FINAL PROJECT REPORT

M22CS7.403 Statistical Methods in AI Monsoon 2022

Finding Duplicate Questions in the Quora Dataset

[GITHUB LINK]

Prepared By

Team -14 Members :

Sarthak Aggarwal	2020101008
Ayan Agarwal	2020101034
Tarun	2020102056
Eshan Gupta	2019102044

ABSTRACT

Our goal for this project was to determine whether two Quora questions are similar or not using various traditional ML and DL models with an extensive feature engineering. Till now, we've extracted **TF-IDF** and **N-Grams** features from pairs of questions and used them for training different variants of **Logistic Regression** as well as **SVM**. We tried to replicate the results as mentioned in the assigned paper and analysed the optimal hyperparameters found through **GridSearch Cross-validation**. Further, manual features are generated for **Tree-based** models upon which we'll be training our models like **Decision and Random Forest**. So far we're testing with ML models which, according to paper, serves as baseline models for the DL models like **CBOW and LSTM**. We have treid to cover up all the model performances, comparison, analysis as mentioned in the paper along with a final touch on Transformer based models in our final evaluation report.

Project Description

The aim is to determine whether the two questions are duplicates of each other, i.e., whether they reflect the same meaning or not, using the Quora question pair dataset. As a result, this task is essentially a binary classification problem, with a 0/1 response dependent on whether or not the questions are comparable. This project is partially a replication of the cited paper.

Data Overview

The dataset that is presently accessible has been found to be substantially unbalanced. 255,027 (63.08 %) of the 404,290 question pairings have a negative (0) label, while 149,263 (36.92 %) have a positive (1) label. The question pairs, their corresponding ids, sample id, and the accompanying label are shown in the sample from the dataset below

	id	qid1	qid2	question1	question2	is_duplicate
0	0	1	2	What is the step by step guide to invest in sh...	What is the step by step guide to invest in sh...	0
1	1	3	4	What is the story of Kohinoor (Koh-i-Noor) Dia...	What would happen if the Indian government sto...	0
2	2	5	6	How can I increase the speed of my internet co...	How can Internet speed be increased by hacking...	0
3	3	7	8	Why am I mentally very lonely? How can I solve...	Find the remainder when [math]23^{24}[/math] i...	0
4	4	9	10	Which one dissolve in water quikly sugar, salt...	Which fish would survive in salt water?	0

Figure 1: A sample from the available dataset

Observing the dataset, the dataset is being splitted into three sections namely train, validation and test with the ratio of 70:10:20 which is similar to the way mentioned by the authors in the referenced research papers.
The following plots clearly depict the variou essential components of the whole dataset as well as the factors used for further usage in the following mentioned sections.

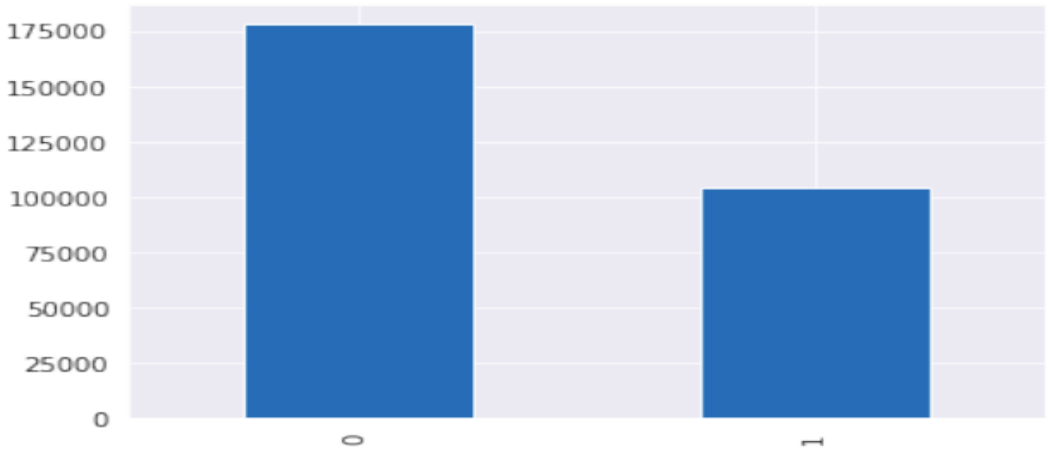
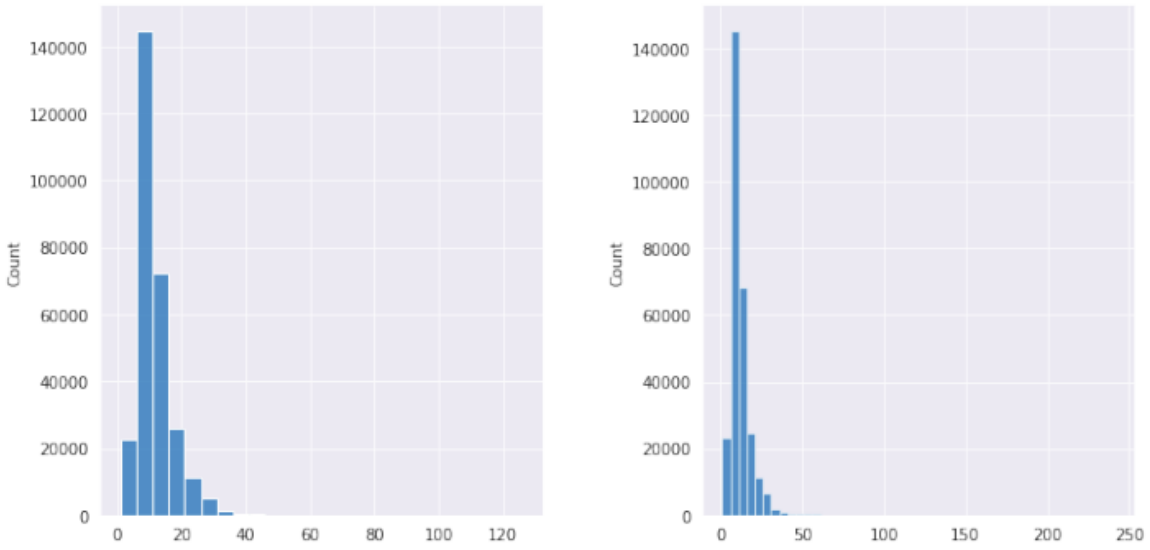


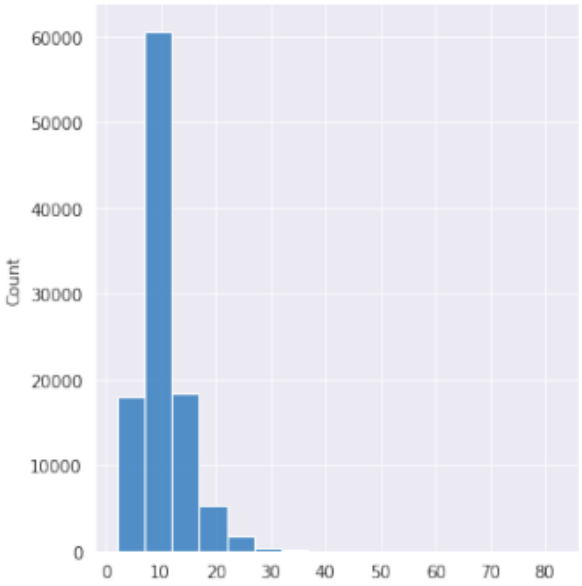
Figure 2: Class Distribution



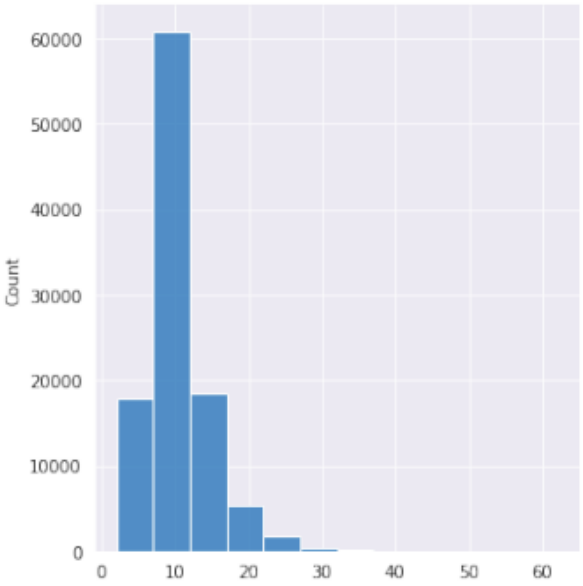
(a) Frequency of Question-1 Lengths

(b) Frequency of Question-2 Lengths

Figure 3: Frequency of Questions Lengths

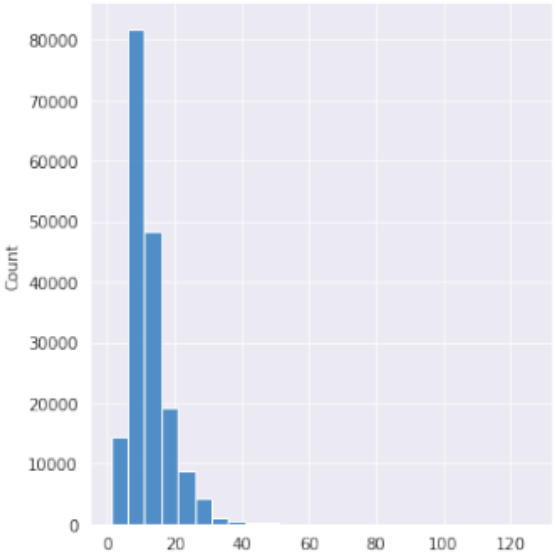


(a) Frequency of Duplicate Question-1

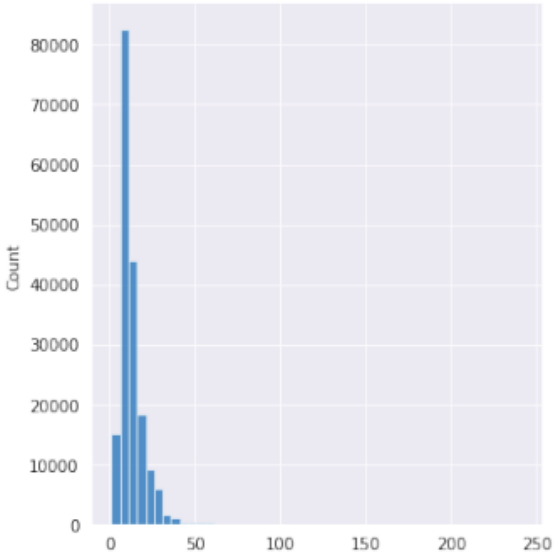


(b) Frequency of Duplicate Question-2

Figure 4: Frequency of Duplicate Questions



(a) Frequency of Non-Duplicate Question-1



(b) Frequency of Non-Duplicate Question-2

Figure 5: Frequency of Non-Duplicate Questions

3 Linear Models

Linear Models are trained on N-gram features like **Unigram, Bigrams, Trigrams**. A N-Gram is a series of N words in NLP. If n = 1 it is called **unigram.** If Number of words are 2 then it is called **Bigram**. If the count is three then it is called **trigram**.

3.1 Logistic Regression

- **Preprocessing:**

  ->The training file is read using read_csv() function**.**

  ->The train.dropna() function is used to remove the missing values in the data set.

```
data = pd.read_csv('train.csv')
data = data.dropna(how="any").reset_index(drop=True)
```

  -> The recommended preprocessing for linear models in the paper is to remove non-ASCII characters and tokenize using the nltk tokenizer but since we are using the **CountVectorizer**() function for feature extraction, which tokenizes and does the preprocessing internally, we are skipping this step.

- **Feature Extraction:**

  N-grams: sequences of N words. N-gram counting involves counting how often word-sequences occur in both the sentences.

  - From each data-point, the counts of each N-gram in each question are to be extracted.
  - N-gram counts extracted using the **CountVectorizer**() function are stored in a dictionary representation.
  - The features are then converted to SciPy's sparse CSR matrix format

  Feature extraction for unigram models (N=1):

```
cnt = CountVectorizer(analyzer='word', ngram_range=(1,1), lowercase=True)
cnt.fit(pd.concat((data['question1'],data['question2'])))
q1 = data['question1']
q2 = data['question2']
q1 = cnt.transform(q1)
q2 = cnt.transform(q2)
```

  - 86152 features were obtained for unigram model from the datset used.
  - Then these feature dataset is split into validation, test and train dataset in the ration of 1:2:7 and then trained the models.
  - Blind Split was done to maintain the balance between isduplicate true and false.

Features extraction for Bigram models (N = 2):

- Similar procedures and splits are followed.

```
data2 = pd.read_csv('./train.csv')
data2 = data2.dropna(how="any").reset_index(drop=True)
cnt = CountVectorizer(analyzer='word', ngram_range=(1,2), lowercase=True)
cnt.fit(pd.concat((data2['question1'],data2['question2'])))
q1 = data2['question1']
q2 = data2['question2']
q1 = cnt.transform(q1)
q2 = cnt.transform(q2)
```

- 1250860 features were obtained from the given dataset for the biagram models.

Features extraction for Bigram models (N = 3):

- Similar procedure and splits are followed.
- 3858251 features were obtained for trigram models from the given dataset.

```
data3 = pd.read_csv('./train.csv')
data3 = data2.dropna(how="any").reset_index(drop=True)
cnt = CountVectorizer(analyzer='word', ngram_range=(1,3), lowercase=True)
cnt.fit(pd.concat((data3['question1'],data3['question2'])))
q1 = data3['question1']
q2 = data3['question2']
q1 = cnt.transform(q1)
q2 = cnt.transform(q2)
xx = scipy.sparse.hstack((q1,q2))
yy = data3['is_duplicate']
```

- **Training the models:**
  - We used scikit-learn's implementation with a penalty = l2, alpha = 0.0001, and the number of iterations = 20 initially.
- **Logistic regression with Unigrams:**
  - Initially our model gives an accuracy of 74.83% and F score of 63.62% against the test dataset.

```
uni = SGDClassifier(penalty='l2',alpha=0.00001, n_iter_no_change=20)
uni.fit(x_train_u,y_train_u)
y_pred = uni.predict(x_val_u)
acc = accuracy_score(y_val_u, y_pred)
f1 = f1_score(y_val_u,y_pred)
print('accuracy',acc)
print('f score',f1)
```

- Accuracy for validation set = 75.3556
- F score for validation set = 62.5875
- Accuracy for test set = 74.8348
- F score for test set = 63.627913

- **Logistic regression with Bigrams:**
  - Initially with the said parameters our model gives an accuracy of 77.92% and F score of 69.90 against the test set

```
bi = SGDClassifier(penalty='l2',alpha=0.00001, n_iter_no_change=20)
bi.fit(x_train_b,y_train_b)
y_pred = bi.predict(x_val_b)
acc = accuracy_score(y_val_b, y_pred)
f1 = f1_score(y_val_b,y_pred)
print('accuracy',acc)
print('f score',f1)
```

  - Accuracy for validation set = 77.9737
  - F score for validation set = 70.0410
  - Accuracy for test set = 77.929209
  - F score for test set =69.906579
- **Logistic regression with Trigrams:**
  - Initially with the said parameters our model gives an accuracy of 79.47% and F score of 71.54 against the test set

```
tri = SGDClassifier(penalty='l2',alpha=0.00001, n_iter_no_change=20)
tri.fit(x_train_t,y_train_t)
y_pred = tri.predict(x_test_t)
acc = accuracy_score(y_test_t, y_pred)
print("Accuracy",acc)
f1 = f1_score(y_test_t,y_pred)
print("F1 score",f1)
```

  - Accuracy for validation set = 80.1677014
  - F score for validation set = 70.051919

- Accuracy for test set = 79.9302
- F score for test set =71.473773

- **Hyper - parameter Tuning:**
  - Clearly, the trigram model is the best among the three.
  - $\alpha = 0.0001$ gives an accuracy of 80.13% and F score = 70.32. Highest F-score=71.25 for $\alpha = 0.00001$.
  - Tuning our best model by varying the regularization parameter and the number of iterations and testing on the validation dataset, the results are as follows.

alpha	Accuracy	F score
0.1	63.825 %	3.624
0.01	70.397 %	39.1931
0.001	75.125 %	56.65
0.0001	80.135 %	70.235
0.00001	79.98 %	71.25
0.000001	79.54 %	70.89

Effect of varying the regularization parameter on the Trigram linear model

No of iteration	Accuracy	F score
5	80.165 %	70.10
10	80.111 %	70.25
15	80.27 %	70.17
20	80.235 %	70.20
25	80.254 %	70.21
30	80.255 %	70.06

Effect of varying the number of iterations on the Trigram linear model.

- **Logistic regression with Trigrams and hyperparameter tuning (n_iter = 15, alpha = 0.0001):**

```
tri_t = SGDClassifier(penalty='l2',alpha=0.0001, n_iter_no_change=15)
tri_t.fit(x_train_t,y_train_t)
y_pred = tri_t.predict(x_test_t)
acc = accuracy_score(y_test_t, y_pred)
f1 = f1_score(y_test_t,y_pred)
print('accuracy',acc)
print('f score',f1)
```

- Accuracy for test set = 80.18254 %
- F score for test set = 70.12677

- **Observations:**
  - The trigram model outperforms the unigram and bigram models.
  - After hyperparameter tuning, the model gives slightly better accuracy of 80.18 on the test data, but a lower F-score.

3.2 Support Vector Machines

- **Preprocessing and Feature Extraction:**
  - Since the preprocessing steps and the features are the same in all linear models, we used the previously obtained data.
- **Training the model:**
  - The data is normalized to reduce the fitting time since even linear SVM implementations have very high fitting times.
  - Scikit-learn's LinearSVC implementation is used for training.
  - Initially, C(penalty parameter) is set to 1.0.

- **SVM with Unigrams:**

```
from sklearn.svm import LinearSVC
from sklearn.preprocessing import normalize

x_test_u1 = normalize(x_test_u, norm='l1', axis=1)
x_train_u1 = normalize(x_train_u, norm='l1', axis=1)

uni = LinearSVC(C=1)
uni.fit(x_train_u1,y_train_u)
y_pred = uni.predict(x_test_u1)
acc = accuracy_score(y_test_u, y_pred)
f1 = f1_score(y_test_u,y_pred)
print('accuracy',acc)
print('f score',f1)
```

  - Accuracy for test set = 75.62 %
  - F score for test set = 64.93 %

- **SVM with Bigrams:**

```
x_test_b1 = normalize(x_test_b, norm='l1', axis=1)
x_train_b1 = normalize(x_train_b, norm='l1', axis=1)

uni = LinearSVC(C=1, max_iter=1000)
uni.fit(x_train_b1,y_train_b)
y_pred = uni.predict(x_test_b1)
acc = accuracy_score(y_test_b, y_pred)
f1 = f1_score(y_test_b,y_pred)
print('accuracy',acc)
print('f score',f1)
```

  - Accuracy for test set = 75.57 %
  - F score for test set = 60.45 %

- **SVM with Trigrams:**

```
x_val_t1 = normalize(x_val_t, norm='l1', axis=1)
x_train_t1 = normalize(x_train_t, norm='l1', axis=1)

tri = LinearSVC(C=50)
tri.fit(x_train_t1,y_train_t)
y_pred = tri.predict(x_val_t1)
acc = accuracy_score(y_val_t, y_pred)
f1 = f1_score(y_val_t,y_pred)
print('accuracy',acc)
print('f score',f1)
```

  - Accuracy for test set = 79.89 %
  - F score for test set = 70.78 %

- **Hyperparameter Tuning:**
  - The accuracies are indicated for various values of C. C = 50 for a trigram model seems to give the highest accuracy on the validation set.

C	Unigram Model	Bigram Model	Trigram Model
0.005	65.6%	63.9%	63.26%
0.01	67.6%	65.2%	64.2%
0.1	72.1%	71.0%	69..8%
0.5	74.1%	74.14%	73.4%
1.0	74.7%	75.29%	74.7%
10	75.7%	78.75%	79.6%
50	75.6%	79.8%	81.1%

- **Observations:**
  - After tuning, the trigram model gives an accuracy of 81.09% and an F-score of 71.89% on the test set.

4. Decision Trees

Preprocessing:

1. The dataset has been read through read_csv() function on the python inbuilt library.
2. Dropped the missing value in the data.
3. Removed the punctuation form the questions before extracting the features as suggested in the research paper to improve accuracy.

```
def pun(text):
   text = re.sub(r'[^\w\s]','',text)
   return text
```

4. Feature Extraction: We extracted 7 sets of hand engineered features as suggested in the paper and added them incrementally. For testing effectiveness, we tested it on decision trees, random forest as well as gradient boosted tree. The seven features are:
    a. Length based: length for question 1(l1) and question 2(l2) difference in length(l1-l2), ratio of lengths(l1/l2).

```
def length(text):
   return len(text)
```

    b. Number of common lowercase words: count, count/length of the longest sentence

```
def com_low(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check for lower case
   test1_low=set()
   test2_low=set()
   # check if the word is lower case or not
   for word in text1:
       if word.islower():
           test1_low.add(word)
   for word in text2:
       if word.islower():
           test2_low.add(word)
   #check for common lower case words
   common_low=test1_low.intersection(test2_low)
   return len(list(common_low))
```

c. Number of common lowercased words, excluding stop-words: count, count/length of longest sentence.

```
def stop(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check for stop words
   text1_low=set()
   text2_low=set()
   stop_words=set(stopwords.words('english'))
   # check if the word is stop word or not
   for word in text1:
       #checkif word is lower case
       if word.islower():
           text1_low.add(word)

   text1=list(text1_low)
   for word in text2:
       #check if word is lower case
       if word.islower():
           text2_low.add(word)
   text2=list(text2_low)
   #check for lowercase as well as stop_words
   text1=[w for w in text1 if not w.lower() in stop_words]
   text2=[w for w in text2 if not w.lower() in stop_words]
   text1=set(text1)
   text2=set(text2)
   #check for common stop words
   common_stop=text1.intersection(text2)
   return len(list(common_stop))
```

d. Same last word

```
def last(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=list(text1.split(" "))
   text2=list(text2.split(" "))
   #check for last word
   if text1[len(list(text1))-1]==text2[len(list(text2))-1]:
       return 1
   else:
       return 0
```

e. Number of common capitalized words: count, count/length of longest sentence.

```
def com_cap(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check for common capital words
   test1_cap=set()
   test2_cap=set()
   # check if the word is capital or not
   for word in text1:
       if word.isupper():
           test1_cap.add(word)
   for word in text2:
       if word.isupper():
           test2_cap.add(word)
   #check for common capital words
   common_cap=test1_cap.intersection(test2_cap)
   return len(list(common_cap))
```

f. Number of common prefixes, for prefixes of length 3-6: count, count/length of the longest sentence.

```
def pre(text,n):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check for common prefix
   str1=text1
   str2=text2
   str1.sort()
   i=0
   j=0
   str2.sort()
   ans = 0
   while True:
       if i >= len(str1) or j >= len(str2):
           break
       if len(str1[i]) < n:
           i += 1
           continue
       if len(str2[j]) < n:
           j += 1
           continue
       s1 = str1[i][0:n]
       s2 = str2[j][0:n]
       if s1 == s2:
           ans += 1
           i += 1
           j += 1
       else:
           if str1[i]>str2[j]:
               j += 1
           else:
               i += 1
   return ans
```

g. Whether questions 1, 2, and both contain “not”, both contain the same digit, and the number of common lowercased words after stemming.

```
def misc1(text):
   #check if not is present of not
   text=text.split()
   if 'not' in text:
       return 1
   else:
       return 0

def misc2(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check is not is present in both
   if 'not' in text1 and 'not' in text2:
       return 1
   else:
       return 0

def misc3(text):
   text1=text[0]
   text2=text[1]
   #split text1 and 2
   text1=text1.split()
   text2=text2.split()
   #check if word is in digit and is in text2 as well
   for word in text1:
       if word.isdigit():
           if word in text2:
               return 1
   return 0
```

```
def misc4(text):
   text1=text[0]
   text2=text[1]
   ps=PorterStemmer()
   #tokenize word1 and word2
   text1=text1.split()
   text2=text2.split()
   text_s1=[]
   text_s2=[]
   #stemming
   for word in text1:
       text_s1.append(ps.stem(word))
   for word in text2:
       text_s2.append(ps.stem(word))
```

```
text1_low=set()
text2_low=set()
# check if the word is lower case or not
for word in text_s1:
    if word.islower():
        text1_low.add(word)
for word in text_s2:
    if word.islower():
        text2_low.add(word)
#check for common lower case words
common_low=text1_low.intersection(text2_low)
return len(list(common_low))
```

The data is then split into the ratio of 7:2:1 into train, validation, and test datasets respectively.

5. We are using Scikit-learn's implementation of DecisionTreeClassifier, RandomForestClassifier as well as GradientBoostingClassifier.
6. For Decision Trees, we got the best results for max_depth=10 and min_samples_leaf=10
7. For Random Forest, we got the best results for max_depth=30 and min_samples_leaf=100 and n_estimators 100
8. For Gradient Boosted Tree, we got the best results for max_depth=10 and min_samples_leaf=100 and n_estimators 500
9. the **F1 score** is shown below:

Features	**Decision Tree**	**Random Forest**	**Gradient Boosted Tree**
Length Based	0.312840247794	0.314093833654	0.323447893569
Common Lowercased Words	0.553281155930	0.546067570357	0.5620041556
Common Lowercased words excluding stop-words	0.553076176775	0.545417024935	0.546824615593
Same last word	0.531886659901	0.559065019602	0.564132285289

Number of common capitalized words	0.536409740234	0.566977509599	0.574701944076
Number of common prefixes	0.619337164238	0.621370701345	0.633495890597
Misc	0.612443517553	0.621737601125	0.652506658575

6. Neural Network based models

It has been proved as well as observed that neural network models are much efficient in performing modeling on a unstructured data. One of the many reasons in this behavior of neural networks can be accounted to the Universal Approximation nature to approximate any complex function. Therefore, for sake of this project, we built different set of experiments to understand the behavior of the models and identifying the models configuration to give the best results among all.

6.1 CBOW (Common Bag-of-words):

Implemented a **Multilayer Perceptron Model** (MLP) which which stacks a set of dense non-linear layers followed a by softmax operation. We used 3 layer MLP model and used **'Relu'** activation function to induce non-linearity in the model. These 3 dense layers are followed by **softmax** layer to get the probabilities of two classes. Used **'binary-cross entropy'** to calculate the loss and **'Adam'** optimiser as the optimisation algorithm. Adam optimiser uses the combination momentum based movement and learning rate decay technique for better and fast convergence.

6.1.1 Word Embeddings and usage:

CBOW (Continuous Bag of Words) - CBOW is one of the major breakthrough and widely used word embedding technique in text processing space. CBOW embeddings are generated using a Feed Forward Neural Network where the objective is to predict the target word given the context words as input to the network. We used word2vec 300D embeddings for this project to generate the vectors of the words in the question. Extraction of Sentence Embeddings We used the same technique as mentioned in paper to extract the embeddings from both questions and add them to build the relation between questions

Model :-

Built 3 layer MLP model followed by softmax layer. Used sum of the average of the embeddings generated for each question as input to the model. Used 300D word2vec embedding for text vectorisation.

6.1.2 Result

Original Accuracy	Original F1 Score	Our Accuracy	Our F1 score
83.4%	77.86%	85.2% (whole dataset) 71.34% (60k rows out of 400k)	78.71% (whole dataset) 48.2% (60k rows out of 400k)

6.2 GloVe + LSTM

6.2.1 Initial preprocessing with GloVe word embeddings

We employed GloVe embeddings as input embeddings for the creation of an LSTM-based model, as specified in the paper.

GloVe:- GloVe is an unsupervised learning technique that generates word vector representations. The resulting representations highlight intriguing linear substructures of the word vector space, and training is based on aggregated global word-word co-occurrence statistics from a corpus.

Extraction of sentence embeddings:- We used a strategy to translate sentences into vector representations because GloVe provides word vector representations. The approach is same as the research paper authors. For each word in a question, we calculated the word vector representations and added them together. After that, the representations for both questions were concatenated, and the difference and element wise dot product were calculated. The LSTM-based model was then fed the concatenated version of these representations as input.

6.2.2 Replicating previous work along with hyper-parameter tuning

While training the model, we employed different hyper-parameter tuning to check the performance of the model. The summary of the results is described below:-

1. Paper Implementation: LSTM layer with dropout of 0.2.

We used softmax as the final classifying layer for the above configuration. Adam is used as an optimizer with **learning rate = 0.009** and **binary cross entropy** as a loss function.

Original Accuracy	**Original F1 Score**	**Our Accuracy**	**Our F1 accuracy**
81.4%	75.4%	73.6%	64.96%

7 Final Observations and Results :

Model	**Accuracy (%)**	**F-score**
LR with Unigram	74.83	63.62
LR with Bigram	77.92	69.90
LR with Trigram	79.93	71.47
LR with Trigram and tuning (n_iter=15, alpha=0.0001)	80.18	70.12
SVM with Unigram	75.62	64.93
SVM with Bigram	75.57	60.45
SVM with Trigram (after tuning)	81.09	71.89
Decision Tree	72.42	61.24
Random Forest	73.40	62.17

Gradient boosted tree	74.50	62.25
LSTM	73.6	64.96
CBOW	85.2	78.71

We can see that among the linear models, SVM with Trigrams (after tuning), with n-gram counts gave the best accuracy (81.09%), where as in tree based models, with the suggested hand engineered features, the tuned gradient boosted tree and random forest models gave the highest accuracies. In neural networks, CBOW gave highest accuracy of 85.2%.

8 Contribution :

Models

1. Linear Models:

a. Logistic Regression:

Feature extraction, Report, tuning - Eshan Gupta

Preprocessing, Training- Tarun Jindal

Training, report- Eshan Gupta

b. Support Vector Machines:

Preprocessing, feature extraction, Report - Tarun Jindal

Tuning and Training - Eshan Gupta

2. Trees:

a. Decision Trees, Random Forest and Gradient Boosted Tree:

Preprocessing, feature extraction and tuning -Sarthak Aggarwal

preprocessing, training and report - Sarthak Aggarwal

3. Neural Networks:

a. LSTM:

Preprocessing, feature extraction, training, report - Ayan Agrawal

Training,report - Tarun Jindal

b. CBOW:

Preprocessing, embeddings, feature extraction, training, report Ayan Agrawal

training, report - Ayan Agrawal