Course Project: News Stance Detection

Shivam Dhall - 13033974

1 Data loading

When loading both the training and testing datasets, we extract all unique headlines and store them as values in a dictionary that are referenced by unique *headline id's*; the same procedure is repeated for the article bodies. We then maintain a list of training and testing examples, where each example consists of a *stance* a *headline id* and a *body id*. Storing the headlines and bodies separately from the examples allows us to process/vectorise each unique headline and body a single time only – this drastically improves the computational efficiency during feature-generation.

2 Train-Validation Split

In order to ensure that the training and validation subsets have similar ratios of the four classes, we perform the split for each class separately as follows: We first filter the training examples by class label, we then count the total number of examples contained within that class – this count is then multiplied by 0.9 to get the number of examples that should be added to the training set, the remainder of the examples of that class are added to the validation set. This process is repeated for all class labels. Once completed, the training and validation subsets will be ordered by class, we therefore randomly shuffle each subset – this is shown to improve training performance of ML algorithms. The figures below represent the statistics for each class after the split is performed:

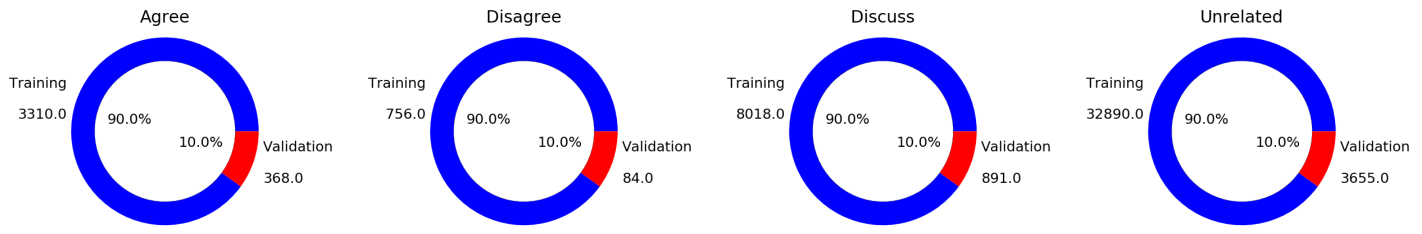


Figure 1: Train-Validation statistics for each class

3 Preprocessing of Headlines and Bodies

Before generating vector representations, we first preprocess each unique headline and body this is done as follows: We first convert the string to a lower case representation, we then use a regular expression to tokenise the string (tokens consist of all alpha-numeric characters that are separated by white space or punctuation, however, sequences of characters separated by a hyphen are considered to be a single token). We then perform stemming and lemmatization, this involves passing each token through a lemmatizer that only removes affixes if the resulting word (stem) is contained within the WordNet dictionary. The final stage of preprocessing involves the extraction of stop words from the list of tokens, to do this we use a database that consists of 318 unique stop words.

4 Vector Representation of Headlines and Bodies

In our implementation, we generate vector representations of the headlines and bodies using a bag-of-words (BOW) methodology. Since we use a BOW, it is first necessary to generate a vocabulary – we do this using all the preprocessed headlines and bodies that are contained within the training subset only. This gives us a vocabulary that contains 21324 unique words. Using this vocabulary, we then generate a term-frequency (tf) vector for each unique headline and body contained within the training, validation and testing sets (Note, the tf is simply based on the raw count of each word contained with the headline/body). As a result, all headlines and bodies are represented individually as 21324 dimensional vectors. We then calculate the term frequency-inverse document frequency (tf-idf) vector representations for each document and headline. We do this because tf-idf takes into consideration the number of headlines/bodies a word occurs in, hence commonly occurring words which are likely to be of no importance are given a small weighting. We then use these tf-idf vectors to calculate the cosine similarity between the the headlines and the bodies. Below is the formula used to calculate the idf of a token t with smoothing:

$$idf(t) = log\left(1 + \frac{N+1}{n_t+1}\right) \tag{1}$$

Where N is the total number of unique headlines and bodies in the training set, and n_t is the number of unique headlines and bodies that term t occurs in. The cosine similarity between tf-idf vectors $\mathbf{a}$ and $\mathbf{b}$ can be calculated as follows:

$$cosine\_sim(a,b) = \frac{\mathbf{a}\cdot\mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \tag{2}$$

5 KL-Divergence

In order to calculate the KL-divergence between a headline and a body, we first construct a language model for each. While constructing the language models, we incorporate Dirichlet smoothing as it provides an estimate for the probabilities of missing words. The language model for a headline can be constructed as follows:

$$(M_H) = p(w|H) = \frac{N_H}{N_H + \mu}\frac{f_{(w,H)}}{N_H} + \frac{\mu}{N_H + \mu}\frac{f_{(w,C)}}{N_C} \tag{3}$$

Where N_H is the number of tokens in the headline, N_C is the number of tokens in the corpus (all unique headlines and bodies), $f_{(w,H)}$ is the frequency of word w in the Headline, $f_{(w,C)}$ is the frequency of word w in the corpus and μ is a constant parameter that controls the extent of smoothing. Once a language model has been constructed for each headline and body, we can then calculate the KL-divergence between a headline and a body as follows:

$$D(M_{Headline}||M_{Body}) = \sum_{w \in V} p(w|Headline)\ log\frac{p(w|Headline)}{p(w|Body)} \tag{4}$$

6 Additional Features/Distances

The first feature we generated is the Jaccard index between the headlines and the bodies, this is a statistic that is commonly used for comparing the similarity and diversity between finite sample sets. The Jaccard index between a headline and a body is computed as the size of the intersection of unique terms contained in the headline and body, divided by the size of the union of unique terms contained in the headline and body. We can therefore compute the Jaccard index between a binarised tf vector representation of a headline **h** and a body **b** as follows:

$$J(\mathbf{h}, \mathbf{b}) = \frac{\mathbf{h} \cdot \mathbf{b}}{\sum_i \mathbf{h}_i + \sum_j \mathbf{b}_j - (\mathbf{h} \cdot \mathbf{b})} \tag{5}$$

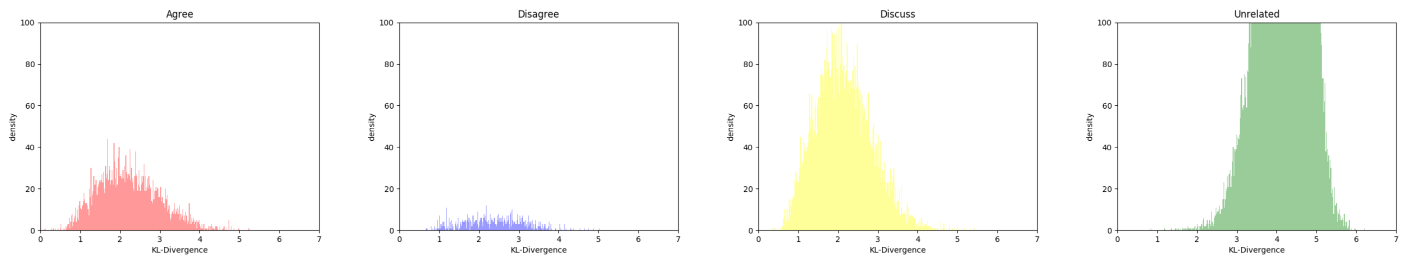
The next feature we generated is the absolute difference between the normalized count of the number of negative tokens that are present within a headline, and the normalized count of the number of negative tokens that are present within the corresponding body pair. This feature has shown to be successful in previous literature for discriminating between agree and disagree labels, if the body and the headline contain a similar proportion of negative tokens, then the feature has a value close to 0 in which case the label is more likely to be agree. On the other hand, if the body contains a higher proportion of negative tokens as compared to the headline (or vice-versa), then the feature will have a value closer to 1 in which case the label is more likely to be disagree. In order to calculate this feature, we use a database that consists of 4783 unique negative tokens, we then create a BOW vector representation of these tokens using our vocabulary – this gives us a binary vector **m** of size 21324. The feature can then be calculated as follows:

$$F(\mathbf{h}, \mathbf{b}) = \left| \frac{\mathbf{h} \cdot \mathbf{m}}{\sum_i \mathbf{h}_i} - \frac{\mathbf{b} \cdot \mathbf{m}}{\sum_j \mathbf{b}_j} \right| \tag{6}$$

Where **h** and **b** are tf vectors of the headline and body respectively.

The final feature/distance we generate is the cosine similarity between vector representations that are generated using latent semantic indexing (LSI), this is a common technique that is used to analyse the relationship between a set of documents and the terms they contain; LSI assumes that words that are close in meaning (e.g synonyms) will occur in similar pieces of text (the distributional hypothesis). In order to perform LSA we first construct a tf-idf matrix where each row represents a unique headline/body that is present in the training set and the columns represent the words in our vocabulary. Singular value decomposition (SVD) is then used to reduce the number of columns (in our implementation, we reduce the number of columns to 100) while preserving the similarity structure among the rows. This rank lowering is thought to merge dimensions associated with terms that have similar meanings; as a result, this gives LSI the ability to work with both *synonyms* and *polysemys*. Each tf-idf vector representation of a headline or a body can then be cast into this low rank vector representation of size 100, the cosine similarity can then be computed between 2 such vectors (as in equation 2).

7 Distibution of Features Across Classes

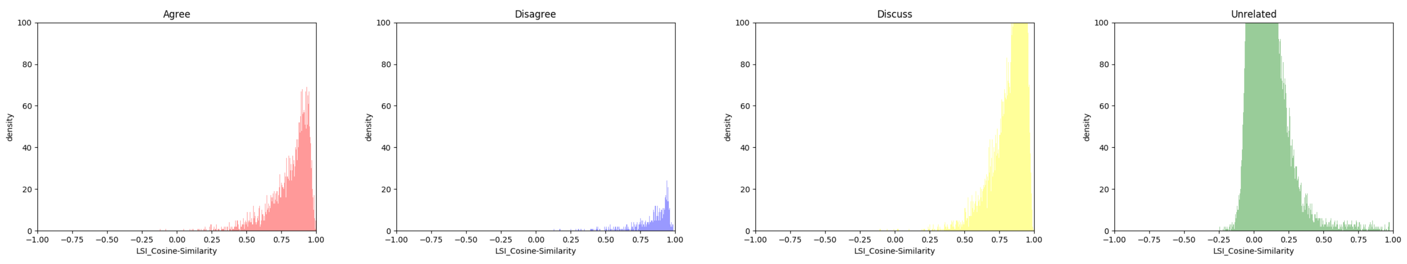


(a) Agree (b) Disagree (c) Discuss (d) Unrelated

Figure 2: Histograms representing the distribution of the KL-divergence feature across all classes

I believe KL-Divergence is an important feature because it first estimates a language model for both the headline and the corresponding body, it then directly compares the similarity between these two language models. As a result KL-divergence combines the advantages of both a headline-likelihood model $p(Head|M_{Body})$ and a body-likelihood model $p(Body|M_{Head})$.

Language models consider how likely a given string is in a given language, hence they are generally more expressive as compared to vector space models. From the histograms above, it can be seen that there is a clear difference between the mean of the distribution for the unrelated class (approx 4.25) as compared to all other classes (approx 2). Hence this feature is extremely useful when predicting if a headline and a body are unrelated, however it may not be as useful for differentiating between the other three classes.



(a) Agree (b) Disagree (c) Discuss (d) Unrelated

Figure 3: Histograms representing the distribution of the LSI cosine similarity across all classes

I believe LSI cosine similarity is a good feature to use because the vector representations generated by LSI are much more expressive as compared to a simple tf vector representation that is based on a BOW. This is because the LSA vector representation has a lower rank (100 as opposed to 21324), this rank lowering merges dimensions associated with terms that have similar meanings; as a result, this gives LSI the ability to work with both *synonyms* and *polysemys* – tf vectorised representation are unable to deal with either of these scenarios. From the histograms above, it can be seen that the mean of the distribution of the unrelated class is visibly lower (approx 0) as compared to all other classes which have a mean that is in the range 0.75 – 1, it can also be seen that the mean of the distribution of the Discuss class is marginally higher as compared to both the Agree and Disagree classes. As a result, this feature is very useful for predicting if a headline and body are unrelated, additionally it can also be used to predict if the label is Discuss as opposed to Agree or Disagree – though this may not be as accurate.

8 Implementation of Linear regression and Logistic Regression Algorithms

In order to perform multi-class classification using liner regression, we develop 4 linear regression models each corresponding to a different class label. For each model, we encode the labels in a binary format, a value of 1 signifies that the label is part of the class that the model is trained to predict, a value of 0 signifies that the label is not part of that class. Each model is then trained individually using batch gradient descent with a MSE loss for 1000 epochs (Note, the implementation in the code is fully vectorised, hence the models are trained simultaneously). Given an input feature vector $\mathbf{x}$ and a one-hot encoded label $\mathbf{y}$, we can compute the predictions of each model as $\mathbf{y}' = (\mathbf{x}^{\mathbf{T}} \cdot \mathbf{W})$ where $\mathbf{W}$ is a matrix of weights where each column contains the the parameters of one of the four linear models – this results in a row vector of size 4, the final prediction is the class corresponding with the index of the maximum value in the resulting vector. In the case of batch gradient descent, below is the formula used to update the weight matrix $\mathbf{W}$:

$$\mathbf{W} = \mathbf{W} - \frac{\alpha}{batch\_size}(\mathbf{X}^{\mathbf{T}} \cdot (\mathbf{Y}' - \mathbf{Y})) \tag{7}$$

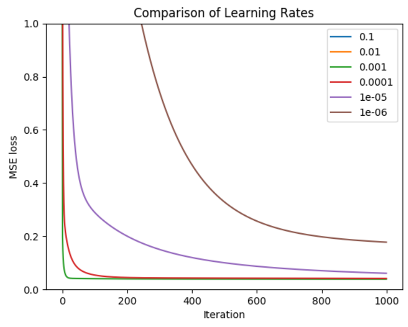
Where α is the learning rate, $\mathbf{X}$ is a batch of inputs, $\mathbf{Y}'$ is a batch of predicted outputs and $\mathbf{Y}$ is a batch of one-hot encoded labels. (Note that the rows of the matrices corresponding to a single example)

In order to perform multi-class classification using logistic regression, we use a one vs all approach which is similar to that used for linear regression. We develop 4 logistic regression models each corresponding to a different class label. Each model is then trained individually using batch gradient descent with a cross-entropy loss for 1000 epochs. Given an input feature vector $\mathbf{x}$ and a one-hot encoded label $\mathbf{y}$, we can compute the predictions of each model as $\mathbf{y}' = sigmoid(-(\mathbf{x}^{\mathbf{T}} \cdot \mathbf{W}))$; once again, this results in a row vector of size 4, the final prediction is the class corresponding with the index of the maximum value in the resulting vector. The update equation for logistic regression remains the same as that for linear regression (equation 7).

9 Analysis of performance of models

We use 3 different metrics to evaluate the performance of a model. The first metric is the overall accuracy of the model, this is simply the number of correct predictions divided by the total number of predictions. The second metric is a weighted mean of the F1 scores of each class. The F1 score of a single class can be calculated as
$F1 = 2 \times (Precision \times Recall)/(Precision + Recall)$ This is a good metric to use because it includes both precision and recall, it also takes class imbalance into account as we compute a weighted mean of the F1 scores associated with each class. Finally, we compute a normalised (we normalise due to class imbalance) confusion matrix of the results generated by each model – this helps identify classes in which the model performs well and poorly.

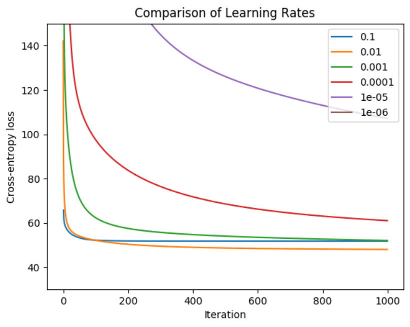
For each model, we try a range of different learning rates, for each learning rate we calculate the average loss after each epoch and plot the results, these can be seen in the figures below:



(a) Linear regression: Learning curves

Learning Rate	Accuracy	F1
0.1	0.736	0.01
0.01	0.736	0.01
0.001	0.897	0.861
0.0001	0.896	0.859
0.00001	0.873	0.834
0.000001	0.818	0.774

(b) Linear regression: Validation Results



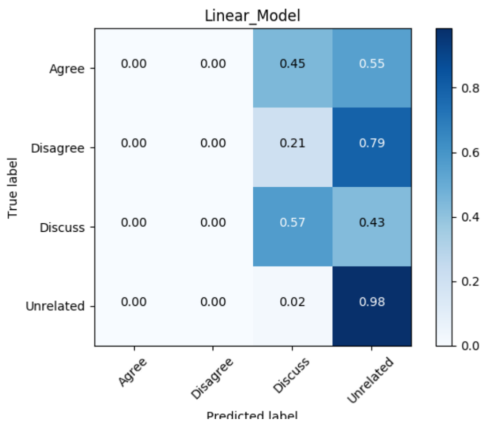
(c) Logistic regression: Learning curves

Learning Rate	Accuracy	F1
0.1	0.890	0.861
0.01	0.896	0.864
0.001	0.895	0.870
0.0001	0.879	0.841
0.00001	0.762	0.658
0.000001	0.677	0.591

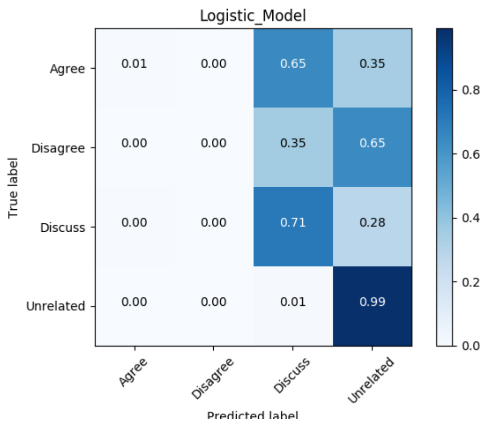
(d) Logistic regression: Validation Results

Figure 4: Comparison of learning rates for linear and logistic regression (Statistics based on validation set)

From the learning curves plotted above as well as the validation statistics, it can be seen that the choice of learning rate has a significant impact on the performance of a model. Based on the validation statistics, we conclude that the best learning rate for both linear and logistic regression is 0.001 – this is because this learning rate yield the highest accuracy and f1 score for both linear and logistic regression. We then evaluate the performance of the models trained using this learning rate on the testing set, below are the results:



(a) Linear Model: Accuracy=0.801, F1=0.760



(b) Linear Model: Accuracy=0.841, F1=0.806

Figure 5: Statistics of linear and logistic model on testing set

From the diagrams above, the logistic model is seen to perform better as compared to the liner model in terms of both overall accuracy and F1 score. From the confusion matrices, it can be seen that the linear model tends to predict the Unrelated label most of the time, in addition, it never predicts the Agree or Disagree label. On the other hand, the logistic regression model sometimes predicts the Agree label, however it still never predicts the Disagree label. The accuracy/recall for the Unrelated, Agree and Discuss classes are also higher for the logistic model.

10 Feature Importance

Feature	Accuracy	F1
TF-IDF Cosine Similarity	0.789	0.699
KL-Divergence	0.791	0.718
Jaccard Index	0.652	0.640
Negative Token Analysis	0.650	0.651
LSI Cosine Similarity	0.799	0.743

(a) Linear regression statistics

Feature	Accuracy	F1
TF-IDF Cosine Similarity	0.826	0.790
KL-Divergence	0.832	0.799
Jaccard Index	0.723	0.609
Negative Token Analysis	0.722	0.605
LSI Cosine Similarity	0.840	0.803

(b) Logistic regression statistics

Figure 6: Feature importance statistics on test set

In order to determine the importance of features, we retrain both the linear and logistic regression models with a single feature for 1000 epochs, we then evaluate the performance of these models on the testing set – the results are presented in the tables above. For both linear and logistic regression, LSI cosine similarity and KL-divergence are seen to be the 2 most important features, the models trained on these features alone have a similar performance to the models trained on all features (Figure 5). On the other hand, the Jaccard index and the Negative token analysis features are the least important – models that are trained on these features have the lowest performance. Overall, the models that are trained on all features are still better as compared to all other models that are trained on a single feature, this suggests that a combination of features is better as compared to any single feature.

11 Literature Review

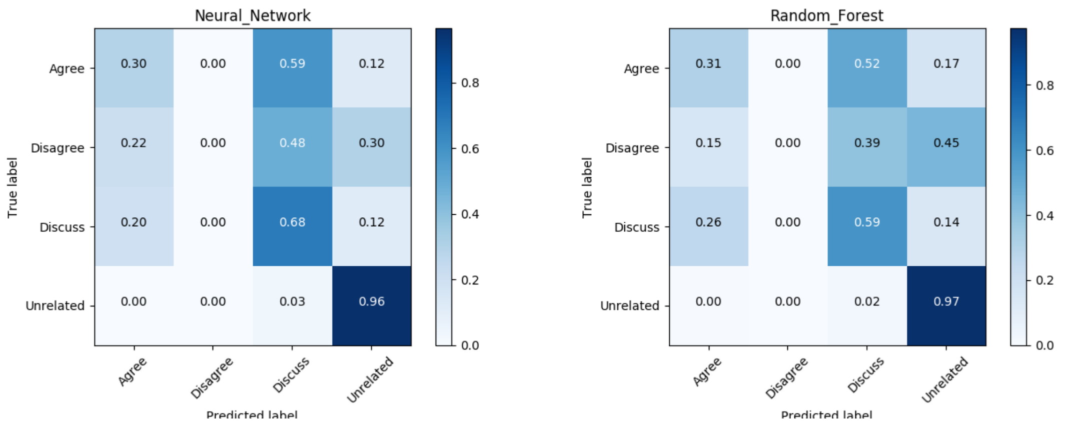
Many implementations of the stance detection task build upon the features that are already provided in the baseline, for example [1], [2], [4] and [5] generate additional features based on word-embeddings that are pre-trained over datasets, these commonly include Google's Word2Vec and Stanford's GloVe embeddings. These embeddings are then averaged across every word contained in a headline/body the resulting vectors are then used to calculate distance metrics such as cosine similarity and Jaccard similarity. Another common feature used in Neural Network models is a simple BOW tf vector representation of the headline and body with no further processing [2][3]. Similarly, it is also common to use the concatenation of the average Word2Vec headline and body embeddings as features [4]. Another prevalent feature is the number of overlapping refuting words between the headline and its corresponding body pair [1][6][4]. Finally, many of these features are repeated for word N-grams (where n = 2,3,4) [1][6].

Many ML algorithms have successfully been used in the literature to generate models with a good performance, such models include SVM's with RBF kernels [5][6] as well as random forest models [4][5]. However, most of the recent papers tend to use deep-learning approaches, more precisely Recurrent Neural Networks (RNN) consisting of Long-Short Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells along with attention mechanisms [2][5][7][8][9]. Such RNNs can use their internal state (similar to memory) to process a sequence of inputs, hence they are capable of exhibiting dynamic temporal behavior for time sequence data such as text. Consequently, such recurrent models take as input a concatenated headline and body pair where each word is represented as a vector using a Word2Vec or BOW embedding [2][5][7][8][9]. Finally, some deep learning approaches implement a conditionally encoded LSTM – such models consist of two separate LSTMs, one for the headline and one for the body. The headline is fed through the first LSTM which generates a hidden state, this hidden state is then used to initialize the LSTM of the body, thus "conditioning" the body LSTM on the headline [2][5][8]. LSTM's such as those described are the current state of the art, and have shown to produce F1 scores greater than 0.86 on the FNC-1 challenge [7].

12 Improvements

The first improvement we make is to standardise all the input features before they are input into the learning algorithm, standardization is a well know technique that has proven to improve the convergence of ML algorithms. Given that there is a high class imbalance in our training dataset, it would be useful to re-sample the dataset to even-up the classes, this can be done by either adding copies of instances from the under represented classes or removing instances from the over represented classes. Another way of dealing with the class imbalance is to impose a penalty/additional cost on the ML models for making classification mistakes on the minority classes during training, it is common to set the class penalties so that they are inversely proportional to class frequencies in the input data (All these techniques are utilised in the models trained below).

Finally, as opposed to using ML models that can only fit linear mappings, it would be worthwhile to explore other ML algorithms which are capable of modeling non-linear mappings, such algorithms include SVM's, Random Forests (RF), and Neural Networks (NN). Below we train a RF model, during training we impose an additional cost that is inversely proportional to the class frequencies. Finally, we implement a NN with 5 hidden layers containing 50, 100, 100, 100, and 50 hidden neurons respectively. Between each layer we add ReLU non-linearities, the output of the final layer is passed though a soft-max layer to get class probabilities. In order to prevent over fitting, we add additional dropout layers between each hidden layer, we also use an adaptive learning rate which reduces by a constant factor each time two consecutive epochs fail to decrease the training loss. Though not implemented, it would be useful to experiment with Recurrent Neural Networks (Either LSTM's or GRU's) with attention mechanisms as these models have shown to produce the best results in the FNC-1 challenge [7].



(a) Neural Netwrk: Accuracy=0.861, F1=0.836 (b) Random Forest: Accuracy=0.843, F1=0.821

Figure 7: Statistics of Neural Network and Random Forest models on testing set

From the statistics and confusion matrices above, both the RF and NN models have a significant performance improvement over the previous models, additionally these models are also able to predict the Agree class label with a greater accuracy as compared to the previously implemented models.

13 References

[1] - Jayanth, R.R., Meghan, D. and Dushyanta, D., FNC-1: Stance Detection
[2] - Chaudhry, A.K., Baker, D. and Thun-Hohenstein, P., Stance Detection for the Fake News Challenge: Identifying Textual Relationships with Deep Neural Nets.
[3] - Benjamin, R., Isabelle, A., George, S. and Sebastien, R., UCL Machine Reading – FNC-1 Submission
[4] - Thorne, J., Chen, M., Myrianthous, G., Pu, J., Wang, X. and Vlachos, A., 2017. Fake news stance detection using stacked ensemble of classifiers. In Proceedings of the 2017 EMNLP Workshop: Natural Language Processing meets Journalism (pp. 80-83).
[5] - Chopra, S., Jain, S. and Sholar, J.M., 2017. Towards Automatic Identification of Fake News: Headline-Article Stance Detection with LSTM Attention Models.
[6] - Gupta, N.A. and Kadam, Y., Stance Detection for the Fake News Challenge.
[7] - Rakholia, N. and Bhargava, S., "Is it true?"–Deep Learning for Stance Detection in.
[8] - Pfohl, S., Triebe, O. and Legros, F., 2017. Stance Detection for the Fake News Challenge with Attention and Conditional Encoding.
[9] - Zeng, Q., Zhou, Q. and Xu, S., Neural Stance Detectors for Fake News Challenge.