5 Neural Network Models

We have trained and tested the accuracy across 4 different Neural Network models, viz:

1. Continuous Bag of Words (CBOW)
2. Long Short Term Memory RNN (LSTM)
3. Bidirectional LSTM (BiLSTM)
4. LSTM with word-by-word attention

Since the dataset is quite large, we have trained the dataset in batches.
To ensure modularity, we have made a parent neural network class and a subclass for each neural network.

5.1 Prepossessing

Since we are using GloVe vectors, it is necessary to convert all the characters to lowercase. Additionally, we have also removed the extra spaces, Non-ASCII character and stopwords. We have also removed the punctuations since we want to deal with word-by-word semantic similarity. We have tokenized the word using ‘keras.preprocessing.text.Tokenizer‘.

5.2 Feature Extraction

The tokenizer sorts all the words in descending order based on their frequency. Based on this, each word is assigned an index. We now import the 300-dimensional GloVe dataset, and store the embeddings in a dictionary. Using the word indices of tokenization, we build an embedding matrix, where each row corresponds to the feature vector of the corresponding word in tokenization index. We pass the word sequence with a desired maximum length to the neural network models. We pad the sequence in case the sequence length is greater than or lesser than the specified maximum length.

5.3 Models

5.3.1 Continuous Bag of Words

Let S_1 and S_2 be the sentence embeddings. Then we have to pass the concatenation of $S_1 + S_2$, $S_1 - S_2$ and $S1 * S2$ as an input to the neural network (note that $*$ refers to block-wise convolution or pair wise multiplication). We tried two different methods for this : by precomputing the concatenated vectors and passing it as input, and in the other method, we made separate layers for concatenation. The former method was computationally much faster.

```
def fit(self,xtrain, xval, ytrain, yval):
    self.model.add(Dense(300, input_shape = (900,), activation = "sigmoid"))
    self.model.add(Dropout(0.1))
    self.model.add(Dense(200, activation = "sigmoid"))
    self.model.add(Dropout(0.1))
    self.model.add(Dense(100, activation = "sigmoid"))
    self.model.add(Dropout(0.1))
    self.model.add(Dense(2, activation = "softmax"))
```

The CBOW model consists of a three-layered MLP followed by a sigmoid activation function.

5.3.2 Long Short Term Memory RNN (LSTM)

For the LSTM model, we use the final state of the LSTM layer as the vector representation. LSTM consists of a single recurrent Tanh layer. The activation function is tanh and the recurrent activation function is sigmoid.

```
def train_model(self):
        inp1 = Input(shape = (128,))
        inp2 = Input(shape = (128,))
        emb1 = Embedding(output_dim=300, weights = [self.embedding_matrix],
        trainable = False, input_dim=self.vocab_size, input_length=128)(inp1)
        emb2 = Embedding(output_dim=300, weights = [self.embedding_matrix],
        trainable = False, input_dim=self.vocab_size, input_length=128)(inp2)
        concat = Concatenate(axis = -1)([emb1 + emb2, emb1 - emb2, emb1 * emb2])
```

```
lstm = LSTM(150, return_sequences=False, dropout=0.1,
return_state=True)(concat)
out = Dense(2, activation = "softmax")(lstm[2])
```

We do the training in batches of 8 training samples and 4 validation samples. Note that LSTM based models take a lot of time for training, so we have taken 1/4th of the dataset for training and trained it on 4 epochs.

5.3.3 BiLSTM

The BiLSTM model is quite similar to an LSTM model, except that the layer is bidirectional and we use we used average of all the states of the BiLSTM for the vector representation

```
concat = Concatenate(axis = -1)([emb1 + emb2, emb1 - emb2, emb1 * emb2])
out = tf.keras.layers.Bidirectional(
tf.keras.layers.LSTM(150, kernel_regularizer='l2', dropout=0.1,
return_sequences=True))(concat)
out = tf.keras.backend.mean(out, axis=1, keepdims=False)
```

5.3.4 LSTM with word-by-word Attention

In LSTM models with attention, we performed attention by taking the hidden states computed by LSTM. This is done by taking the dot product of those outputs. These dot products are then passed through a softmax layer, which we then use to normalize the LSTM outputs.

```
attention = dot([lstm1[0], lstm2[0]], axes=[2, 2])
u_norm = Softmax(axis=-1)(attention)
v_norm = Softmax(axis=1)(attention)
u = dot([u_norm, lstm1[0]], axes=[1, 1])
v = dot([v_norm, lstm2[0]], axes=[1, 1])
```

After this, we do a weighted sum to get the new vector representation. They are then passed through a tanh layer. We concatenate them, and pass the concatenated output through a final softmax layer.

```
WU_bar = Dense(150)(u[:, -1, :])
WV_bar = Dense(150)(v[:, -1, :])
```

```
VU = Dense(150)(lstm1[0][:, -1, :])
VV = Dense(150)(lstm2[0][:, -1, :])

ufinal = Add()([WU_bar, VU])
vfinal = Add()([WV_bar, VV])

ufinal = Activation('tanh')(ufinal)
vfinal = Activation('tanh')(vfinal)

concat = Concatenate(axis = -1)([ufinal, vfinal])
out = Dense(2, activation = "softmax")(concat)
self.model = Model(inputs = [inp1, inp2], outputs = out)
```

5.4 Experimental Setup

We used 300-dimenstional GloVe embeddings to initialize the word embeddings. We used L2 regularization with parameter value 0.001 in all models except CBOW. We used a Dropout rate of 0.1 in all models.

5.5 Model Performance

The neural network models outperformed the linear and tree based models. CBOW seemed to outperform all the models owing to the semantic representation of feature vectors.

Neural Model	**Accuracy**	**F1-score**
Continuous Bag of Words(CBOW)	80.65	83.36
Long Short Term Memory(LSTM)	78.41	73.57
Bidirectional LSTM(BiLSTM)	79.09	76.37
LSTM with Attention	77.51	70.64

6 Final results and Conclusion

LSTM or RNN models in general are considered to be better than CBOW. However, in this task the syntactic information in the sentences is far less important than simple word-by-word semantic similarity. In tree-based models, removing the punctuations showed better results. In linear n-gram models, Trigrams seemed to outperform others.