Text Classification Challenge

Team Name: Rajjeshwar Ganguly

Team Members:

1. Rajjeshwar Ganguly (Kaggle ID: Rajjeshwar Ganguly, UdeM matricule: 20237511)
2. Vaibhav Jade (Kaggle ID: LeoPanzer, UdeM matricule: 20241412)
3. Luong Thuy Chung (Kaggle ID: Chung Luong, UdeM matricule: 20205807)

Introduction:

In this Kaggle competition, we were provided with a dataset comprising of Tweets with labels corresponding to the type of sentiment conveyed. The sentences/tweets were classified as positive, negative or neutral. Our goal was to develop a model that could perform well on the public leaderboard but also on unseen data. The tasks we performed could broadly be classified under the following categories.

- Pre-processing the data
- Algorithm selection and implementation
- Hyperparameter tuning
- Cross-validation
- Result analysis and submitting final predictions to Kaggle for final evaluation

We observed that an ensemble of LSTMs performed the best at classifying the provided text samples and obtained an accuracy of 82.681% on the private leaderboard. Our worst-performing model was an SVM model with a Kaggle private accuracy score of 72.776%. Theoretically, it should have performed close to or better than Naïve Bayes however we ran into memory issues trying to encode our input data using sklearn's TfIdfVectorizer and were constrained to keeping a maximum feature length of 1000 which would explain the lower accuracy. For hyperparameter search, we carried out a manual search on our LSTM model. We tried different model architectures with the general principle of trying to maximize local context for words appearing in each document. There was a class imbalance in our dataset for neutral sentiment data however the number of examples for positive and negative sentiments were balanced.

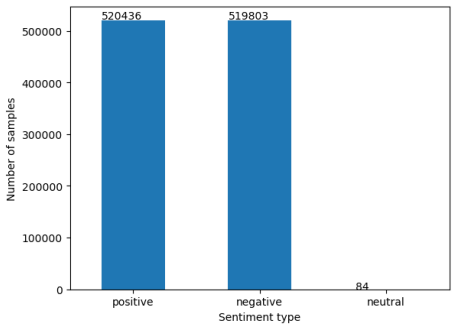

Figure 1. visualizing data distribution

Feature Design and Tokenization:

The dataset comprised of three csv files. This included an unlabeled test set that was used for our final predictions on Kaggle. The train data comprised of _ rows of data, with each row comprising of English sentences/phrases. The 'target' column comprised mainly of two labels, 'positive' and 'negative' while a handful of examples had the 'neutral' label.

	id	text	target
0	0	Anyway Im getting of for a while	positive
1	1	My red, Apache isn't feelin too well this morn...	negative
2	2	@danyelljoy you should be its great. friday w...	positive
3	3	its 11:30pm and i dont wanna sleep; so i debat...	positive
4	4	Why does twitter eat my DM's? Not happy	negative
...	...	...	...
1040318	1040318	getting ready 2 watch mental	positive
1040319	1040319	Wristcutters and Half Nelson are on Sundance F...	negative
1040320	1040320	@t_isfortammy Going out in Northbridge makes m...	negative
1040321	1040321	@IModel_ lol.... Gorgeous...lol! U look sooo m...	positive
1040322	1040322	@iamjonathancook why?	negative

1040323 rows × 3 columns

Figure 2. Original dataframe

	id	text	target
0	0	[anyway, im, get, of, for, a, while]	2
1	1	[my, red, isnt, feelin, too, well, this, morning]	0
2	2	[you, should, be, it, great, friday, will, be,...	2
3	3	[it, pm, and, i, dont, wanna, sleep, so, i, wi...	2
4	4	[why, do, twitter, eat, my, dm, not, happy]	0
...	...	...	...
1040318	1040318	[get, ready, watch, mental]	2
1040319	1040319	[and, half, be, on, free, movie, on, demand, t...	0
1040320	1040320	[go, out, in, make, me, jealous, of, you, im, ...	0
1040321	1040321	[lol, u, look, sooo, much, like, that, vampire...	2
1040322	1040322	[why]	0

1040323 rows × 3 columns

Figure 3. Processed dataframe

Data Cleanup:

We started off by converting all the words in our corpus to lower case alphabets. An important observation we made on the raw data was that many of the sentences were either incomplete, had special characters devoid of any contextual meaning for our classification task (hashtags, mentions) or contained URLs. None of these contributed to an underlying sentiment in most cases, therefore we wanted to get rid of these in our data cleanup process so that our models would not get influenced by random noise. Finally, we removed punctuations from the corpus.

POS Tagging:

A part of speech tagging was performed to aide in lemmatizing our data more effectively in the next step. For this we utilized the Wordnet dictionary of words.

Lemmatization:

Followed by data cleaning we decided to lemmatize our dataset to normalize the words in it. This served two purposes, it helped us reduce the dimensions of our vocabulary while also removing different inflection forms or derivations of the same word. We decided to perform lemmatization instead of stemming as unlike stemming which is simply a heuristic process that is prone to more mistake, lemmatization takes into consideration the morphological context of a word.

Stop Word Removal (high frequency words):

Following lemmatization, we removed stop words from our data. These included high frequency words that are insignificant as standalone words (without context of the entire sentence). We used the NLTK library stop word list for this performing this task. It is important to note that we only removed high frequency words for training our naïve classifiers but did not remove them for training more sophisticated algorithms like neural networks (dense networks, LSTMs, GRUs).

Stop Word Removal (low frequency words):

On the other hand, given that our dataset was obtained from a social media platform it contained a lot of misspelt words. Thus, we wanted to remove this added noise as well. On plotting the list of words in our current vocabulary after we found there were _ words that appeared in our entire dataset less than 100 times. This would imply that their occurrence in the dataset was less than 0.01%. We performed this step for all our models as we thought it was necessary to remove this noise. It also aided in reducing the size of our vocabulary that we needed to encode to train our models on.

Algorithm:

Multinomial Naïve Bayes:

Naïve Bayes is a linear classifier derived from the Bayes classifier therefore they rely on the Bayes theorem but with the assumption that the features of the input data are independent and identically distributed (i.i.d). They work by estimating the maximum likelihood of a given distribution by predicting labels and reducing the resulting loss.

Support Vector Machine (SVM):

SVMs are a class of linear classifiers that work by fitting a hyperplane between two classes of the dataset. The objective is to maximize the minimum distance between the support vectors of either class. Ie: by maximizing the margin of separation. In our case we used a kernelized SVM with the RBF kernel. A kernelized SVM implementation helps classify nonlinear data by changing the feature space such that the data can effectively be classified with the help of a hyperplane.

Fully connected neural network:

Neural networks comprise of an input layer, an output layer and a specified number of hidden layers. Each hidden layer has a predefined number of hidden units or neurons, and these have an associated weight and bias and an activation function. A neural network works by computing a linear combination of the inputs, weights and biases followed by passing it through an activation function depending on a threshold. The predicted output of the neural network is then compared with the actual data label to compute a loss which is iteratively minimized using gradient descent.

Long Short-Term Memory (LSTM):

LSTMs are neural networks that can process long sequences of data. They are recurrent networks that are able to keep short term memory of an input sequence representation for multiple, often thousands of timesteps. LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. They solve the vanishing gradient problem that RNNs are prone to.

Gated Recurrent Unit (GRU):

GRUs are also a type of recurrent neural network, like LSTMs they also have gates however they only have a reset gate and an output gate. They are usually easier to train that LSTM models as they have fewer parameters. Often due to this they are used in situations where the input data is much smaller as they do not overfit due to high capacity.

Convolutional Neural Network (1D):

1D convolutions work on the same principle as 2D and 3D convolution operations, a kernel of fixed window size slides over the input and performs a convolution operation to get a spatial representation of the input data. They are used most to represent text and signal data.

RoBERTa:

RoBERTa (short for "Robustly Optimized BERT Pretraining Approach") is a variant of the BERT (Bidirectional Encoder Representations from Transformers) language model. It is a transformer-based model trained using a combination of supervised and unsupervised learning techniques. RoBERTa modifies key hyperparameters, removing the BERT's next-sentence pretraining objective and training with much larger mini-batches and learning rates [1].

Methodology:

We tried a variety of approaches to solving the classification task and below are the different models that we tried:

a. Multinomial Naïve Bayes with Bag of Words encoding
b. RBF kernelized SVM with TF-IDF encoding
c. Fully connected neural network with Word2Vec embedding
d. GRU with Word2Vec embedding
e. LSTM with Word2Vec embedding
f. Bi-LSTM with Word2Vec embedding
g. Ensemble of CNN-LSTMs with Word2Vec embedding
h. RoBERTa (trained from scratch, no pretrained weights were used)

For encoding our dataset for training and validation we used the following three strategies:

Bag Of Words: We used a bag of words encoding to represent our input data for our naïve bayes classifier. To perform this, we used the sklearn library and the CountVectorizer class to generate vectors of the same length as our final vocabulary size of _. While creating our bag of words encoding for our dataset, we ran into a memory constraint issue as the length of our vectors were too big and thus, they resulted in large sparse vectors which wouldn't fit in memory. Even though scipy provides support for working with

large sparse vectors by storing them as objects instead of numpy arrays, sklearn's implementation of Naïve Bayes does not support scipy sparse objects. Since the bag of words encodings only comprise of integers, we decided to convert the encodings to int8 format to fir out memory constraint.

TF-IDF: For our SVM model we wanted to implement something that captured more meaning from our textual data therefore we used the TF-IDF encoding strategy. This was primarily motivated by our choice to capture the similarity between words in some way. Although naïve and lacking semantic context we chose TF-IDF since it could be an effective way to represent cosine similarity between words in our corpus. For this we used sklearn's TfidfVectorizer. Once again, we faced memory constraint problems owing to the large sparse vectors of our word encodings since TF-IDF encodings work the same way as our earlier bag of words encoding, with the exception that the values are of the python float datatype. Since reducing memory usage by changing a lighter datatype wasn't feasible here, we decided to limit the length of each vector to a length of 1000 for training our SVM.

Word2Vec: We used the Gensim library to train Word2Vec embeddings for our vocabulary. The intuition behind this was to use a more elaborate encoding scheme that would allow us to capture semantic meaning of words. We decided to go for word2vec over other choices like Glove as we wanted our models to capture more local context since social media comments/tweets usually comprise of short sentences and their contextual meaning is often prone to a lot of variances between different users.

Word2Vec typically has two different algorithms to encode words by, namely Continuous Bag Of Words (CBOW) and Skip-gram. Here since our dataset was small, we decided to go for the skip-gram approach to encoding. We wanted to get a better representation of less frequent words that occurred in the data. For eg: words that are common typos when typing on mobile devices, ie: they occurred rarely but frequent enough to not be dismissed as noise.

We decided to keep the length of individual embedding vectors to 100. During training our neural network-based models (dense, LSTM, GRU) we passed this embedding directly into our models embedding layer with the parameter 'trainable' set to false for this player as we had already trained the embedding using the Word2Vec algorithm.

Sequence transformation and padding: Once we had completed setting up our embedding layer utilizing the Word2Vec algorithm we needed to represent our input documents. For this we effectively needed to create a lookup for our embedding layer so that each word from each document would get translated into a vector representation of size 100 on tallying with the embedding layer when training our network. To do this we used the Keras text_to_sequences() function. However, since each document had a different length, we standardized the length of the input data (document sequences) by padding the documents. We padded toward the right as according to Keras documentation right padding is more efficient for their internal optimizations when running recurrent/LSTM layers. Each document sequence thus comprised of the appropriate indices of words for the embedding layer lookup alongside zero padding.

Train-val split: For cross validating our data we created a separate validation set which was used to detect potential overfitting. To do this we made use of the train_test_split() function. We used a stratified train-validation split to make sure our validation and train sets had the same distribution of data for each class.

Model choices: We tried three different approaches to solving this classification problem. It is widely known that when it comes to classification problems in the domain of natural language some of the best SoTA models/architectures are derived from recurrent networks like RNNs, GRUs, LSTMs and Transformer models. The approaches we tried, and the design choices are briefly discussed below:

1. **GRU:** Since our dataset is quite small and GRUs are less complex than LSTMs and easier to train (owing to only two gates as opposed to three for LSTMs and therefore fewer parameters) we decided to start experimentation with a model utilizing GRUs to see how big of an improvement we could get over our previous approaches. For our input we used our previously created word2vec embeddings. Tinkering with the number of hidden units and number of layers and other hyperparameters did not however improve our results by a suitable margin. We found the model to be prone to overfitting as well and therefore tried increasing capacity while maintaining both a spatial dropout as well regular dropout to ensure that we did not overfit on our training set.

2. **LSTM:** For LSTM based approaches we tried using a single LSTM layer with 256 hidden units alongside no dropout as a starting point. Due to its higher performance than our GRU model we tried increasing the contextual capacity of the model by replacing the LSTM layer with a Bidirectional LSTM instead. We found that the performance of the model barely changed. Our

next approach was inspired by the local positional context representation in n-grams (refer comparison under Appendix.). Since we were using our word based Word2Vec embedding utilizing a skip gram algorithm, intuitively it made sense to have some form of n-gram representation for our LSTM layer. Thus, we used a 1D convolution layer to convolve our input vectors effectively encoding spatial information into the input for our LSTM layer. We created an ensemble of 4 LSTM models each identical except for the kernel size of the convolutional layer which ranged from 1 to 4. Upon completion of training, we performed a soft voting on the probabilities obtained for each class to get our final output.

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 50, 100)	905600
conv1d_1 (Conv1D)	(None, 49, 128)	25728
spatial_dropout1d_1 (SpatialDropout1D)	(None, 49, 128)	0
lstm_3 (LSTM)	(None, 49, 128)	131584
lstm_4 (LSTM)	(None, 49, 128)	131584
lstm_5 (LSTM)	(None, 64)	49408
dense_1 (Dense)	(None, 3)	195

Total params: 1,244,099
Trainable params: 338,499
Non-trainable params: 905,600

Figure 4. CNN-LSTM architecture

3. **RoBERTa:** RoBERTa has the same architecture as BERT but uses a byte-level Byte-pair encoding (BPE) as a tokenizer and uses a different pretraining scheme [2]. Thus, we trained the byte-level BPE first which allowed decomposing of all words into tokens (so no more unknown tokens during training). Then we used the trained encoding as input for a task of mask language model (MLM) which predicts how to fill arbitrary tokens that we randomly mask in the dataset [3]. The output of MLM would be used for RoBERTa Model transformer with a sequence classification head on top. Therefore, the whole models were trained from scratch without using pre-trained models. In the original paper, the authors used 2, 4, and 10 epochs on large-scale reading comprehension datasets [4]. Furthermore, the author of the BERT paper recommended only 2-4 training epochs for fine-tuning on a particular NLP task [5]. Since our dataset is not too large, and with limited time and hardware, we choose to train on 3 epochs.

Results:

Even though we got a higher score and cleared the baseline model of 80% accuracy score on the public leaderboard we knew there was room for improvement. The best accuracy we got using this approach was an accuracy score of 82.681% on the final Kaggle leaderboard.

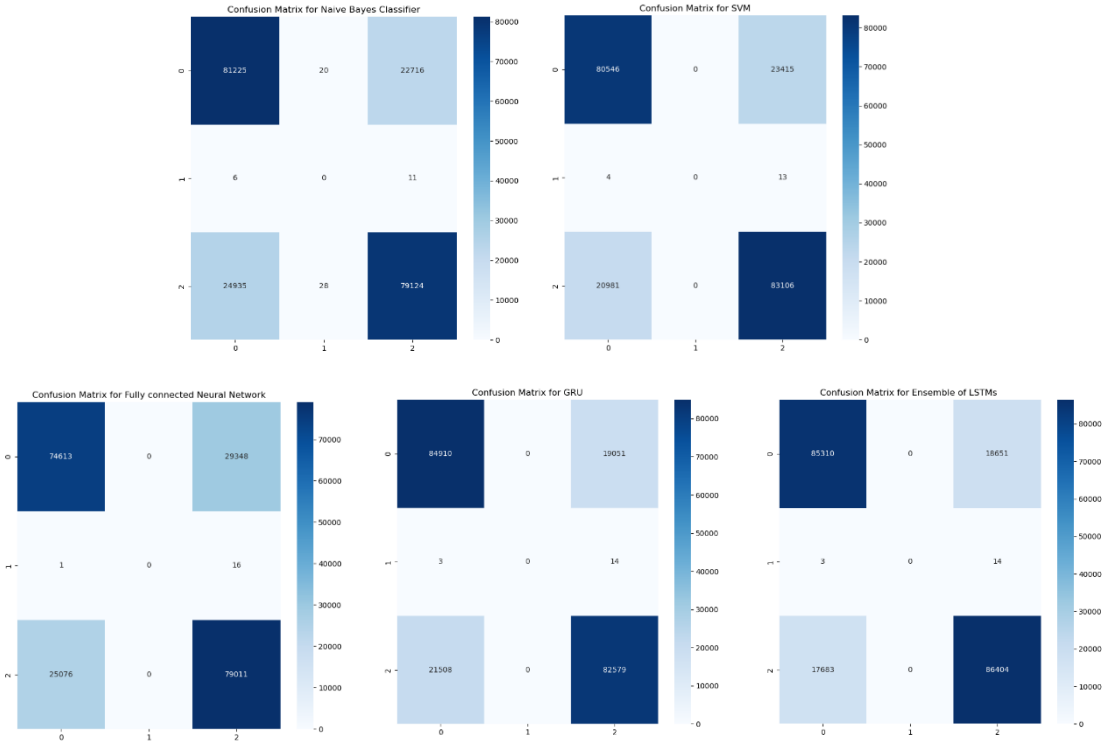

Figure 5. From top left confusion matrices for Naïve Bayes, SVM, NN, GRU, CNN-LSTM ensemble

From the above figures of confusion matrices, The algorithms in their performance order are as follows:

Ensemble of LSTM, GRUs, SVM with TF-IDF encoding, Naïve Bayes and fully connected NN. Although fully connected NN also use word2vec encoding, it does not perform well.

1. Multinomial Naïve Bayes: Naïve Bayes use Maximum A Posteriori (MAP) estimation to calculate the prediction probability. Due to it's feature independence assumption, it does not perform very well. We achieve validation accuracy of 77% and final Kaggle score of 76.94%.

2. Support Vector Machine (SVM): For linear SVM with TF-IDF as word encoding, with dense vector representation, we could fit 1000 feature vector, which resulted in Kaggle entry of 72.776% accuracy. With using sparse vector representation, we could use 90,000-dimensional feature vector, for which the validation accuracy is 79%.

3. Fully connected neural network: We tried a dense Neural Network with 2 hidden layers. On Kaggle leaderboard, we achieve 73.1% accuracy. Following graphs plot the accuracy and loss comparison on train and validation set:

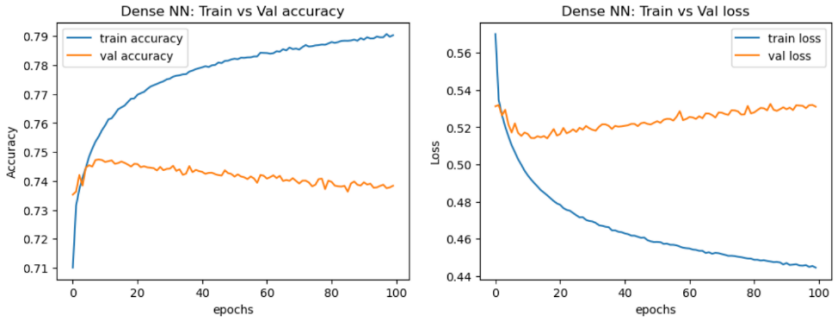

Figure 6. Accuracy and loss for NN

As observed from graph, the model starts overfitting after 20 epochs. We save the model with best validation accuracy as metric to avoid this overfitting.

4. Gated Recurrent Unit (GRU): We train GRU with 1 hidden layer with 256 units. We achieve 80.62% validation accuracy and 80.615% accuracy on Kaggle private leaderboard. Following is train/validation statistics:

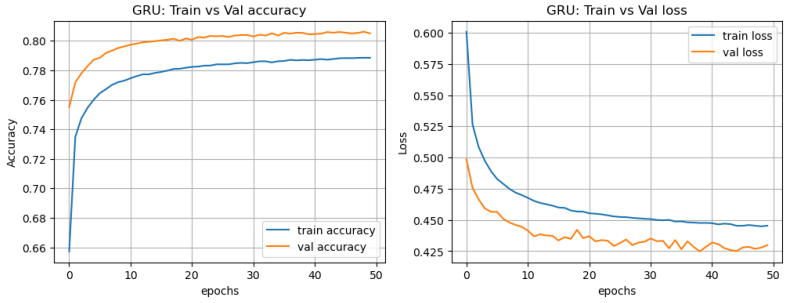

Figure 7. Accuracy and loss for GRU

5. Long Short Term Memory (LSTM): We tried using an LSTM network with just one hidden layer of 256 units and found it to already be surpassing our GRU performance on the validation set. We wanted to investigate this further and tried a variety of network architectures. Counter intuitively using a Bi-LSTM layer in place of the LSTM layer barely changed the accuracy score on the validation set at all. This could be attributed to the overall small size of our dataset.

   Following are train and validation loss and accuracy results for the individual models. We observe that around 20-30 epochs the models start overfitting a bit. While saving, we used Keras checkpoint configured to save based on best validation accuracy, which work as early stopping to avoid overfitting. All models reach validation accuracy in the range of 82%, whereas the ensemble of soft voting achieve best results of 83% validation accuracy.

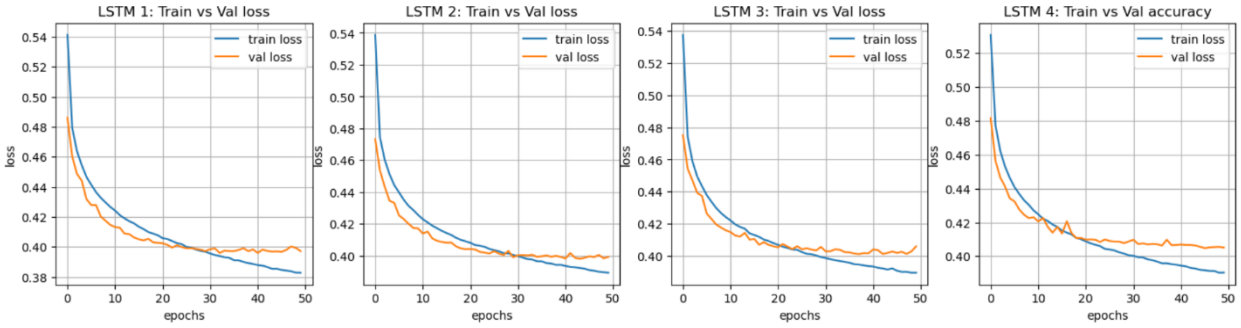

Figure 8. Loss graphs for each CNN-LSTM model of ensemble

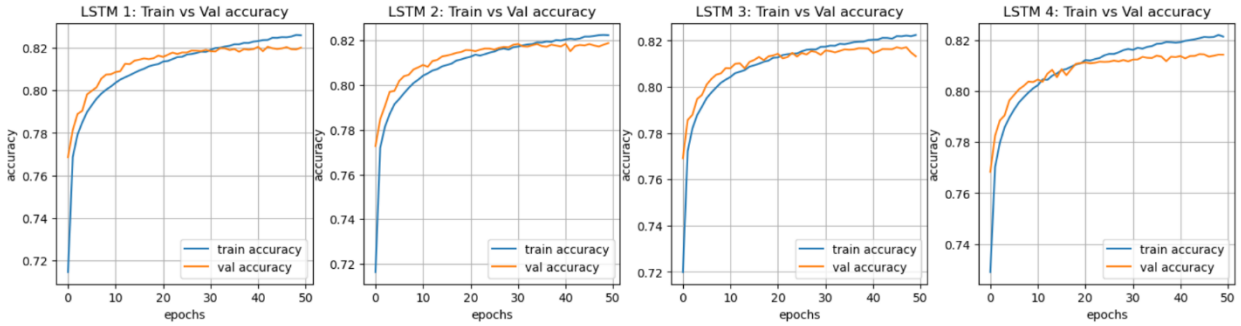

Figure 9. Accuracy graphs for each CNN-LSTM model of ensemble

Explainability:

Multi-layered models such as LSTM, dense NNs lose direct explainability through input weights, which we otherwise gain in linear models. There are many proposed methods to tackle this. Based upon factors such as,

1. If we have access to weights and gradients to the model (works for only NN-like models)
2. Should the method be algorithm dependent or agnostic
3. Computational cost of explainability method

For our task, we will be using Local interpretable model-agnostic explanations (LIME). LIME utilizes the given model as black box, with access to only inputs and outputs to the model. For a given input, LIME will create random local samples in the neighbourhood, and weight them according to the distance from the original sample. Now, LIME uses an interpretable method (Ridge regressor by default) to train on this new synthetic data. We use the original model predictions on this synthetic data as labels for regressor. The weights of this model will give us the feature importance in original sentence/input.

For this task, we have used LSTM model 4, as all models have similar performance. Following are the explainability examples of our LSTM model using LIME:

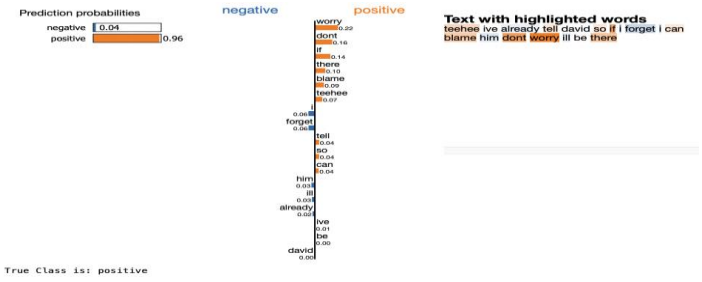

Figure 10. positive sentence visualization

For this positive sentence, words like “don’t worry” influence the prediction towards positive sentiment, although there are sequence like “i forget”, they have relatively lower weight, hence we have overall positive sentiment.

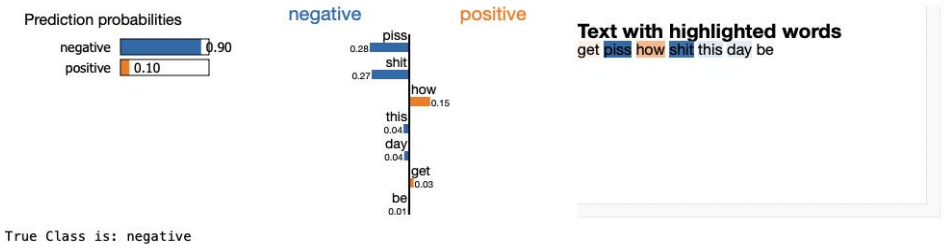

Figure 11. negative sentence visualization

Here, the LSTM model is clearly influenced by the negative words like "piss", "shit" and hence model is confident in negative sentiment prediction.

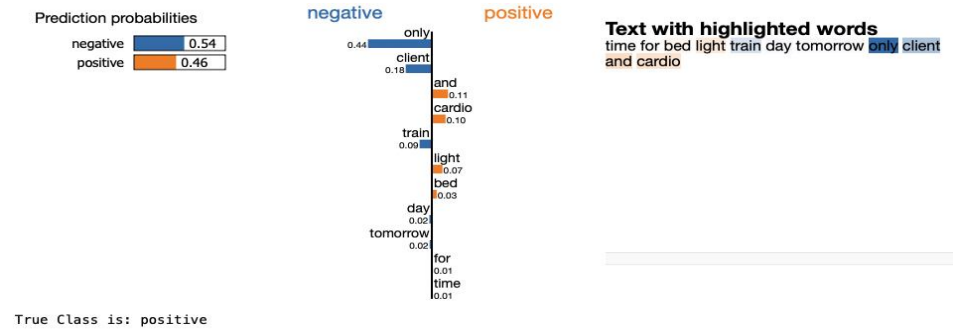

Figure 12. misclassification visualization

For this given positive sample, our LSTM model miss-classifies it as a negative sample. We can see that words like "cardio", "light" contributes towards positive sentiment. But words like "only" and "client" are perceived as negative intent and has more relative weight towards overall sentiment. Hence, we misclassify the sample as negative by a small margin.

Discussion:

1. Our dataset had a huge class imbalance against the 'neutral class', therefore none of our models performed well when predicting 'neutral' sentiments and in turn made mistakes which lowered the overall performance as it essentially adds noise to our model. Removing it from our dataset entirely would improve our model's accuracy on the majority classes.
2. We tried using Word2Vec and RoBERTa (might not have performed as well because of relatively small dataset) representations to encode our input data, however in the future we could use other general embeddings like Glove which are better at understanding and encoding global context.
3. Recurrent dropout could be added to our recurrent models as they help in better regularization than spatial1d or normal dropout. This could help us in increasing capacity of our models to better learn the sequence representations.
4. We only trained the RoBERTa model for 3 epochs (due to computational constraint), since the loss was yet to be optimized further it is very likely that training for 5 or 6 epochs would have yielded better results.

Statement of Contributions:

Our work was divided equally, and the given responsibilities were as follows:

Vaibhav Jade: Explainability, SVM implementation, Naïve Bayes implementation, corresponding parts in the report.

Rajjeshwar Ganguly: Defining the problem, data preprocessing and feature engineering, coding CNN-LSTM, GRU and NN, corresponding parts in the report.

Luong Thuy Chung: RoBERTa implementation, developing methodology, corresponding parts in the report.

We hereby state that all the work presented in this report is that of the authors.

References:

1. https://ai.facebook.com/blog/roberta-an-optimized-method-for-pretraining-self-supervised-nlp-systems/
2. https://huggingface.co/docs/transformers/model_doc/roberta
3. https://huggingface.co/blog/how-to-train

Appendix:

Hyperparameter Tuning in LSTM: We experimented with changing LSTM architectural style, namely a combination of Convolution (1D with 2 kernel size), Bi-directional LSTM, and vanilla LSTM layer. We got best result with Conv-LSTM architecture. Following plot shows this:

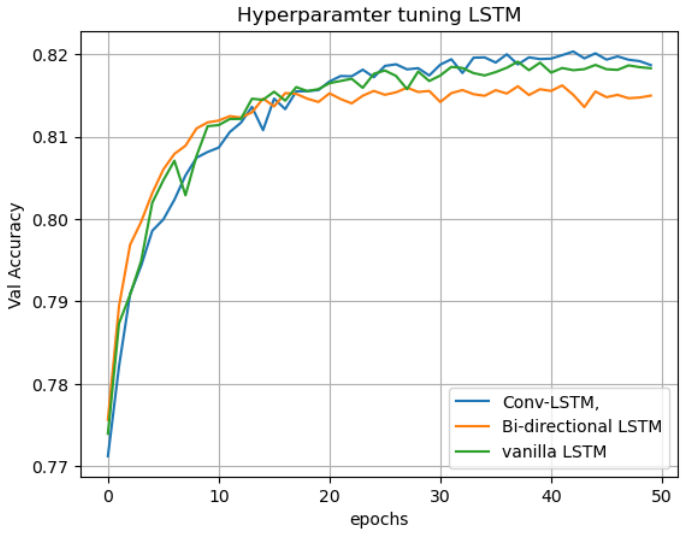

Figure 13. comparison of different lstm architectures tried

Kaggle scores:

Algorithm	Private score	Public score
Multinomial Naïve Bayes	0.7694	0.7699
SVM	0.7277	0.7274
NN	0.7318	0.7326
GRU	0.8061	0.8062
CNN-LSTM	0.8268	0.8262
RoBERTa	0.8071	0.8072

Figure 14. kaggle scores