APPLIED MACHINE LEARNING SYSTEMS II (ELEC0135) 19/20 REPORT : QUORA DUPLICATE QUESTION DETECTION

SN: 19107985

ABSTRACT

Quora is a popular question-and-answer platform, where users are able to ask and answer a range of different questions. Due to its popularity, roughly 300 millions users a month, it is unavoidable to have multiple duplicate questions. Being able to detect these duplicate questions and act accordingly will drastically improve the users' experience on the platform. It is, however, a non-trivial task to detect duplicate questions. What makes detecting duplicate questions difficult can be simplified into two problems: the semantic gap and essential constituent matching.

In this paper, models are proposed and implemented to solve the duplicate question detection problem, using the Quora Question Pairs dataset downloaded from Kaggle. Firstly, following suite of most current literature, a proposed model will use standardised word representations, using Word2Vec embedding, to represent each question. Therefore, this model is solving the semantic gap problem associated with duplicate question detection. A Siamese Bi-Directional Long Short-Term Memory (Si-Bi-LSTM) architecture is used as the base architecture for this model. The similarity measures which are used to detect duplicate questions, based on the encoded questions of each Siamese tower, are the Manhattan Distance or a Feed Forward Network. A comparison of the results obtained by each similarity measure is conducted. Secondly, in an attempt to solve the essential constituent matching problem, a modified version of the approach proposed by Zhang et al. [1] was proposed. A Support Vector Machine using a Radial Basis Function (SVM-RBF), using specialised constituent features is used. Finally, using the SVM-RBF and Si-Bi-LSTM and best performing similarity measure, an ensemble model is created. A weighted combination of the two models outputs is used to perform the final prediction. The effect on the accuracy due to the ensemble model is explored.

The Si-Bi-MaLSTM and Si-Bi-FFNLSTM models were able to achieve an accuracy of 80.42% and 80.21% respectively. The exploration showed that using an ensemble resulted in a 71.92% accuracy, in comparison to the sub-models, where the deep learning model and statistical model achieved accuracies of 70.55% and 65.85% respectively. The code to this project is made available *here*.

1. INTRODUCTION

Quora, an American question-and-answer platform, has about 300 million people who make use of it every month [2]. Users can be broken up into three categories: asker, answerer and seeker. An Asker, as the name suggests, posts questions on Quora which they wish to be answered. An Answerer, typically experts in the field of question [2], answer the questions. Seekers neither post nor answer questions. Instead they rather search through the currently answered questions on Quora, in the hope they will find an answer to a question they have. Due to the sheer amount of users that Quora experiences, and in turn, the amount of questions being posted, it is unsurprising that Askers tend to post questions with the same intent, but with a different wording. Questions with the same intent, and thus can share the same answer, are considered to be duplicate questions. The problem with duplicate questions is that Seekers may have to view several of the same question to find the best answer and therefore wasting their time. Additionally, Answerers may feel obliged to answer multiple versions of the same question, to provide a better chance for Seekers to find then best answer as quickly as possible. Lastly, Askers might waste their time asking a question which has already been asked-and-answered. Together these problems can cause a negative user experience on all fronts, and is therefore crucial for Quora to find a solution.

These problems can all be solved if existing duplicate questions are grouped together and if Askers are referred to existing questions which are duplicates of the one which they are asking. Quora challenged the Kaggle community to apply state-of-the-art natural language processing (NLP) and machine learning (ML) techniques to solve this duplicate question detection problem. They were, at the time of release, using a Random Forest model to identify duplicate questions [3].

According to Zhang et al. [1], the duplicate question detection problem has two major obstacles associated with it. These are semantic gaps and essential constituents matching. The semantic gap refers to the difference between two descriptions of the same object by different linguistic representations. The difference in representations can make comparing two questions difficult. A constituent is a word or a group of words that function as single units within a sentence or question. Therefore, essential constituent matching refers to the matching of constituents between different questions, which are important to the meaning of the questions.

Currently, the work done on solving the duplicate question detection problem mainly focuses on solving the semantic gap problem. This is typically done by either using specialised similarity features or learned standardised representations from neural networks [1]. While these techniques have been shown to perform well, it has been said that essential constituent matching will provide further information, thus further increasing the systems performance.

In this paper, models are proposed and implemented to solve the duplicate question detection problem, using the Quora Question Pairs dataset downloaded from Kaggle. Firstly, following suite of most current literature, a proposed model will use standardised word representations, using Word2Vec embedding, to represent each question. Therefore, this model is solving the semantic gap problem associated with duplicate question detection. A Siamese Bi-Directional Long Short-Term Memory (Si-Bi-LSTM) architecture is used as the base architecture for this model. The similarity measures which are used to detect duplicate questions, based on the encoded questions of each Siamese tower, are the Manhattan Distance or a Feed Forward Network. A comparison of the results obtained by each similarity measure is conducted. Secondly, in an attempt to solve the essential constituent matching problem, a modified version of the approach proposed by Zhang et al. [1] was proposed. A Support Vector Machine using a Radial Basis Function (SVM-RBF), using specialised constituent features is used. Finally, using the SVM-RBF and Si-Bi-LSTM and best performing similarity measure, an ensemble model is created. A weighted combination of the two models outputs is used to perform the final prediction. The effect on the accuracy due to the ensemble model is explored.

This report is organised into 8 sections:

1. **The Introduction**: Provides an overview of the methodologies used to solve the task.

2. **The Literature Survey**: Provides incite into the current literature and the methodologies used to solve related tasks.

3. **The Description of Models**: Describes the models used for each task as well as the reasons for choosing them.

4. **Implementation**: Provides the detailed implementation of the models used.

5. **Experimental Results and Analysis**: Describes and discusses the results obtained.

6. **Conclusion**: Provides a brief review of the results obtained as well as possible ways to improve the models.

7. **References**

8. **Appendix**

2. LITERATURE SURVEY

2.1. Current Approaches

Zolaktaf et al. [4] used six different datasets for detecting duplicate questions, namely; Quora, AskUbuntu, Android, Sprint, Superuser, and Apple. They further joined the Quora and AskUbuntu datasets to form a seventh dataset. They showed that to overcome class imbalance issues, SMOTE [5] can be used. Subsequently, they tested three statistical machine learning (ML) algorithms, namely, Logistic Regression (LR), Random Forrest (RF) and Extreme Gradient Boosting (XGBoost). The features used for these algorithms are n-gram TF-IDF (term frequency–inverse document frequency), character level TF-IDf and Bag of Words (BOW). They further experimented with deep neural network (DNN) models, namely a Siamese DNN [1] using a Long Short-Term Memory (LSTM) which makes use of the Manhattan Distance, referred to as MaLSTM. The MaLSTM model uses Word2Vec word embeddings with dimension of 300 as input features. Their top performing statistical model, Random Forrest, achieved an accuracy of 84.0%. The MaLSTM model achieved an accuracy of 67.56%. They further recommended that other DNN models be tested, such a one which utilised convolutional neural networks (CNN).

Guo et al. [6] proposed two approaches for detecting duplicate questions using the Quora dataset. The first approach used Word2Vec word embeddingss. Each word embedding was then scaled based on its TF-IDF multiplier, before finding the mean word embedding of the question. This was done in an attempt to make up for the information loss due to averaging of the questions. The similarity (based on the inner product of the two question vectors), number of common words and the length difference are used as input features to machine learning algorithms Decision Tree, Random Forest, K-nearest Neighbours (KNN), Support Vector Machine and Adaboost. The second approach was based on an LSTM, with word embeddings as inputs. They subsequently achieved an accuracy of 82.80% using the KNN model and 77.0% using the LSTM model.

Saedi et al. [7] presented a paper which tests and compares the results of popular methods for duplicate question detection using the Quora dataset. In particular, how these methods perform on different dataset sizes. The three approaches used to detect duplicate questions are as follows. A rule based approach which uses the Jaccard Index [2]. A Support Vector Machine which uses features extracted from the questions, such as the Jaccard Index, number of negative words, commons words/nouns etc. A Siamese Network,

[1]Siamese networks are networks that have two or more identical sub-networks/towers in them.

[2]also known as the Jaccard similarity coefficient, is a statistic used in understanding the similarities between samples.

where each tower/path is fed an embedded question. Each tower consists of an identical Convolution Neural Network followed by fully connected dense layers. The cosine distance is used to determine the similarity of the questions. The third approach outperformed the others and achieved an accuracy of 71.49%.

Imtiaz et al. [2] used the Quora dataset to detect duplicated questions. They proposed a system which uses a Siamese MaLSTM. They then used three types of word embeddings, namely, GoogleNewsVector, FastText crawl, and FastText crawl subword embeddings. They compared the results obtained using each embedding separately as the input to the system, and then used a blend of the three embeddings as the input to the system. It was shown that their proposed model, which uses the blended embeddings as the input outperforms state-of-the-art models, achieving an accuracy of 91.14%. It is notable that the systems which used GoogleNewsVector, FastText crawl, and FastText crawl subword embeddings separately, achieved accuracies of 81.77%, 82.77% and 82.57% respectively.

As discussed above, there are two major problems associated with duplicate question detection, namely, the semantic gap and the essential constituents matching. Zhang et al. [1] proposes two approaches to tackle the constituents matching problem, using the Quora dataset. Both approaches aim to integrate FrameNet [3] parsing with neural networks. The first approach uses an ensemble which combines a traditional statistical machine learning algorithm, such as a Gradient Boosting Decision Tree (GBDT), which uses features based on the FrameNet parsing and a neural network model, such as Siamese Bi-Directional LSTM network, which uses word embeddings. The second approach converts the FrameNet parsings into embeddings, which are then combined with standard word embeddings to form the input features for a neural network. Zhang et al, [1] showed that the ensemble approach outperformed the embedding approach. Additionally, it was shown that the ensemble approach increases the accuracy achieved for a variety of models previously shown to perform well on similar tasks. The models, on average, achieved an accuracy of 82.32% before using the ensemble. The proposed ensemble then achieved an average accuracy of 87.44%.

Homma et al. [9] presented a paper which focuses on optimising the use of Siamese Neural Network methods to detect duplicate questions from the Quora dataset. They evaluate two types of neural networks, a Recurrent Neural Network (RNN) and a Gated Recurrent Unit (GRU). They then further explore different methods of determining the similarity between the outputs of each Siamese tower. The first method makes use of a variety of different distance measures, such as cosine distance, euclidean distance and Manhattan distance. These distances were then fed into a Logistic Regression model, to make a prediction. The second method used is a standard feed-forward neural network which takes a concatenation vector made up by the output vectors of the Siamese towers, to make a prediction. Homma et al. [9] used data augmentation to further increase the performance of the system, as it reduces the systems over fitting. Of the methods mention above, the top performing model was a Siamese GRU, which used a feed-forward network to predict the similarity. It achieved an accuracy of 85.0%.

Addair [10] performed a series of experiments, testing a series of deep neural networks for detecting duplicate questions from the Quora dataset. They tested Siamese networks using Convolutional Neural Networks (CNN), Bi-LSTMs and a hybrid model. Similar to the work done by Homma et al. [9], the outputs of the Siamese towers were used to create a concatenation vector to fed into a feed-froward network, to perform the final prediction. They were able to show that the Bi-LSTM system out performed the CNN system, while the hybrid system provided no additional improvements. It is noteworthy that each system did, however, outperform, traditional duplicate question detection baselines. The Siamese Bi-LSTM achieved an accuracy of 81.07%.

2.2. Related Literature

The following topics were further research due to their evident popularity in the current literature, as seen above.

2.2.1. Word Embeddings

Word embeddings are projections of words into a continuous space, such that the semantic and syntactic similarities are preserved [11]. Word embeddings can thus be thought of as a way to represent words such that words with similar meaning have a similar representation. Word embeddings represent meaning through geometry. A good word embedding provides vector representations of words such that the relationship between two vectors mirrors the grammatical relationship between the two words [12]. A practical example of how these embeddings interacting is seen in the following : embedding("Madrid") - embedding("Spain") + embedding("France") is closer to embedding("Paris") than to any other word embedding [13].

Word embeddings have been show to be successful in natural language processing (NLP) and related tasks [11]. A popular word representation is the N-gram model, which is used for statistical language modeling [14]. The N-gram model can be used on nearly all available data. However, this technique and all other similar techniques can no longer provide better performance in a large range of tasks. Due to the increase in processing power and progress of machine learning techniques [14], it has become possible to train far more complex models, on larger datasets. Subsequently, providing better performance than the simpler models, such a N-grams.

[3]FrameNet is based on a theory of meaning called Frame Semantics [8]. The idea is that most words can best be understood on the basis of a semantic frame.

An example of such a model that has attracted a great amount of attention in recent years is the Word2Vec model [15]. Mikolov et al. [14] [13] introduced the Word2Vec model in 2013 and has subsequently been used in a variety of NLP tasks over the years [4] [6] [7]. The Word2Vec model can be trained using two different architectures, namely Continuous Bag-of-Words [14] and Skip-gram [14]. Additionally, the Word2Vec model has three main variants which aim to deal with the inefficiencies of the base Word2Vec model [16] [17]:

1. Word-Phrase Pairing: Treat common words and phrases as single words, thus reducing the number of unique words in the corpus.

2. Sub-sampling: For each word encountered in the corpus, the chance of 'deleting' it from the text is based on its frequency. This means that words such as 'the' will more likely be removed, thus greatly reduce the training samples.

3. Negative Sampling: Each training iteration, only a small number of the models weights are updated, thus greatly reducing the time taken to train.

Ghannay et al. [11] presented a paper which performs an evaluation of a number of popular word embedding architectures, namely, Continuous Bag-of-Words (CBOW), Skip-gram, Global Vectors for Word Representation (GloVe), CSLM and dependency based variant of Word2Vec (w2vf-deps). It was shown that of the word embedding architectures tested, it was observed that w2vf-deps achieved the highest accuracy of all experiments. However, they went on to say that while this approach performed the best, it also requires labeled data, which is not always possible. Thus, of the simpler approaches, Skip-gram and CBOW performed better than the other two approaches. In particular, Skip-grams performed the best for the similarity tasks.

2.2.2. *Long Short-Term Memory (LSTM)*

As mentioned above, due to recent improvements in processing power and deep learning methods, there has been momentous payoff in tasks such as semantic equivalence detection [9]. They have subsequently surpassed the traditional machine learning techniques, such as Support Vector Machines, which use specially designed features as inputs. Additionally, it has been shown that most deep learning systems which aim to detect semantic equivalence make use of Siamese neural networks [9] [4] [7] [1] [10]. The Siamese network, originally introduced by Bromley et al. [18], is a network which consists of two identical sub-networks/towers joined at their outputs, normally by some form of similarity measure. In particular, Siamese Long Short-Term Memory (LSTM), and its variants, have been shown to perform as well and better that state-of-the-art methods in semantic equivalent detection problems [2]. Additionally, as demonstrated by Chen et al. [19], carefully deigned sequential inference models based on LSTMs can outperform all previous models, such as complicated CNN models.

LSTMs are an evolution of Recurrent Neural Networks (RNN). They were originally introduces by Hochreiter et al. [20] to deal with the short fall of RNNs, which was its incapability to learn long-term dependencies [21]. Much like RNNs, they are made up of a chain of repeating nodes. Its nodes however, have a different structure, where each node consists of a memory cell and three regulator units, namely, an input gate, output gate and a forget gate [22]. Together, the regulator gates control the current memory cell and the current hidden state based on the old hidden state and the current input [23].

There are a many different implementations of the an LSTM network. One in particular is the Bi-Directional LSTM, which has been used with great success in duplicate question detection and other semantic similarity problems [1] [10]. The reason for this is as follows: it is often beneficial to have access to future as well as past context when dealing tasks which rely on sequential data. However, the basic LSTM network does not take into consideration the future data, as it only processes the data in temporal manner [23]. Bi-LSTMs on the other hand, by introducing a second hidden layer connected in the opposite direction, allows for the network to take advantage of both the future and the past.

3. DESCRIPTION OF MODELS

3.1. Data Processing

3.1.1. *Removing accented characters*

Accent marks are extremely rare in the English language. Thus, it seems appropriate to replace all accented letters which appear in the corpus with their non-accented counterparts. This will make questions, and their words more comparable. A simple example of this would be replacing ë with e.

3.1.2. *Expanding contractions*

A contracted word is one which has been shortened from its original version, such as **won't**, which is the contraction of **will not**. The shortening of the word normally consists of removing one of the vowels. While contractions exist in both written and spoken forms of the English language, and thus are not incorrect, it is important to ensure that all words with in the corpus are comparable. Therefore, to normalise the corpus, all contractions are converted to their full length version.

3.1.3. *Removing special characters and digits*

Special characters are characters that are not alphabetic or numeric. Examples of such characters are punctuation marks.

Together, special characters and digits can add to the noise in a corpus. Thus, to ensure as little noise as possible is present, all special characters and digits are removed.

3.1.4. Stemming and lemmetisation

A stem is the base form of a word, from which new words can be created by attaching affixes. An example of a stem is **swim**, where new words based off this stem are **swimming** and **swims**. Stemming is the process of obtaining a words' stem by removing the affixes from the word. The process of stemming is used to normalise the corpus, much like expanding contractions. This will in turn assist in the detection process performed later on.

Lemmetisation, much like stemming, is the process of removing affixes from a word to obtain its root word. The difference, however, is that the resulting root word obtained is always present in the dictionary, referred to as a lemma. This ensures that the semantics of the word is not affected, but the corpus is still normalised. Note that while this constraint makes the process of lemmetisation much more accurate, it is also much slower than stemming.

3.1.5. Removing stop words

Stop words are words which are commonly used, such as **the**, **a** and **in**. Stop words are removed from the corpus, as they are unlikely to add any additional information, and will only serve as a hindrance during future processes.

3.2. Word Embedding

As discussed in 2.2, in Natural Language Processing (NLP) tasks traditional machine learning techniques, which make use of specially designed features, have been surpassed by deep learning models which make use of word embeddings. It is for this reason that a deep learning model has been chosen as one of the methods used to solve the duplicate question detection task presented by this paper.

Word embeddings are used as a form of word representation, as described in section 2.2, that bridges the human understanding of language to that of a machine, while ensuring that the semantic and syntactic similarities are preserved. As described in section 2.1, the Word2Vec model has been shown a great amount of attention in recent years, and is implemented in many NLP tasks with great success. It is for these reasons that the Word2Vec word embeddings were used in this paper. The training of a Word2Vec model on the Quora database, while it would likely provide better results, is out of scope. It was thus decided that a pre-trained model made available by google [4] will be used. This model uses the Skip-gram algorithm and negative sampling to output word embeddings with a dimention size of 300.

[4]googlenews-vectors-negative300.bin.gz

As described by Mikolov et al. [14] and the tutorial by McCormick [17] the Skip-Gram Model works as follows. Firstly, a 'fake' task is undertaken. The reason it is considered to be 'fake' is that at the end of the training stage, only the hidden layers weight are of importance, the output values are discarded. The 'fake' task is such that a model is trained to output the probability for every word in the corpus being nearby the input word. This can be seen in figure 1 below.

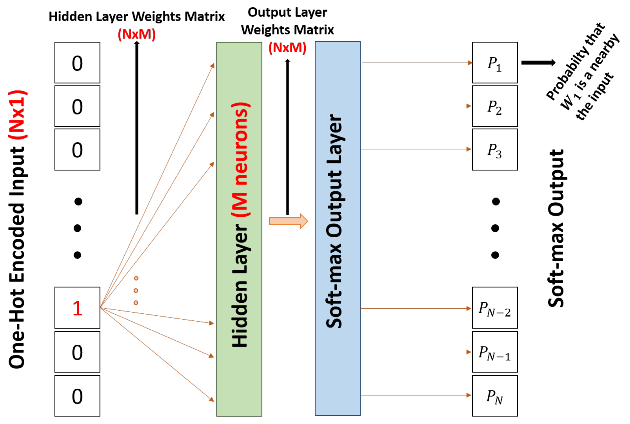

Hidden Layer Weights Matrix (NxM)
Output Layer Weights Matrix (NxM)
One-Hot Encoded Input (Nx1)
0
0
0
1
0
0
Hidden Layer (M neurons)
Soft-max Output Layer
P_1
P_2
P_3
P_{N-2}
P_{N-1}
P_N
Probability that W_1 is a nearby the input
Soft-max Output

Fig. 1. Figure showing the Skip-Gram model used to solve the 'fake' task.

'Nearby' words are defined by a parameter referred to as the window size, which defines how many words before and after the input word should be considered. The output probabilities are related to how likely one is to find each word 'nearby' the inputted word. The network is thus trained by feeding word pairs from the training corpus. For example, give the sentence 'The quick brown fox jumps over the lazy dog' , as each word is fed into the network, the corresponding 'labels' would be words that fall within the window. This means that if fox was the selected word, the word pairs would subsequently be (fox, quick), (fox, brown), (for, jumps), (fox, over) where the window size is two. The input word is represented as a one-hot encoded vector, as seen in figure 1 above, where only the selected word is high. The label is also a one-hot encoded vector. The output layer of the network is a softmax layer.

If the number of unique words in the corpus is equal to N and the number of neurons in the hidden layer is M, the hidden layers' weight matrix is NxM. Once the model is trained, the weight matrix represents the word vector lookup table. This results in N word embeddings, each with a dimension of M features. This can be seen in figure 2.

The intuition behind the Skip-gram architecture is as follows. Given two words which are semantically similar, the pairs of words which they are trained on are likely to be similar. Therefore, as the network is trained to perform the fake task, the corresponding weights of each of these words are likely to be very similar. Similarity can be determined using

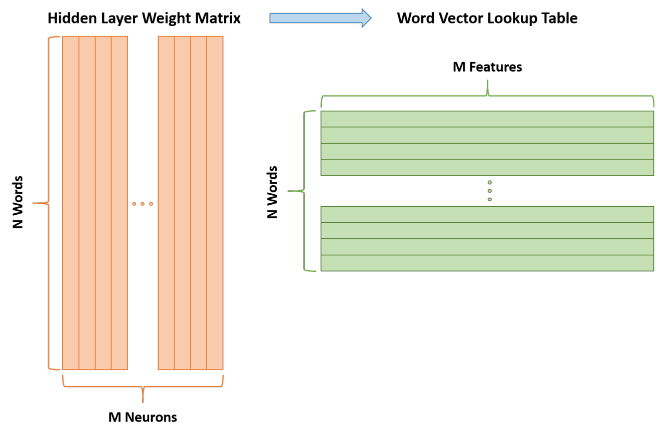


Fig. 2. Figure showing the relationship between the hidden layer matrix and the word vector lookup table.

a distance metric.

Refer to figure 2, where the hidden layer weight matrix and the output layer matrix are both (NxM). This means that there are 2x(NxM) weights which, in theory, get updated every training sample. However, since the input is one-hot encoded, only a small number of the hidden weights are trained every iteration, in particular, the ones linked to the inputted word. However, there are still (NxM) weights which need to be trained. As one can imagine, this becomes a massive task very quickly as the number of unique words in the corpus increases (N) and the word embedding size increases (M). For example, if there are 10000 unique words in the corpus and the dimension of the embeddings are 300. This means that 3 million output weights need to be updated very training iteration.

Negative sampling is one of three methods, as mentioned in section 2.2 above, introduced to deal with this problem. The way in which negative sampling deals with this problem is by having only a few of the weights adjusted every training iteration, apposed to every weight. The weights to be trained are chosen as follows. A small number of 'negative' words, words which are not nearby the inputted word, are randomly chosen. Now, all output weights linked to the negative words along with the output weights linked to the training label are all updated during the training iteration. This means that now instead of (NxM) weights, only a small number of weights are trained every iteration. This makes the process far more efficient.

3.3. Siamese Bi-Directional Long Short-Term Memory (Si-Bi-LSTM) Network

As discussed in section 2.2, it has been shown that Siamese Neural Networks have been utilised with great success in semantic equivalence tasks. It is for these reasons that the Siamese architecture has been used in this paper.

The Siamese network is a network which consists of two identical sub-networks (often referred to as towers) joined at their outputs, normally by some form of similarity measure. This can be seen in figure 3 below.

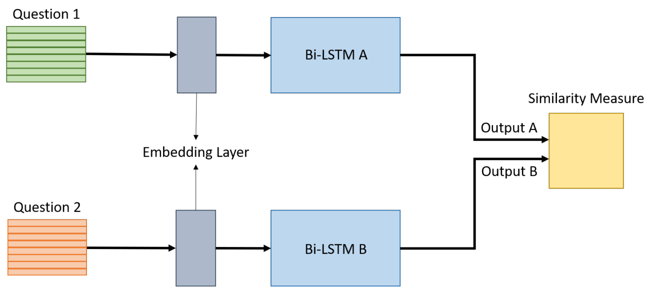


Fig. 3. Figure showing the architecture of a Siamese network.

Furthermore, as discussed in section 2.2 above, the combination of the Siamese network and the LSTM model has been shown to perform as well as, if not better than, state-of-the-art solutions to semantic equivalence problems. Therefore, it is for these reasons that the LSTM model is chosen as the network of choice to be paired with the Siamese architecture.

LSTMs, as discussed briefly in section 2.2, are very similar to RNNs. However, instead of a single layer, it is made up of multiple. The LSTM node is made up of a memory cell and three regulator units, namely, an input gate (i_t), output gate (o_t) and a forget gate (f_t). Together, the regulator gates control the current memory cell (C_t) and the current hidden state (h_t) based on the old hidden state (h_{t-1}) and the current input (x_t). The full structure of the LSTM architecture can be seen in figure 4 below.

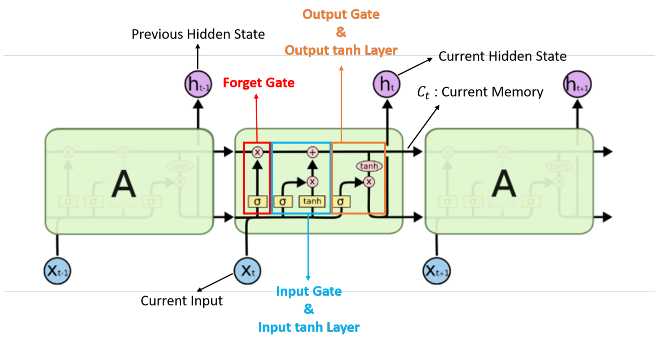


Fig. 4. Figure showing the architecture of a Long Short-Term Memory network. Modified from [17].

The forget gate, a sigmoid layer, decides which information is going to be forgotten/disregarded from the previous memory cell state (C_{t-1}) by multiplying C_{t-1} with the output of the forget gate:

$$f_t = \sigma((W_f)[h_{t-1}, x_t] + b_f) \quad (1)$$

$$\therefore C_{t1} = C_{t-1} \times f_t$$

where W_f and b_f are the weights and biases of the forget gate.

The input gate, a sigmoid layer, decides which new information is going to be stored in the current memory cell state:

$$i_t = \sigma((W_i)[h_{t-1}, x_t] + b_i) \quad (2)$$

where W_i and b_i are the weights and biases of the input gate.

The output of the input gate is then multiplied by the output of the input tanh layer, referred to as the new candidate values (G_t). That is then added to the current memory cell status:

$$G_t = tanh((W_G)[h_{t-1}, x_t] + b_G) \quad (3)$$

$$\therefore C_t = C_{t1} + G_t \times i_t$$

$$\therefore C_t = C_{t-1} \times f_t + G_t \times i_t$$

The output gate, a sigmoid layer, decides what information will be outputted as the cells current hidden state (h_t):

$$o_t = \sigma((W_o)[h_{t-1}, x_t] + b_o) \quad (4)$$

$$\therefore h_t = o_t \times C_t$$

where W_o and b_o are the weights and biases of the output gate.

Lastly, as previously discussed, Bi-Directional LSTMs have been shown to be highly effective in duplicate question detection problems as well as other semantic equivalence problems. It is for these reasons that the Bi-Directional variant of the LSTM network is used in this paper. The structure of a Bi-Directional LSTM can be seen in figure 5.

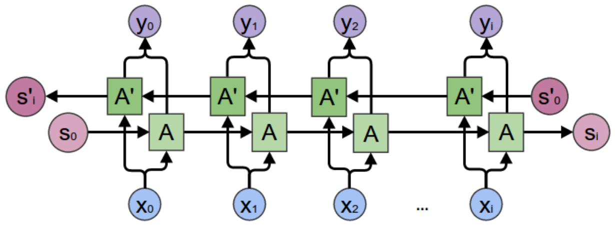


Fig. 5. Figure showing the architecture of a Bi-Directional network. Taken from [21].

Bi-Directional LSTMs allow the network to take advantage of both the future and the past values within a sequence of data. This can be seen in figure 5, where the network flows in both directions.

3.4. Similarity Measure

In this paper, two techniques were proposed to measure the similarity between the output of each of the Siamese towers, as described above. Firstly, the Manhattan Distance (Si-Bi-MaLSTM) was used to calculate the distance between the two output vectors. The reason for selecting the Manhattan distance over other distance metrics is as follows. The Manhattan Distance paired with the LSTM architecture, often referred to as MaLSTM, has been used often to solve semantic equivalence problems, such as the ones describe in these papers [2] [4]. Additionally, Homma et al. [9] further compared the performance of each of the distance metrics in a duplicate question detection problem. It was shown than the the Manhattan Distance outperformed the other distance metrics, namely, the cosine distance and the euclidean distance.

The Manhattan Distance is described as the distance between two points measured along axes at right angles (refer to figure 16 for clarification) and is calculated as follows:

$$M(a, b) = \sum_i |a_i - b_i| \quad (5)$$

The second technique used is a feed-froward neural network (Si-Bi-FFNLSTM), which takes a concatenation vector [10] as its input, to make the final prediction. The concatenation vector is an input vector (V) made up of the two tower output vectors (T):

$$V = [T_1,\ T_2,\ (T_1 - T_2)^2,\ T_1 \odot T_2] \quad (6)$$

The feed-forward network was decided to be two layers deep. This decision was based on the work done by Homma et al. [9] and Addair [10]. Homma et al. [9] compared the results between one layer and two layer deep models in detecting similarity, showing that the two layer deep model had a better performance.

3.5. Ensemble System

As discussed in section 1, there are two major problems associated with duplicate question detection. The semantic gap and essential constituents matching [1]. The Si-Bi-MaLSTM and Si-Bi-FFNLSTM models solve the semantic gap problem. Therefore, in an attempt to solve the essential constituents matching problem, a modified version of the solution described by Zhang et al. [1] is proposed.

Essential constituents of a question refer to parts of the question that are important to its meaning. Note that essential constituents of two non-duplicate questions may be related, but are not the same. For example, questions asking a route to a specific destination, where the destinations are different. However, while the destinations are different, the word embedddings of each destination are likely similar, as they will probably be paired with similar words. This means that a model which relies on purely word representations will likely classify more questions incorrectly as to be duplicate, if question like this exist in the corpus.

It is for this reason that a method to compare the essential constituents between questions is considered. However, it is not a trivial problem to extract and compare the essential constituents of a question. Thus, Zhang et al. [1] instead extracted the semantic frames from the question, as discussed in section 2.2, as it has been shown that there is a considerable relation between the semantic frames the essential constituents.

Due to time and scope limitations, instead of using semantic frames, questions are each broken into their individual constituents. Namely, a question can be broken up into multiple groups of words, where each group is linked to a specific constituent label. The label level feature is the Jaccard Similarity between the label sets of each question. The word level feature is the Jaccard Similarity between each word set, which corresponds to specific constituent label. If a constituent label exist in one question but not the other, this represent the lowest similarity. These features may be likened to the frame and semantic level features described by Zhang [1].

The Jaccard similarity is a comparison of the elements of two sets. The more members which are shared, the higher the Jaccard Similary. The Jaccard Similarity ranges between 0 and 1. It is calculated as follows:

$$J(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{7}$$

The label and word level features are then fed into a standard statistical machine learning model. Zhang el at. [1] used a gradient boost decision tree. For this paper a Support Vector Machine using a Radial Basis Function (RBF) is used. This algorithm was chosen as it has been shown to perform well for a range of different tasks [24] [25], and is thus a good starting point.

As described by Zhang et al. [1], an ensemble architecture between a semantic gap model and a constituents model provides a substantial increase in overall performance, when compared to the performance of each model separately. In this paper, the ensemble is made up of the Si-Bi-LSTM model and the SVM-RBF model. The outputs of each of the models are combined by using a weighted average, from which the label with the highest score is the predicted label. This can be formalised as follows [1]:

$$P = \alpha P_1 + (1-\alpha)P_2 \tag{8}$$

$$y_{pred} = argmax(P)$$

where $\alpha \in [0,1]$ is a hyper-parameter and P_1 and P_2 are the outputs of the two models respectively. P is a two dimensional vector, where each element corresponds to the probability of the label being a 0 or 1. The intuition behind the weight, α, is that as it moves closer to either 0 or 1, one of the two models output will have a bigger effect on the final ensembles prediction. Therefore, and α value of 0.5 is chosen as a good starting point, as this means each model is considered equally. The full ensemble architecture can be seen in figure 6.

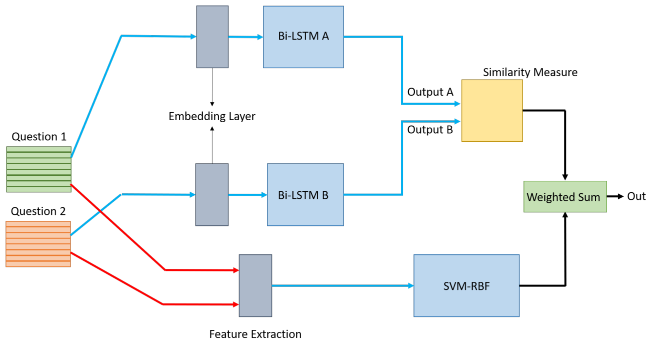


Fig. 6. Figure showing ensemble architecture block diagram.

4. IMPLEMENTATION

The Quora dataset contains 404290 question pairs. Each pair has been subsequently labeled, by a human expert, as being either duplicate questions or not. There are 149263 duplicates and 255027 non-duplicates. There are 105764 more non-duplicate question pairs.

The dataset has the following fields:

1. id: The id of a training set question pair.
2. qid1: Unique ID for question 1.
3. qid2: Unique ID for question 2.
4. question1: The full text of question 2.
5. question2: The full text of question 2.
6. is_duplicate: This is the label, where if the questions are duplicates, the corresponding value is 1, otherwise it is 0.

The following libraries, as well as some other auxiliary libraries, were used to implement the models described in section 3:

1. NumPy: A library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
2. Pandas: A library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series.

3. Scikit-learn: Also known as sklearn, is a machine learning library for the Python programming language. It contains modules capable of various classification, regression and clustering algorithms and is designed to work in conjunction with the Python numerical and scientific libraries NumPy and SciPy.

4. Gensim: An open-source library for unsupervised topic modeling and natural language processing, using modern statistical machine learning.

5. Spacy: An open-source software library for advanced natural language processing.

6. unicodedata: This module provides access to the Unicode Character Database which defines character properties for all Unicode characters. The Unicode standard defines various normalisation forms of a Unicode string.

7. Imbalanced-learn: An open-source python toolbox aiming at providing a wide range of methods to cope with the problem of imbalanced datasets frequently encountered in machine learning and pattern recognition.

8. NLTK: Natural Language Toolkit as a suite of libraries and programs for symbolic and statistical natural language processing for English.

9. allennlp: An open-source NLP research library, built on PyTorch

10. Tensorflow: A open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is often used for machine learning applications such as neural networks

11. Keras: An open-source neural-network library written in Python, capable of running on top of TensorFlow.

4.1. Data Preprocessing

4.1.1. Remove class imbalance

Zolaktaf et al. [4] used a class balancing package to correct the class imbalance. It was thus decided to remove the class imbalance. To remove the class imbalance present in the Quora dataset, the non-duplicate samples were undersampled using *imblearn.under_sampling.RandomUnderSampler*. The *RandomUnderSampler()* object is used to define the desired ratio between the two output classes, namely , $R = N_1/N_2$. Therefore, to correct the imbalance entirely, a ratio of $R = 1$ was chosen. This can be seen in figure 17.

The resulting number of duplicate questions remained unchanged, however, the number of the non-duplicate questions was reduced to a value equal to the number of duplicate questions, namely, 149263. This means that a total of 105764 samples were disregarded. While this is a large number of samples, the number of training samples remaining should be sufficient.

4.1.2. Removing accented characters

unicodedata, as mentioned above, provides a number of normalisation methods. Therefore, unicodedata's *normalize()* function, which takes two parameters is used. The first parameter is the form, which defines the normalisation form to be used, and the second is the corpus text. The form used to perform accent removal is the NFKD which applies compatibility decomposition. This means that all characters are replaced with their compatible equivalents. Following which, the normalised text is reformatted accordingly. This implementation can be seen in figure 18.

4.1.3. Expanding contractions

A contractions dictionary is used to replace all contractions with their expanded versions. A snippet of example extractions can be seen in figure 19. If a word from the corpus matches a contracted word from the dictionary, it is subsequently replaced with its paired expansion from the dictionary. It is important to note that this contraction mapping does not represent all possible contractions in the English language. Therefore, the resulting corpus, after performing contraction expansion, is likely to still have a few contractions. The implementation of this can be seen in figure 20.

4.1.4. Removing special characters and digits

To remove the special characters and digits from the questions, the Regular Expression Operations (RegEx) module is used, namely, *re*. The *re* module has a *sub()* function which takes in a string of characters (particularly the ones which we want substituted), the text which should replace each of these characters if found, and the original text. This implementation can be seen in figure 21.

4.1.5. Lemmetisation

To perform the lemmetisation, a pre-trained Spacy model is used, namely, *en_core_web_sm*[5]. This model is an English multi-task CNN trained on OntoNotes. It Assigns context-specific token vectors, POS tags, dependency parse, named entities etc. to each word in the text. Included in this is the lemma of each word. Therefore, as seen in figure 22, each word is replaced with its lemma, unless it is a pronoun, in which case it remains unchanged.

4.1.6. Removing stop words

To remove the stop words from the questions, the stop word list provided by NLTK is downloaded. Following which,

[5]https://spacy.io/models/en

negation words, such as no and not were subsequently removed from the stop word list. This was done since negation words tend to provide useful information in semantic analysis problems. This can be seen in figure 23.

Now that the stop word list is up to date, the stop words are removed from the text. Firstly, each question is tokenised [6] using the NLTK *ToktokTokenizer()*. Subsequently, the tokens which do not belong to the stop word list are kept, the rest are discarded. This can be seen in figure 24 .

4.2. Word Embedding

The next step taken was to create the Word2Vec word embeddings as discussed in section 3 above. First the Word2Vec model needs to be uploaded. This is done using gensims' *load_word2vec_format()* function. This can be seen in figure 25.

Following which, the corpus' vocabulary of unique words is created. This is done by iterating over all questions, adding words which exist in the Word2Vec vocabulary and that have not already been added to the vocabulary. Once the vocabulary has been created, each word in the vocabulary is iterated through, and its corresponding 300 dimension word embedding is created. This can be seen in figure 26.

4.3. SI-Bi-LSTM Network

The network and its layers are built using Keras and its various number of different layer objects. Firstly, the input layers of the network, one for question 1 and one for question 2 are created. This is done using the Keras *Input* layer. Question 1 and question 2 are referred to as left input and right input respectively.

Next, the Keras *Embedding* layer is defined, where the word embeddings created are used as the weights for the left and right embedding layers. The left and right input layers are set as the input to their respective embedding layers. This can be seen in figure 27.

The next Keras layer to be used is the *LSTM* layer along with the *Bidirectional* layer. The *Bidirectional* layer, refer to figure 28, takes as an input the Keras layer which is intended to be Bidirectional. In this case, that is the Keras *LSTM* layer. The *Bidirectional* layer additionally has a merge-mode. By default, it merges the forward and backwards output's by concatenating them. For the purpose of this paper, this will remain unchanged.

The *LSTM* layer, refer to figure 29, has a number of parameters which are capable of taking on a number of different values. However, according to the Keras documentation, for the layer to use the cuDNN-based [7] implemenation, specific requirements need to be met. These are as follows:

[6]Tokenisation is the process by which big quantity of text is divided into smaller parts called tokens.

[7]https://developer.nvidia.com/cudnn

1. Activation function: tanh
2. Recurrent activation function: sigmoid
3. No recurrent dropout
4. Unrolling of the network is False
5. The layer must use biases
6. Inputs are not masked or strictly right padded.

As seen in figure 29, the default parameters already meet these requirements. Thus, no alterations to the *LSTM* layers' parameters were made, except for the size of the *LSTM* layer, which is tuned. The code used to create the left and right Bi-LSTM layers can be seen in figure 30.

Following the creation of the left and right Bi-LSTM networks, the similarity of the left and right outputs must be determined. Firstly, to calculate the Manhattan distance between the two output tensor vectors of the left and right, the Keras *backend* module is used. The Keras *backend* module was introduced such that low-level operations between tensors could be performed, such as tensor products and convolutions. A helper function, *exponent_neg_manhattan_distance()* as seen in figure 31, takes the left and right output tensors and returns the Manhattan distance between the two.

To call the *exponent_neg_manhattan_distance()* helper function in the training process, Keras' *Lambda* layer is used. The *Lambda* layer wraps arbitrary functions as a layer object as well as defines the output shape of the layer. Therefore, the Manhattan Distance output layer is implemented as seen in figure 32.

The second similarity measure, the feed-forward network, is now considered. To create the concatenation vector described by equation 6, Keras' *backend* is utilised once more. The helper function *create_concat_feature_vector()* is used. This can be seen in figure 33.

As mentioned above, Keras' *Lambda* layer is used to wrap the *create_concat_feature_vector()* helper function. The output of the *Lambda* layer is fed into a two hidden layer deep feed forward network. Each hidden layer is created using Keras' *Dense* layer. Both use the Scaled Exponential Linear Unit (SELU) activation function along with the LeCun normal kernel initialiser. These were chosen based on the article by Lavanya Shukla [26]. They refer to the SELU activation function as being 'an overall robust activation function', and since there is no strict ruling with regards to activation function selection, the SELU function was thus chosen based on their recommendation. Following on from this, Shukla [26] went on to say that the SELU activation function is best paired with the LeCun normal kernel initialiser. Next, a regulariser which applies both L1 and L2 penalties is selected as the regulariser for both layers. This can be seen in figure 34.

Lastly, to bring all of the above together, the model is defined using Keras and is subsequently compiled. This can be seen in figure 35. Note that the optimiser used is the Adam with Nesterov Momentum (Nadam) optimiser. The Nadam optimiser was selected because the Adam optimiser is often referred to as a good starting point, and the Nadam optimiser is merely a variant of the Adam optimiser which uses the Nesterov trick, subsequently speeding up the convergence of the model. Additionally, The loss measure of the model is chosen based on the similarity measure selected. If the Manhattan distance is used, then the Mean-Square-Error is used as the loss function, otherwise, the Sparse Categorical Cross-Entropy loss is used.

4.4. Training and Hyper-Parameter Tuning of Si-Bi-LSTM Network

The data is first split into training, validation and test data. Of the original data, 10% is kept aside as test data, the rest forms the 'full set' of training data. The 'full set' of training data is then further split, 90% of which is kept for the training and 10% is kept for validation.

Up to now, a Siamese Bi-Direction Long Short-Term Memory with Manhattan Distance Similarity Measure (Si-Bi-MaLSTM) model and a Siamese Bi-Direction Long Short-Term Memory with Similarity Network (Si-Bi-FFNLSTM) model have been built but not yet trained. However, as made clear in section 4, there are many hyper-parameters which lend themselves to being tuned. In specific, the parameters of the LSTM layers, Dense layers, output layers, the loss function, the optimisation function and so on. Therefore, to ensure optimal performance from each model, these hyper-parameters should be optimised. However, due to time and processing constraints, not all parameters can be tuned. Only the following parameters were tuned:

1. LSTM and Dense hidden layer size: [32, 64 , 128]
2. Batch size : [32, 64, 128]
3. The learning rate of the optimiser: [1e-2, 1e-3, 1e-4]
4. The decay of the learning rate: [1e-6, 1e-9, 0]
5. The gradient clipping: [0.75, 1, 1.25]
6. L1 regulariser values: [0, 0.01, 0.001, 0.0001]
7. L2 regulariser values: [0, 0.01, 0.001, 0.0001]

The above parameters were chosen to be tuned, as they lend them selves to be more task specific, compared to parameters such as the SELU activation function, which is referred to as being robust across different tasks.

Again, due to time and processing constraints, doing a grid search across all possible combinations of parameters is not plausible. Therefore a randomised grid search is implemented instead. A random parameter value is selected for each of the parameters seen above. The model is then trained using these parameters. The training and validation accuracy are subsequently recorded. This process is repeated a number of times, until sufficient results have been obtained. Each model is trained over 5 epochs, as it is assumed that by this point, a good model will have likely distinguished itself from a bad model.

The Si-Bi-MaLSTM model is only dependent on the LSTM hidden layer size, Batch size, Learning rate, Decay rate, and the gradient clipping. The rest of the parameters are specific to the Si-Bi-FFNLSTM model. The best performing Si-Bi-MaLSTM model used the following hyper-parameters (refer to figure 42):

1. LSTM hidden layer size: 128
2. Batch size : 128
3. The learning rate of the optimiser: 0.001
4. The decay of the learning rate: 0
5. The gradient clipping = 1

The Si-Bi-FFNLSTM model is dependent on all the hyper-parameters mentioned above. The best performing model used the following hyper-parameters (refer to figure 43):

1. LSTM hidden layer size: 64
2. Dense hidden layers 1 and 2: 64 and 128
3. Batch size : 64
4. The learning rate of the optimiser: 0.001
5. The decay of the learning rate: 0
6. The gradient clipping = 1
7. L1 reguliser values: 0.001
8. L1 reguliser values: 0.0001

It should be noted that for both models hyper-parameters, the 'optimal' decay is 0. It could be assumed that the models with 0 decay outperform those with constant decay because the models learning rate diminishes prematurely, and the model becomes stuck in a sub-optimal position. However, without a decay of some-sort the model will never settle, and will instead constantly bounce around the minimum. Therefore, to ensure the learning rate does not decay prematurely, but does decay as it reaches the minimum, a learning rate scheduler is used. This is done using *ReduceLROnPlateau()*, where the learning rate is reduced only when the performance does not increase. It was decided that the scheduler should half the learning rate every time there was no increase in performance for two epochs in a row.

Therefore, each model is trained using the optimised parameters, except for decay which was replaced by the learning-rate scheduler. Each model was allowed to train for a maximum of 25 epochs or until convergence. This was done using the Keras *EarlyStopping* method, where if certain criteria are met, the training is stopped. The chosen criteria, were if there was no increase in the validation performance for five epochs in a row, the training would be stopped and the best model saved. This somewhat ensures that the models do not over-fit the training data.

The Si-Bi-MaLSTM stopped training after 25 epochs, and the Si-Bi-FFNLSTM stopped training after 25 epochs. Figures 44 and 45, show the training epochs of each model.

As seen in the training-validation-accuracy plot for the Si-Bi-MaLSTM model, figure 7, the training accuracy achieved is high, which is an indication of a model with a low bias. Furthermore, the variance in accuracy between the training and validation accuracy is roughly 8% and seems to be further increasing. This is likely indicative that the model has started to over-fit the training data (low-bias and high variance). It is worth noting however, that the validation accuracy does not increase significantly, only enough to prevent early stopping, from around epoch 10. These findings cal be further verified by the training-validation-loss plot, in figure 46.

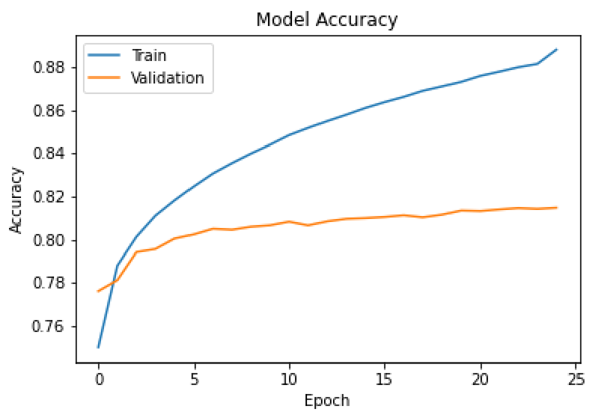


Fig. 7. Figure showing validation and training accuracy achieved by Si-Bi-MaLSTM model during training epochs.

At training epoch 10 the bias is still relatively low, with a training accuracy of 84% but the variance is much lower. Therefore, using a more strict stopping criteria, the Si-Bi-MaLSTM model was retrained. The resulting training-validation-accuracy plot can be seen in figure 8. As expected, the resulting model has shown a low bias with a low variance. This means that the model is well fitted to the training data without over-fitting. It is for these reasons, that this model will be considered further for testing.

As seen in the training-validation-accuracy plot, figure 9, the training accuracy achieved by the Si-Bi-FFNLSTM is high. This is an indication of a model with a low bias. However, the difference in accuracy between the training and validation accuracy is roughly 10%, which indicates a high variance. This might lead us to assume the model might be overfitting to the training data. That being said, as before, is can be seen that the validation accuracy plateaued early on during the training epochs, only increasing in performance enough to prevent training from being stopped early. If you consider epoch 10, the validation accuracy is the same as it was at the end of the training process, but the variance between the training and validation accuracy is roughly halved, now 5%. These findings can be further verified by the training-validation-loss plot, in figure 47.

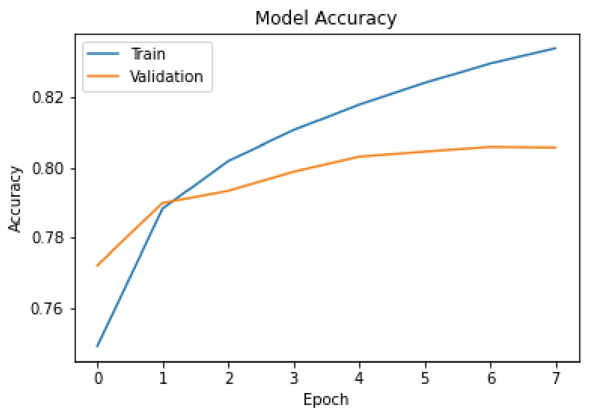


Fig. 8. Figure showing validation and training accuracy achieved by Si-Bi-MaLSTM model during training epochs, with a more strict stopping criteria.

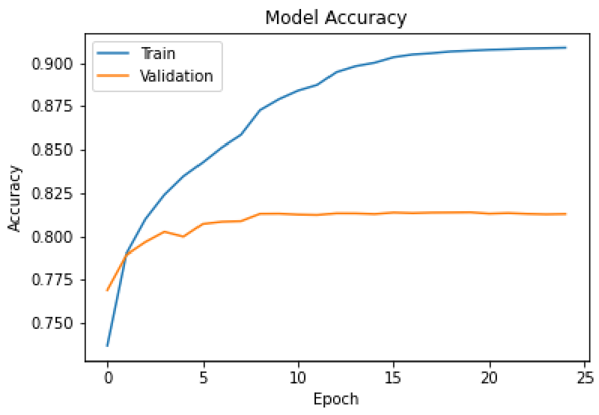


Fig. 9. Figure showing validation and training accuracy achieved by Si-Bi-FFNLSTM model during training epochs.

As for the Si-BiMaLSTM mode, using a more strict stopping criteria, the Si-Bi-FFNLSTM model was retrained. As seen in in the training-validation-accuracy plot, figure 10, with a more strict stopping-criteria the Si-Bi-FFNLSTM model only trained for 7 epochs. As expected, the bias of the model is still low, but with a lower vairance aswell, roughly 6%. This means that the model is well fitted to the training data without over-fitting. It is for these reasons, that this model will be considered further for testing.

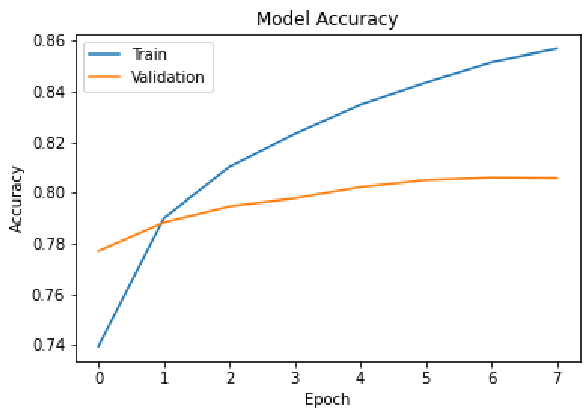


Fig. 10. Figure showing validation and training accuracy achieved by Si-Bi-FFNLSTM model during training epochs, with a more strict stopping criteria.

4.5. Ensemble Model

Firstly, to build the ensemble model, the constituent features need to be created. This is done using the *load_archive()* function from the allennlp library, a pre-trained constituency parser is loaded, as seen in figure 36. Once the pre-trained parser was loaded, the constituents of each question are extracted and used to form a NLTK *Tree*. This is seen in figure 37. Each *Tree* is subsequently decomposed into its most basic constituents, its words and the words respective labels.

To calculate the label-level-feature, two lists (one for each question) made up of the decomposed labels mention above are formed. The Jaccard Similarity, see figure 40, between the two lists is calculated. This can be seen in figure 38.

To calculate the word-level-features, a list of Jaccard Similarity measures is formed, one for each label that exists in the question pair. A mean of the similarity values is then calculated and returned. This can be seen in figure 39. The total time taken to calculate the label and word features for 10000 questions was one hour. Therefore, to calculate the label and word features for all 400000 question pairs would take roughly 40 hours. Therefore, due to time and processing constraints, the ensemble model will be trained, validated and tested on a sample size of 10000 question pairs. These results with serve as an indication of the validity of the proposed architecture.

Once the features for the 10000 question pairs were calculated, they were split into training, validation and testing. As discussed above, the SVM-RBF is used. The SVM-RBF has two main hyper-parameters, C and Gamma. A cross-validation grid-search across a range of possible hyper-parameter values was done using the *GridSearchCV()* method. The resulting best parameters are as follows: $C = 1000$ and $gamma = 0.1$. Using these parameters the final model is trained and tested.

The learning curve of the SVM-RBF model, as seen in figure 11 was plotted to further analyse the bias and variance of the model. The learning curve shows a relatively low settling accuracy (+-66%) when compared to the class ratio (+-50%), however, the validation accuracy closely follows the training accuracy. This means that the SVM-RBF model has a high bias and a low variance. This mean that the model is underfitted to the data. However, it is clear that the training and validation accuracy has plateaued (does not increase after 4000 training samples) and will thus likely not benefit from additional training data. Furthermore, a SVM-RBF model with a very large C and gamma value, 100000 and 100 respectively, showed that the situation does not improve. Therefore, it can be deduced that a more complex feature space needs to be considered.

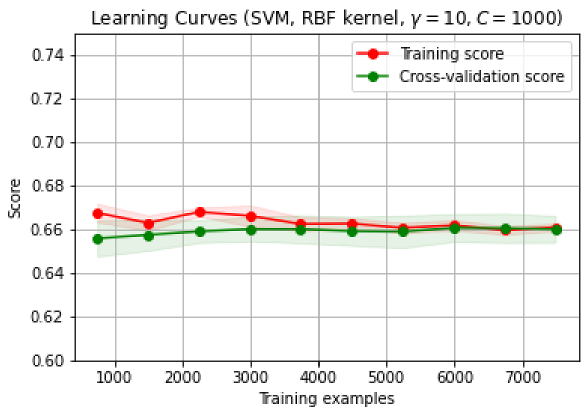


Fig. 11. Figure showing learning curve of SVM-RBF model.

Based on the results obtained following the testing of the Si-Bi-MaLSTM and Si-Bi-FFNLSTM models, the Si-Bi-FFNLSTM model is used as part of the ensemble with the SVM-RBF model discussed above. The Si-Bi-FFNLSTM model is first retrained using the same training samples as the SVM-RBF model. The model stopped early at 22 epochs. As seen in figure 12, the training accuracy achieved is high, indicating a low bias. Furthermore, the variance in training and validation accuracy is roughly 15%. This indicates that the model is very likely to suffer from over-fitting. This result is not surprising, as the training sample size was significantly smaller than the one which the original Si-bi-FFNLSTM model was trained on, thus the model is able to fit the smaller data much easier. That being said, as discussed before, the validation accuracy seems to have somewhat plateaued after the 10th epoch, where the variance in accuracy is roughly

10%.

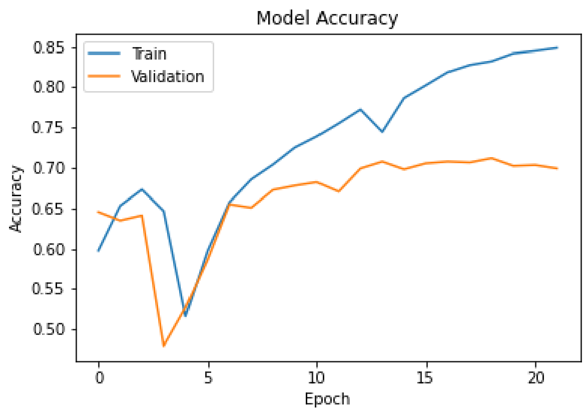


Fig. 12. Figure showing validation and training accuracy achieved by ensemble Si-Bi-FFNLSTM model during training epochs.

As before, the model is therefore retrained using a more strict stopping criteria. As seen in the training-validation-accuracy plot, figure 10, with a more strict stopping-criteria, the Si-Bi-FFNLSTM model only trained for 10 epochs. As expected, the bias of the model is still low, but with a lower vairance aswell, roughly 10%. This means that the model is better fitted to the training data with less over-fitting. It is for these reasons, that this model will be considered further for testing.

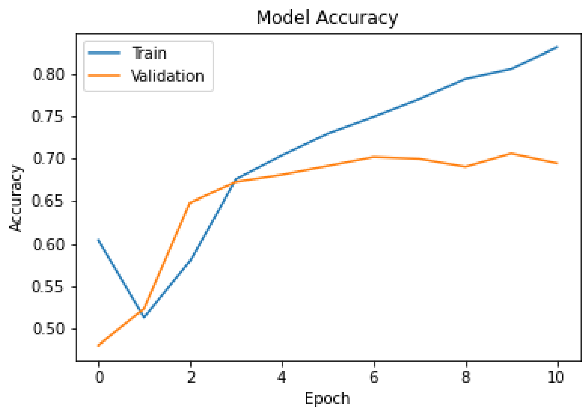


Fig. 13. Figure showing validation and training accuracy achieved by ensemble Si-Bi-FFNLSTM model during training epochs, with a more strict stopping criteria.

Now that both models have been trained, the ensemble can be formed. As mentioned in section 3.5, the final prediction is a weighted sum of the two models predictions, as shown in figure 41.

The ensemble model was tuned by stepping through weight values 0 to 1 in increments of 0.1, and subsequently calculating the validation accuracy. It was found that the weight value of 0.6 provided the highest validation accuracy. Finally, the ensemble architecture was tested and the accuracy recorded.

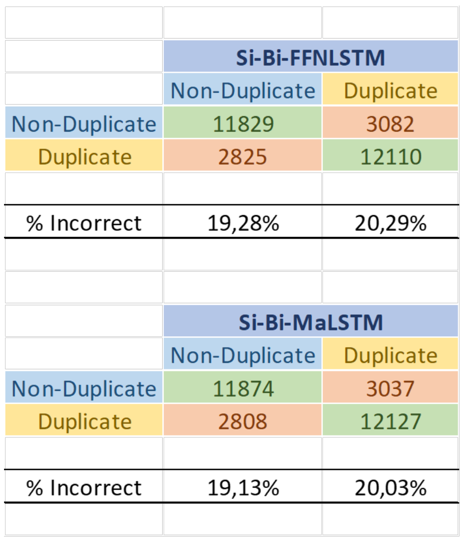
5. EXPERIMENTAL RESULTS AND ANALYSIS

5.1. Results

The following results, tables 1 and 2, were obtained using the described final models in section 4.

Model	Train Acc	Val Acc	Test Acc
Addair [10]	N/A	N/A	0.8107
Homma et al. [9]	N/A	N/A	0.8500
Zhang et al. [1]	N/A	N/A	0.8744
Imtiaz et al. [2]	N/A	N/A	0.8177
Saedi et al. [7]	N/A	N/A	0.7149
Guo et al. [6]	N/A	N/A	0.7700
Zolaktaf et al. [4]	N/A	N/A	0.6756
Si-Bi-MaLSTM (25)	0.8933	0.8148	0.8146
Si-Bi-MaLSTM (7)	0.8437	0.8057	0.8042
Si-Bi-FFNLSTM (25)	0.9089	0.8129	0.8133
Si-Bi-FFNLSTM (7)	0.8581	0.8050	0.8021

Table 1. Table showing accuracies achieved on training, validation and test data, as well the accuracies achieved by models from reviewed literature. Note that the bracketed values represent the total number of training epochs.



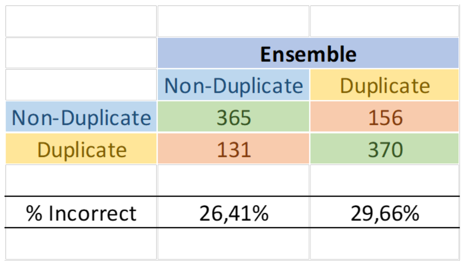
	Si-Bi-FFNLSTM	
	Non-Duplicate	Duplicate
Non-Duplicate	11829	3082
Duplicate	2825	12110
% Incorrect	19,28%	20,29%

	Si-Bi-MaLSTM	
	Non-Duplicate	Duplicate
Non-Duplicate	11874	3037
Duplicate	2808	12127
% Incorrect	19,13%	20,03%

Fig. 14. Figure showing confusion matrix for models Si-Bi-FFNLSTM (7) and Si-Bi-MaLSTM (7).

Model	Train Acc	Val Acc	Test Acc
Si-Bi-FFNLSTM (22)	0.8522	0.6990	0.7241
Si-Bi-FFNLSTM (10)	0.7923	0.7001	0.7055
SVM-RBF	0.6590	0.6864	0.6585
Ensemble	N/A	0.7307	0.7192

Table 2. Table showing accuracies achieved using ensemble architecture on training, validation and test data. Note that the bracketed values represent the total number of training epochs.



	Ensemble	
	Non-Duplicate	Duplicate
Non-Duplicate	365	156
Duplicate	131	370
% Incorrect	26,41%	29,66%

Fig. 15. Figure showing confusion matrix for ensemble model.

5.2. Analysis and Discussion

5.2.1. Si-Bi-LSTM Models

Let us consider the results seen in table 1. As discussed in section 4.4, the Si-Bi-FFNLSTM (25) model over-fitted to the training data when using a more relaxed stopping criteria. It was therefore decided to train another Si-Bi-FFNLSTM (7) model using a more strict stopping criteria such that the model becomes less over-fitted to the training data, and thus more generalisable. As seen in table 1, the Si-Bi-FFNLSTM (7) model was still able to achieve similar test accuracy of 80.42% as the Si-Bi-FFNLSTM (25) model, which achieved a accuracy of 81.46%. The same analysis can be done for the Si-Bi-MaLSTM (25) model, which also over-fitted to the training data when using a more relaxed stopping criteria. As seen in table 1, both the Si-Bi-MaLSTM (25) and Si-Bi-MaLST (7) models achieved similar test accuracies, 81.33% and 80.21% respectively.

It should also be noted that the Si-Bi-FFNLSTM model performed similarly to the Si-Bi-MaLSTM model, which is somewhat surprising. According to the work done by Homma et al. [9], the feed-forward network variant was expected to outperform the Manhattan Distance variant. Additional tests need to be performed to validate the current results. Furthermore, in figure 14, the confusion matrices for models Si-Bi-FFNLSTM and Si-Bi-MaLSTM were created. It can be seen that both models perform similarity when detecting both duplicate and non-duplicate questions. An interesting observation from this is that the models trained are unbiased, however, since the models trained are solving the semantic gap problem and not the essential constituency matching problem, it was expected that more questions would be incorrectly categorised as duplicate, as previously discussed in sub-section 3.5. Since this is not the case, it can be assumed that the Quora database has very few questions which rely on essential constituency matching to be classified correctly. Further analysis on this is done in sub-section 5.2.2.

In comparison to the models in the reviewed literature, whose results are summarised in table 1, the Si-Bi-MaLSTM and FFNLSTM models perform on par or better than the majority of the reviewed models, with exception to two models, namely the models proposed by Homma et al. [9] and Zhang et al. [1].

While the performance of both models is sufficient, it is worth noting possible drawbacks which likely effected their performance. There was a large set of training samples made available by Quora. However, a quarter of the data was dropped, to remove the class imbalance present. Therefore, a large portion of the data, which would have provided valuable information to the model was completely discarded. Next, since a pre-trained word embedding model is used, and is therefore not specific to the Quora dataset, a source of irreducible error is introduced from the start. Another drawback is the lack of hyper-parameter tuning. While some tuning was performed, it is very likely that a large portion of the hyper-parameter space was untested, likely leaving the model in a sub-optimal state.

5.2.2. Ensemble Model

The results seen in table 2 are obtained by the ensemble model and its sub-models. The sub-models, Si-Bi-FFNLSTM (10) and SVM-RBF, achieved test accuracies of 70.55% and 65.85% respectively. The ensemble model, using a weight of 0.6, achieved an accuracy of 71.92%. This is an increase in accuracy of almost 2% over the Si-Bi-FFNLSTM (10) model, and around 6% increase over the SVM-RBF model. The increase in the overall accuracy confirms the results obtained by Zhang et al. [1]. However, it is worth noting that our ensemble, unlike the ensemble proposed by Zhang et al. [1], did not use both output of the sub-models equally. Instead, the SVM-RBF was weighted more than the Si-Bi-FFNLSTM (10). This was an unexpected result, since the since the Si-Bi-FFNLSTM (10) sub-model outperformed the SVM-RBF sub-model. Based on these initial results, it can be said that the SVM-RBF model successfully learned additional information based on the constituent features which in turn increase the overall performance of the ensemble model. This confirms the results obtain by Zhang et al. [1]. As seen in figure 15, a confusion matrix for the ensemble model is created. As seen, the model performs similarly when detecting both duplicate and non-duplicate questions. According to the work done by Zhang et al. [1], it was expected that since the previous models, which solely aimed to solve the semantic gap problem, performed equally on both classes, that the ensemble model which takes into consideration essential constituency matching, should have resulted in better performance in predicting non-duplicate questions. This further confirms the previous assumption, that the Quroa database has very few questions which rely on essential constituency matching to be classified correctly. Therefore, the initial conclusion that the ensembles increase in performance due to the inclusion of a constituent based model might be incorrect. Therefore, the observed increase in performance is likely solely due to the ensemble architecture, which itself typically results in an increase in performance.

Therefor out results did not match those achieved by the ensemble proposed by Zhang et al. [1]. This can be assumed to be due to two major factors, the training size and the features used. The size of the training data used was a mere 2.5% of the entire dataset, which would have a drastic negative effect on the overall performance of the model, as well as its reliability. Unlike the features used by Zhang et al. [1], our features did not extract the essential constituents, instead all constituents were used. This likely introduced unnecessary noise, which in turn likely caused the variance in results obtained.

6. FUTURE WORK

6.1. Si-Bi-LSTM Models

There has been shown to be some merit in data augmentation as a method to reduce over-fitting as well as increase the performance of the model, as show by Homma et al. [9]. Therefore, it is recommended that instead of under-sampling the non-duplicate class, one should use data augmentation to increase the number of duplicate question pairs.

Based on the work done by Imtiaz et al. [2], it is recommended that other pre-trained word embeddings models should be tested, since different embeddings word better for certain datasets. Additionally, blending the outputs of each embedding model was shown to greatly increase the performance of the model [2]. This might allow for a more robust word embedding, capable of working on multiple datasets.

As mentioned by other researchers [6] [9], it has been demonstrated that weighting the word embeddings by their corresponding term frequency–inverse document frequency (TF-IDF)[8] values is likely to increase the performance of the model. This ensure that important word and their corresponding embeddings have a higher impact on the results.

Training word embeddings on the Quora dataset will increase the accuracy of the embeddings and subsequently increase the performance of the model. However, it is should be noted that if the embeddings are trained using the Quora dataset, the model will likely perform worse for duplicate questions not provided by Quora.

Based on the model proposed by Homma et al. [9], one might consider testing additional network architectures, such as the General Recurrent Unit (GRU) and other variants of the LSTM, in place of the Bi-LSTM.

Choosing the best hyper-parameters can vastly improve the performance of a model. Therefore, it is recommended that a more complete search of the hyper-parameter space is conducted, where the optimiser, loss function, activation function and so on are also included. In addition to improved performance, selecting the optimal hyper-parameters provide further incite into the inner workings of the model etc. Lastly, it was chosen that each model be stopped after after 25 training epochs. While it seems that the loss and accuracy of the models had plateaued by that point, there might be merit in allowing the models to train for longer.

6.2. Ensemble Model

To overcome the two factors mention above, the following is suggested. First, the number of training samples, as discussed previously, were limited due to the inefficiency of the method used to create the constituent features. Since there was a limit to the amount of processing power available, a more efficient method needs to be considered, such that more training samples can be processed and subsequently used to train the models. Second, the constituent features seemed to struggle to capture as much information as the features described by Zhang et al. [1] did. It is thus suggested that features similar to the FrameNet parsing features used by Zhang et al. [1] are used, in the hope that more information can be captured.

7. CONCLUSION

In this paper two model architectures were proposed, following the work done in previous literature, as a solution to the duplicate question detection problem.

Firstly, to solve the semantic gap associated with duplicate question detection, two models were used: Si-Bi-MaLSTM and Si-Bi-FFNLSTM. Word2Vec word embeddings are used at the inputs to these models. After testing, it was found that the Si-Bi-FFNLSTM model achieved an accuracy of 80.21%, and the Si-Bi-MaLSTM model achieved an accuracy of 80.42%. In summary of section 6.1, the proposed models can be improved by implementing data augmentation to reduce over-fitting as well as increase number of training samples. Testing other pre-trained word embeddings models and their combinations as well as a word embeddings model trained on the Quora dataset should be considered. The word embeddings used should be weighted by their TF-IDF values, to filter out unimportant words. Other network architectures should be also considered and finally, a more complete

[8]Numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.

hyper-parameter optimisation should be followed.

Second, to solve the essential constituent matching problem associated with duplicate question detection, an ensemble model, which consists of two sub-models, a Si-Bi-FFNLSTM and a SVM-RBF is used. A set of constituent features are used as input to the SVM-RBF model and standard word embeddings for the Si-Bi-FFNLSTM. The sub-models, Si-Bi-FFNLSTM and a SVM-RBF, achieved accuracies of 70.55% and 65.85% respectively when tested independently. The ensemble was able to achieve a test accuracy of 71.92%. An increase of 2% and 6% in accuracy was achieved when using the ensemble model. This approach aims to serve as a proof of concept for a solution to the essential constituents matching problem related to duplicate question detection. while the results obtained do not match those in the literature, this proof of concept has shown promise, and should therefore be considered further. As discussed in section 6.2, to improve the performance of the ensemble model, a more efficient and accurate essential constituent feature extractor needs to be used.

8. REFERENCES

[1] Xiaodong Zhang, Xu Sun, and Houfeng Wang, "Duplicate question identification by integrating framenet with neural networks," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[2] Zainab Imtiaz, Muhammad Umer, Muhammad Ahmad, Saleem Ullah, Gyu Sang Choi, and Arif Mehmood, "Duplicate questions pair detection using siamese malstm," *IEEE Access*, vol. 8, pp. 21932–21942, 2020.

[3] "Quora question pairs," .

[4] Neda Zolaktaf and Vaishnavi Malhotra, "Duplicate questions across multiple question-answering forums," .

[5] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[6] Lei Guo, Chong Li, and Haiming Tian, "Duplicate quora questions detection," 2017.

[7] Chakaveh Saedi, Joao Rodrigues, João Silva, Vladislav Maraev, et al., "Learning profiles in duplicate question detection," in *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2017, pp. 544–550.

[8] Collin F Baker, Charles J Fillmore, and John B Lowe, "The berkeley framenet project," in *Proceedings of the 17th international conference on Computational linguistics-Volume 1*. Association for Computational Linguistics, 1998, pp. 86–90.

[9] Yushi Homma, Stuart Sy, and Christopher Yeh, "Detecting duplicate questions with deep learning," in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS*, 2016.

[10] Travis Addair, "Duplicate question pair detection with deep learning," *Stanf. Univ. J*, 2017.

[11] Sahar Ghannay, Benoit Favre, Yannick Esteve, and Nathalie Camelin, "Word embedding evaluation and combination," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, 2016, pp. 300–305.

[12] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims, "Evaluation methods for unsupervised word embeddings," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, 2015, pp. 298–307.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[14] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[15] Xin Rong, "word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[16] "Google code archive - long-term storage for google code project hosting.," .

[17] Chris McCormick, "Word2vec tutorial-the skip-gram model," 2016.

[18] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah, "Signature verification using a" siamese" time delay neural network," in *Advances in neural information processing systems*, 1994, pp. 737–744.

[19] Qian Chen, Xiaodan Zhu, Zhenhua Ling, Si Wei, Hui Jiang, and Diana Inkpen, "Enhanced lstm for natural language inference," *arXiv preprint arXiv:1609.06038*, 2016.

[20] Sepp Hochreiter and Jürgen Schmidhuber, "Lstm can solve hard long time lag problems," in *Advances in neural information processing systems*, 1997, pp. 473–479.

[21] "Understanding lstm networks," .

[22] Shu Zhang, Dequan Zheng, Xinchen Hu, and Ming Yang, "Bidirectional long short-term memory networks for relation classification," in *Proceedings of the 29th*

Pacific Asia conference on language, information and computation, 2015, pp. 73–78.

[23] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis Lau, "A c-lstm neural network for text classification," *arXiv preprint arXiv:1511.08630*, 2015.

[24] Nathan Herr, "Investigation into sEMG Signal Processing for Gesture Classification," pp. 83–92, 2018.

[25] Nathan Herr, "Applied machine learning system elec0132 19/20 report," 2020.

[26] lavanyashukla01, "Training a neural network? start here!," Mar 2020.

9. APPENDIX A

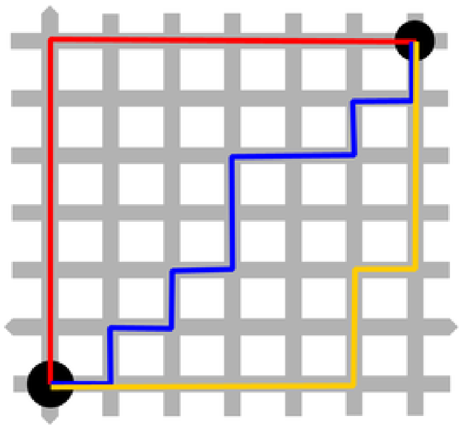


Fig. 16. Figure showing the Manhattan distances between two points

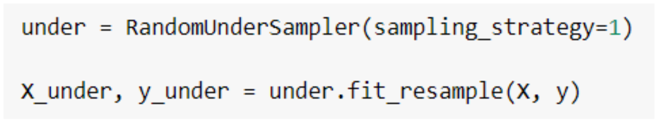

```
under = RandomUnderSampler(sampling_strategy=1)

X_under, y_under = under.fit_resample(X, y)
```

Fig. 17. Figure showing code snippet used to remove class imbalance in data.

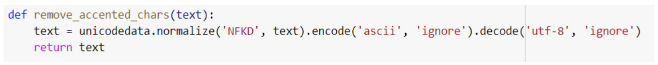

```
def remove_accented_chars(text):
    text = unicodedata.normalize('NFKD', text).encode('ascii', 'ignore').decode('utf-8', 'ignore')
    return text
```

Fig. 18. Figure showing code snippet used to remove accents from corpus.

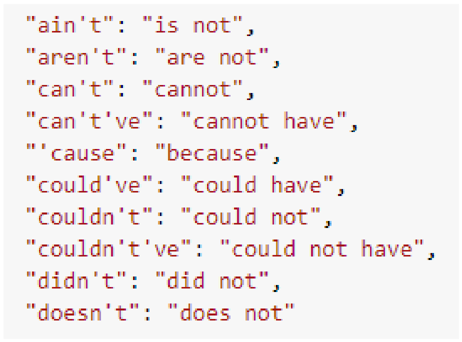

```
"ain't": "is not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not"
```

Fig. 19. Figure showing code snippet of a few example contractions considered.

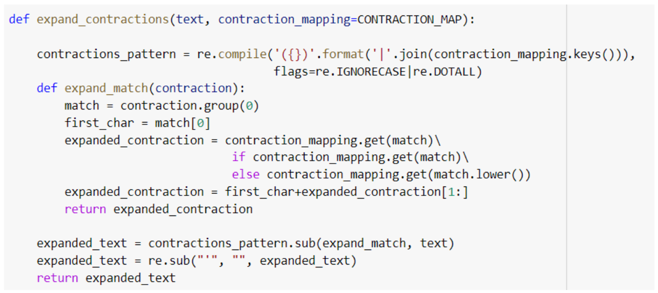

```
def expand_contractions(text, contraction_mapping=CONTRACTION_MAP):

    contractions_pattern = re.compile('({})'.format('|'.join(contraction_mapping.keys())),
                                      flags=re.IGNORECASE|re.DOTALL)
    def expand_match(contraction):
        match = contraction.group(0)
        first_char = match[0]
        expanded_contraction = contraction_mapping.get(match)\
                               if contraction_mapping.get(match)\
                               else contraction_mapping.get(match.lower())
        expanded_contraction = first_char+expanded_contraction[1:]
        return expanded_contraction

    expanded_text = contractions_pattern.sub(expand_match, text)
    expanded_text = re.sub("'", "", expanded_text)
    return expanded_text
```

Fig. 20. Figure showing code snippet used to expand all contraction in corpus.

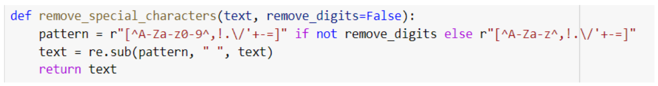

```
def remove_special_characters(text, remove_digits=False):
    pattern = r"[^A-Za-z0-9^,!.\/'+-=]" if not remove_digits else r"[^A-Za-z^,!.\/'+-=]"
    text = re.sub(pattern, " ", text)
    return text
```

Fig. 21. Figure showing code snippet used to remove special characters and digits in corpus.

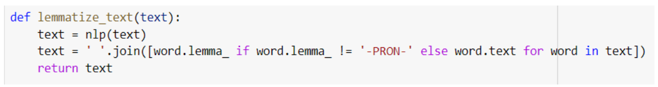

```
def lemmatize_text(text):
    text = nlp(text)
    text = ' '.join([word.lemma_ if word.lemma_ != '-PRON-' else word.text for word in text])
    return text
```

Fig. 22. Figure showing code snippet used to perform lemmatisation on all words in corpus.

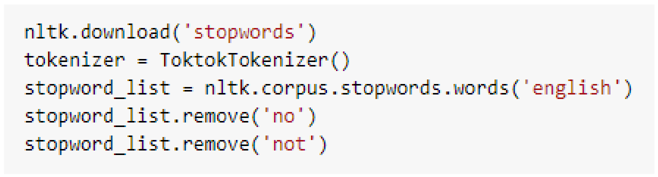
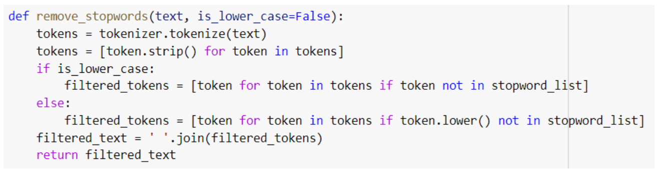
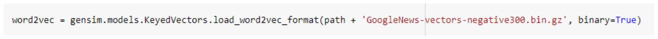

```
nltk.download('stopwords')
tokenizer = ToktokTokenizer()
stopword_list = nltk.corpus.stopwords.words('english')
stopword_list.remove('no')
stopword_list.remove('not')
```

Fig. 23. Figure showing code snippet used to load stop words and remove negation stop words from list.



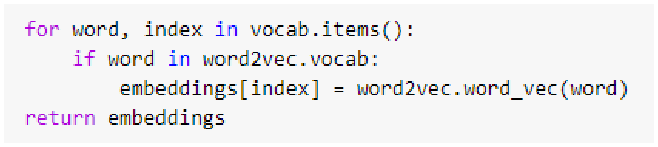
```
def remove_stopwords(text, is_lower_case=False):
    tokens = tokenizer.tokenize(text)
    tokens = [token.strip() for token in tokens]
    if is_lower_case:
        filtered_tokens = [token for token in tokens if token not in stopword_list]
    else:
        filtered_tokens = [token for token in tokens if token.lower() not in stopword_list]
    filtered_text = ' '.join(filtered_tokens)
    return filtered_text
```

Fig. 24. Figure showing code snippet used to remove all stop words from corpus.



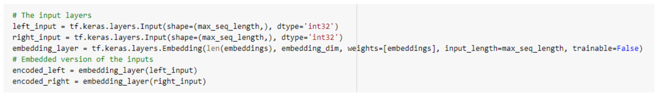
```
word2vec = gensim.models.KeyedVectors.load_word2vec_format(path + 'GoogleNews-vectors-negative300.bin.gz', binary=True)
```

Fig. 25. Figure showing code snippet used to load the Word2Vec model.



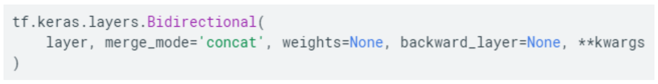
```
for word, index in vocab.items():
    if word in word2vec.vocab:
        embeddings[index] = word2vec.word_vec(word)
return embeddings
```

Fig. 26. Figure showing code snippet used to create word embeddings.



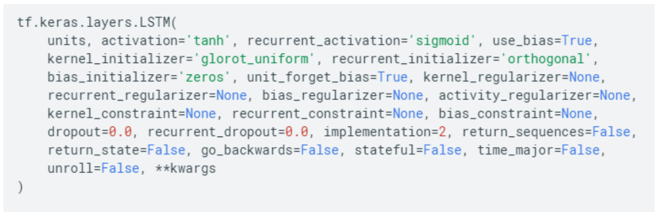
```
# The input layers
left_input = tf.keras.layers.Input(shape=(max_seq_length,), dtype='int32')
right_input = tf.keras.layers.Input(shape=(max_seq_length,), dtype='int32')
embedding_layer = tf.keras.layers.Embedding(len(embeddings), embedding_dim, weights=[embeddings], input_length=max_seq_length, trainable=False)
# Embedded version of the inputs
encoded_left = embedding_layer(left_input)
encoded_right = embedding_layer(right_input)
```

Fig. 27. Figure showing code snippet used to create embedding layer.



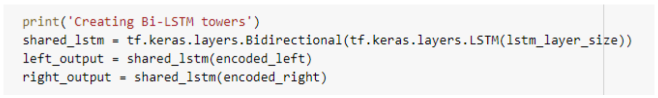
```
tf.keras.layers.Bidirectional(
    layer, merge_mode='concat', weights=None, backward_layer=None, **kwargs
)
```

Fig. 28. Figure showing the parameters and their default values of the Keras Bidirectional Layer.



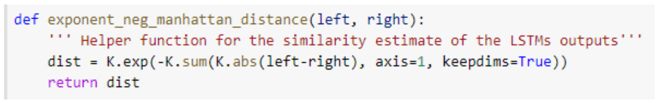
```
tf.keras.layers.LSTM(
    units, activation='tanh', recurrent_activation='sigmoid', use_bias=True,
    kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal',
    bias_initializer='zeros', unit_forget_bias=True, kernel_regularizer=None,
    recurrent_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, recurrent_constraint=None, bias_constraint=None,
    dropout=0.0, recurrent_dropout=0.0, implementation=2, return_sequences=False,
    return_state=False, go_backwards=False, stateful=False, time_major=False,
    unroll=False, **kwargs
)
```

Fig. 29. Figure showing the parameters and their default values of the Keras LSTM layer.



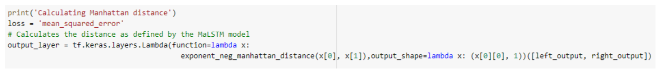
```
print('Creating Bi-LSTM towers')
shared_lstm = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(lstm_layer_size))
left_output = shared_lstm(encoded_left)
right_output = shared_lstm(encoded_right)
```

Fig. 30. Figure showing code snippet used to create LSTM layer.



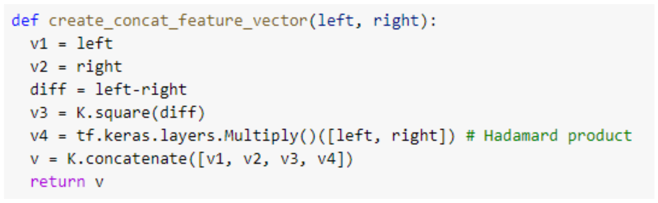
```
def exponent_neg_manhattan_distance(left, right):
    ''' Helper function for the similarity estimate of the LSTMs outputs'''
    dist = K.exp(-K.sum(K.abs(left-right), axis=1, keepdims=True))
    return dist
```

Fig. 31. Figure showing code snippet used calculate the Manhattan distance.



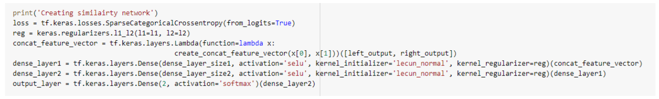
```
print('Calculating Manhattan distance')
loss = 'mean_squared_error'
# Calculates the distance as defined by the MaLSTM model
output_layer = tf.keras.layers.Lambda(function=lambda x:
                                      exponent_neg_manhattan_distance(x[0], x[1]),output_shape=lambda x: (x[0][0], 1))([left_output, right_output])
```

Fig. 32. Figure showing code snippet used to create the Manhattan distance output layer.



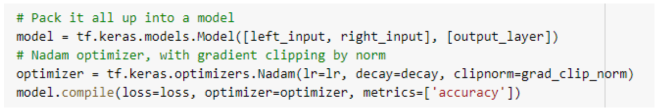
```
def create_concat_feature_vector(left, right):
  v1 = left
  v2 = right
  diff = left-right
  v3 = K.square(diff)
  v4 = tf.keras.layers.Multiply()([left, right]) # Hadamard product
  v = K.concatenate([v1, v2, v3, v4])
  return v
```

Fig. 33. Figure showing code snippet used to create the concatenation vector.



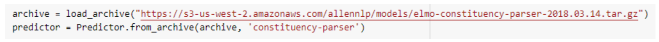
```
print('Creating similairty network')
loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
reg = keras.regularizers.l1_l2(l1=l1, l2=l2)
concat_feature_vector = tf.keras.layers.Lambda(function=lambda x:
                                    create_concat_feature_vector(x[0], x[1]))([left_output, right_output])
dense_layer1 = tf.keras.layers.Dense(dense_layer_size1, activation='selu', kernel_initializer='lecun_normal', kernel_regularizer=reg)(concat_feature_vector)
dense_layer2 = tf.keras.layers.Dense(dense_layer_size2, activation='selu', kernel_initializer='lecun_normal', kernel_regularizer=reg)(dense_layer1)
output_layer = tf.keras.layers.Dense(2, activation='softmax')(dense_layer2)
```

Fig. 34. Figure showing code snippet used create similarity network.



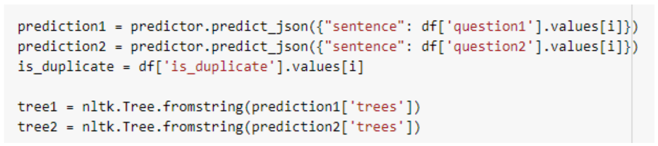
```
# Pack it all up into a model
model = tf.keras.models.Model([left_input, right_input], [output_layer])
# Nadam optimizer, with gradient clipping by norm
optimizer = tf.keras.optimizers.Nadam(lr=lr, decay=decay, clipnorm=grad_clip_norm)
model.compile(loss=loss, optimizer=optimizer, metrics=['accuracy'])
```

Fig. 35. Figure showing code snippet used to compile the model.



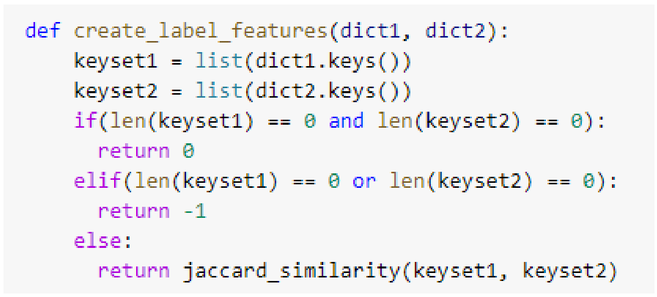
```
archive = load_archive("https://s3-us-west-2.amazonaws.com/allennlp/models/elmo-constituency-parser-2018.03.14.tar.gz")
predictor = Predictor.from_archive(archive, 'constituency-parser')
```

Fig. 36. Figure showing code snipped to load AllenNLP constituent parsing model.



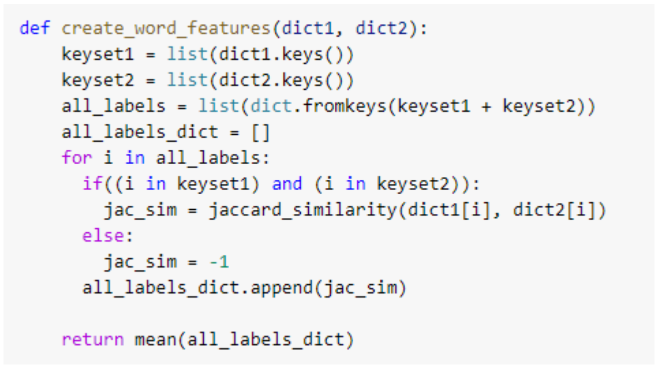
```
prediction1 = predictor.predict_json({"sentence": df['question1'].values[i]})
prediction2 = predictor.predict_json({"sentence": df['question2'].values[i]})
is_duplicate = df['is_duplicate'].values[i]

tree1 = nltk.Tree.fromstring(prediction1['trees'])
tree2 = nltk.Tree.fromstring(prediction2['trees'])
```

Fig. 37. Figure showing code snipped to create constituent trees for each question.



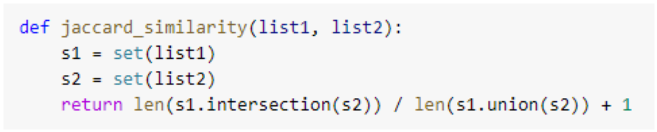
```python
def create_label_features(dict1, dict2):
    keyset1 = list(dict1.keys())
    keyset2 = list(dict2.keys())
    if(len(keyset1) == 0 and len(keyset2) == 0):
      return 0
    elif(len(keyset1) == 0 or len(keyset2) == 0):
      return -1
    else:
      return jaccard_similarity(keyset1, keyset2)
```

Fig. 38. Figure showing code snipped to create label features for each question pair.



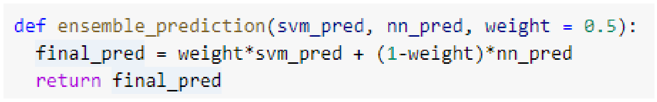
```python
def create_word_features(dict1, dict2):
    keyset1 = list(dict1.keys())
    keyset2 = list(dict2.keys())
    all_labels = list(dict.fromkeys(keyset1 + keyset2))
    all_labels_dict = []
    for i in all_labels:
      if((i in keyset1) and (i in keyset2)):
        jac_sim = jaccard_similarity(dict1[i], dict2[i])
      else:
        jac_sim = -1
      all_labels_dict.append(jac_sim)

    return mean(all_labels_dict)
```

Fig. 39. Figure showing code snipped to create word features for each question pair.



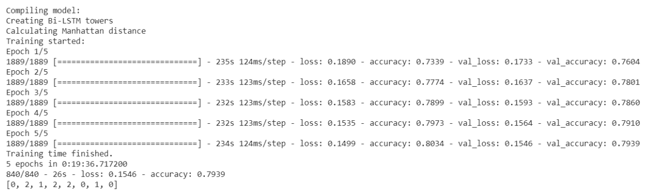
```python
def jaccard_similarity(list1, list2):
    s1 = set(list1)
    s2 = set(list2)
    return len(s1.intersection(s2)) / len(s1.union(s2)) + 1
```

Fig. 40. Figure showing code snipped to calculate Jaccard Similarly.



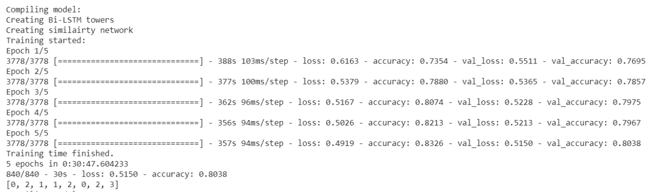
```python
def ensemble_prediction(svm_pred, nn_pred, weight = 0.5):
  final_pred = weight*svm_pred + (1-weight)*nn_pred
  return final_pred
```

Fig. 41. Figure showing code snipped to calculate final prediction.



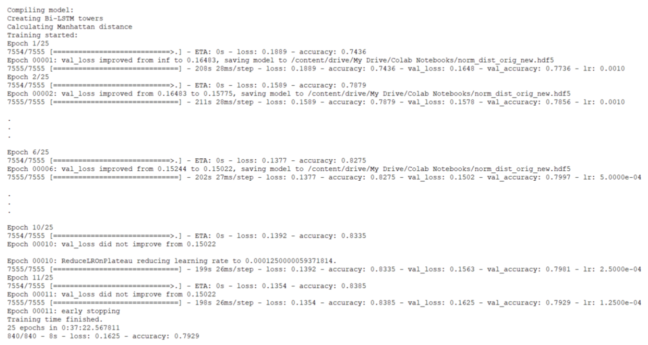
```
Compiling model:
Creating Bi-LSTM towers
Calculating Manhattan distance
Training started:
Epoch 1/5
1889/1889 [==============================] - 235s 124ms/step - loss: 0.1890 - accuracy: 0.7339 - val_loss: 0.1733 - val_accuracy: 0.7604
Epoch 2/5
1889/1889 [==============================] - 233s 123ms/step - loss: 0.1658 - accuracy: 0.7774 - val_loss: 0.1637 - val_accuracy: 0.7801
Epoch 3/5
1889/1889 [==============================] - 232s 123ms/step - loss: 0.1583 - accuracy: 0.7899 - val_loss: 0.1593 - val_accuracy: 0.7860
Epoch 4/5
1889/1889 [==============================] - 232s 123ms/step - loss: 0.1535 - accuracy: 0.7973 - val_loss: 0.1564 - val_accuracy: 0.7910
Epoch 5/5
1889/1889 [==============================] - 234s 124ms/step - loss: 0.1499 - accuracy: 0.8034 - val_loss: 0.1546 - val_accuracy: 0.7939
Training time finished.
5 epochs in 0:19:36.717200
840/840 - 26s - loss: 0.1546 - accuracy: 0.7939
[0, 2, 1, 2, 2, 0, 1, 0]
```

Fig. 42. Figure showing best Si-Bi-MaLSTM model trained during hyper parameter tuning.



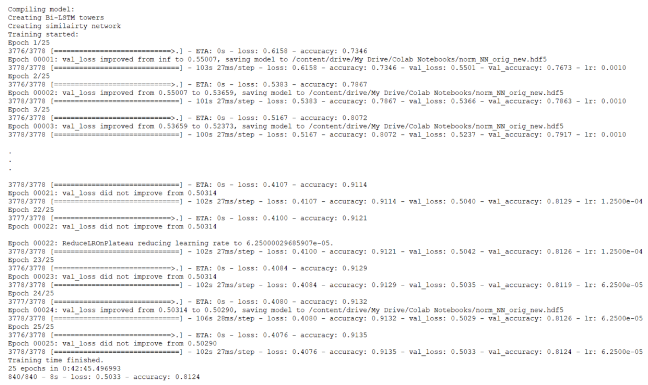
```
Compiling model:
Creating Bi-LSTM towers
Creating similairty network
Training started:
Epoch 1/5
3778/3778 [==============================] - 388s 103ms/step - loss: 0.6163 - accuracy: 0.7354 - val_loss: 0.5511 - val_accuracy: 0.7695
Epoch 2/5
3778/3778 [==============================] - 377s 100ms/step - loss: 0.5379 - accuracy: 0.7880 - val_loss: 0.5365 - val_accuracy: 0.7857
Epoch 3/5
3778/3778 [==============================] - 362s 96ms/step - loss: 0.5167 - accuracy: 0.8074 - val_loss: 0.5228 - val_accuracy: 0.7975
Epoch 4/5
3778/3778 [==============================] - 356s 94ms/step - loss: 0.5026 - accuracy: 0.8213 - val_loss: 0.5213 - val_accuracy: 0.7967
Epoch 5/5
3778/3778 [==============================] - 357s 94ms/step - loss: 0.4919 - accuracy: 0.8326 - val_loss: 0.5150 - val_accuracy: 0.8038
Training time finished.
5 epochs in 0:30:47.604233
840/840 - 30s - loss: 0.5150 - accuracy: 0.8038
[0, 2, 1, 1, 2, 0, 2, 3]
```

Fig. 43. Figure showing best Si-Bi-FFNLSTM model trained during hyper parameter tuning.



```
Compiling model:
Creating Bi-LSTM towers
Calculating Manhattan distance
Training started:
Epoch 1/25
7554/7555 [============================>.] - ETA: 0s - loss: 0.1889 - accuracy: 0.7436
Epoch 00001: val_loss improved from inf to 0.16483, saving model to /content/drive/My Drive/Colab Notebooks/norm_dist_orig_new.hdf5
7555/7555 [==============================] - 208s 28ms/step - loss: 0.1889 - accuracy: 0.7436 - val_loss: 0.1648 - val_accuracy: 0.7736 - lr: 0.0010
Epoch 2/25
7554/7555 [============================>.] - ETA: 0s - loss: 0.1589 - accuracy: 0.7879
Epoch 00002: val_loss improved from 0.16483 to 0.15775, saving model to /content/drive/My Drive/Colab Notebooks/norm_dist_orig_new.hdf5
7555/7555 [==============================] - 211s 28ms/step - loss: 0.1589 - accuracy: 0.7879 - val_loss: 0.1578 - val_accuracy: 0.7856 - lr: 0.0010

.
.
.

Epoch 6/25
7554/7555 [============================>.] - ETA: 0s - loss: 0.1377 - accuracy: 0.8275
Epoch 00006: val_loss improved from 0.15244 to 0.15022, saving model to /content/drive/My Drive/Colab Notebooks/norm_dist_orig_new.hdf5
7555/7555 [==============================] - 202s 27ms/step - loss: 0.1377 - accuracy: 0.8275 - val_loss: 0.1502 - val_accuracy: 0.7997 - lr: 5.0000e-04

.
.
.

Epoch 10/25
7554/7555 [============================>.] - ETA: 0s - loss: 0.1392 - accuracy: 0.8335
Epoch 00010: val_loss did not improve from 0.15022

Epoch 00010: ReduceLROnPlateau reducing learning rate to 0.0001250000059371814.
7555/7555 [==============================] - 199s 26ms/step - loss: 0.1392 - accuracy: 0.8335 - val_loss: 0.1563 - val_accuracy: 0.7981 - lr: 2.5000e-04
Epoch 11/25
7554/7555 [============================>.] - ETA: 0s - loss: 0.1354 - accuracy: 0.8385
Epoch 00011: val_loss did not improve from 0.15022
7555/7555 [==============================] - 198s 26ms/step - loss: 0.1354 - accuracy: 0.8385 - val_loss: 0.1625 - val_accuracy: 0.7929 - lr: 1.2500e-04
Epoch 00011: early stopping
Training time finished.
25 epochs in 0:37:22.567811
840/840 - 8s - loss: 0.1625 - accuracy: 0.7929
```

Fig. 44. Figure showing training epochs of Si-BiMaLSTM model.



```
Compiling model:
Creating Bi-LSTM towers
Creating similairty network
Training started:
Epoch 1/25
3776/3778 [============================>.] - ETA: 0s - loss: 0.6158 - accuracy: 0.7346
Epoch 00001: val_loss improved from inf to 0.55007, saving model to /content/drive/My Drive/Colab Notebooks/norm_NN_orig_new.hdf5
3778/3778 [==============================] - 103s 27ms/step - loss: 0.6158 - accuracy: 0.7346 - val_loss: 0.5501 - val_accuracy: 0.7673 - lr: 0.0010
Epoch 2/25
3776/3778 [============================>.] - ETA: 0s - loss: 0.5383 - accuracy: 0.7867
Epoch 00002: val_loss improved from 0.55007 to 0.53659, saving model to /content/drive/My Drive/Colab Notebooks/norm_NN_orig_new.hdf5
3778/3778 [==============================] - 101s 27ms/step - loss: 0.5383 - accuracy: 0.7867 - val_loss: 0.5366 - val_accuracy: 0.7863 - lr: 0.0010
Epoch 3/25
3776/3778 [============================>.] - ETA: 0s - loss: 0.5167 - accuracy: 0.8072
Epoch 00003: val_loss improved from 0.53659 to 0.52373, saving model to /content/drive/My Drive/Colab Notebooks/norm_NN_orig_new.hdf5
3778/3778 [==============================] - 100s 27ms/step - loss: 0.5167 - accuracy: 0.8072 - val_loss: 0.5237 - val_accuracy: 0.7917 - lr: 0.0010

.
.
.

3778/3778 [==============================] - ETA: 0s - loss: 0.4107 - accuracy: 0.9114
Epoch 00021: val_loss did not improve from 0.50314
3778/3778 [==============================] - 102s 27ms/step - loss: 0.4107 - accuracy: 0.9114 - val_loss: 0.5040 - val_accuracy: 0.8129 - lr: 1.2500e-04
Epoch 22/25
3777/3778 [============================>.] - ETA: 0s - loss: 0.4100 - accuracy: 0.9121
Epoch 00022: val_loss did not improve from 0.50314

Epoch 00022: ReduceLROnPlateau reducing learning rate to 6.25000029685907e-05.
3778/3778 [==============================] - 102s 27ms/step - loss: 0.4100 - accuracy: 0.9121 - val_loss: 0.5042 - val_accuracy: 0.8126 - lr: 1.2500e-04
Epoch 23/25
3776/3778 [============================>.] - ETA: 0s - loss: 0.4084 - accuracy: 0.9129
Epoch 00023: val_loss did not improve from 0.50314
3778/3778 [==============================] - 102s 27ms/step - loss: 0.4084 - accuracy: 0.9129 - val_loss: 0.5035 - val_accuracy: 0.8119 - lr: 6.2500e-05
Epoch 24/25
3777/3778 [============================>.] - ETA: 0s - loss: 0.4080 - accuracy: 0.9132
Epoch 00024: val_loss improved from 0.50314 to 0.50290, saving model to /content/drive/My Drive/Colab Notebooks/norm_NN_orig_new.hdf5
3778/3778 [==============================] - 106s 28ms/step - loss: 0.4080 - accuracy: 0.9132 - val_loss: 0.5029 - val_accuracy: 0.8126 - lr: 6.2500e-05
Epoch 25/25
3776/3778 [============================>.] - ETA: 0s - loss: 0.4076 - accuracy: 0.9135
Epoch 00025: val_loss did not improve from 0.50290
3778/3778 [==============================] - 102s 27ms/step - loss: 0.4076 - accuracy: 0.9135 - val_loss: 0.5033 - val_accuracy: 0.8124 - lr: 6.2500e-05
Training time finished.
25 epochs in 0:42:45.496993
840/840 - 8s - loss: 0.5033 - accuracy: 0.8124
```

Fig. 45. Figure showing training epochs of Si-BiFFNLSTM model.

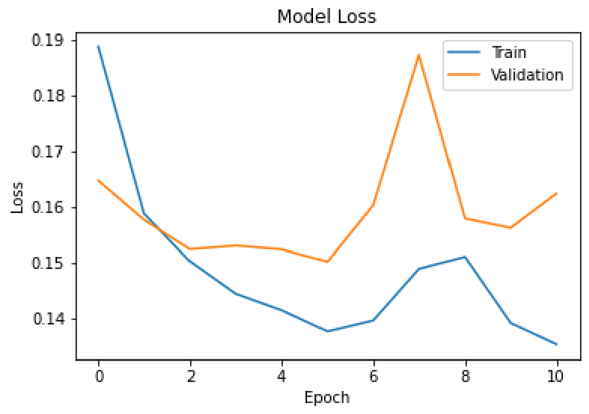


Fig. 46. Figure showing validation and training loss achieved by Si-Bi-MaLSTM model during training epochs.

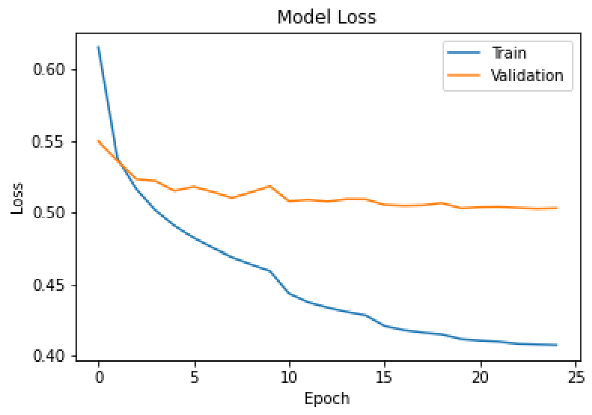


Fig. 47. Figure showing validation and training loss achieved by Si-Bi-FFNLSTM model during training epochs.

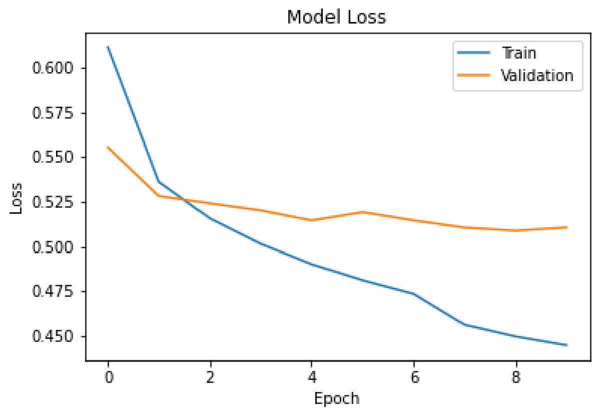


Fig. 48. Figure showing validation and training loss achieved by Si-Bi-FFNLSTM model during training epochs, with a more strict stopping criteria.