Machine Learning Engineer Nanodegree

Capstone Project

Cheuk San Yip
July, 2019

Report

Sentiment Analysis on Conversational Data

I. Definition

Project Overview

This project tackled the challenge around sentiment analysis on conversational data in the natural language processing (NLP) domain. As opposed to most of the open-source sentiment models that were trained with datasets collected from posts on social networks (e.g. Twitter) or movie reviews (e.g. IMDb), the models in this project were trained with the Multimodal EmotionLines Dataset (MELD)[1]. The dataset contains about 13,000 utterances from 1,433 dialogues from the TV-series *Friends*, which more closely resembles the unconventional characteristics of conversational data: short, highly situational and produced alternatingly by the participants.

Problem Statement

The problem to solve is predicting the underlying sentiment when given textual features extracted from dialogues.

In essence, this is a multi-class classification problem. The 3 classes of sentiment – negative, neutral and positive, were derived from the polarities of human's basic emotions. By applying supervised learning and deep learning techniques to the field of natural language processing and text analysis, we should be able to develop models to detect sentiment of the speakers from their speeches/dialogues.

The models that have been investigated and discussed in this project include:

- Support Vector Machine (SVM)
- Random Forest Classifier
- Fully Connected Neural Networks
- Convolutional Neural Networks
- Convolutional LSTM with transfer learning from GloVe

[1] Github source: https://github.com/SenticNet/MELD1

Metrics

Metrics that are commonly used are accuracy, precision, recall and F1 score. They are calculated by evaluating true positives, false positives, true negatives and false negatives (Fig 1)[2] in the model results.

	Predicted Positives	Predicted Negatives
Positives	True Positives	False Negatives
Negatives	False Positives	True Negatives

(Fig 1: confusion matrix)

In Mathematics, their formulae are:

Accuracy = (TP + TN) / (TP + FP + FN + TN)

Precision = TP / (TP + FP)

Recall = TP / (TP + FN)

F1 score = 2 * (Recall * Precision) / (Recall + Precision)

(where TP: true positive; TN: true negative; FP: false positive; FN: false negative)

Accuracy is simply a percentage calculated by the number of times that the model predicts the correct sentiment label out of the total number of predictions made.

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations; whereas Recall is the ratio of correctly predicted positive observations to the all observations in actual class being truly positive. There is always a trade-off between precision and recall. For instance, in medical models one would like to be able to identify as many patients who are truly suffering in illness as possible, hence a high recall model is preferable.

F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. For data with uneven class distribution, F1 is usually more useful than accuracy. Accuracy works best if false positives and false negatives have similar cost. Since I would argue that for our case of sentiment analysis in conversational data is not particularly in favour of one to the other for precision and recall, plus that the dataset will be rebalanced in the preprocessing step, I would use accuracy as the evaluation metric.

[2] Reference:https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/

II. Analysis

Data Exploration

The Multimodal EmotionLines Dataset (MELD)[3] is the dataset used in this project. It contains about 13,000 utterances from 1,433 dialogues from the TV-series *Friends*. Each utterance is annotated with emotion and sentiment labels, and encompasses audio, visual, and textual modalities (Fig 2). The MELD dataset came as 3 separate csv files in which contain the dev, train and test sets respectively.

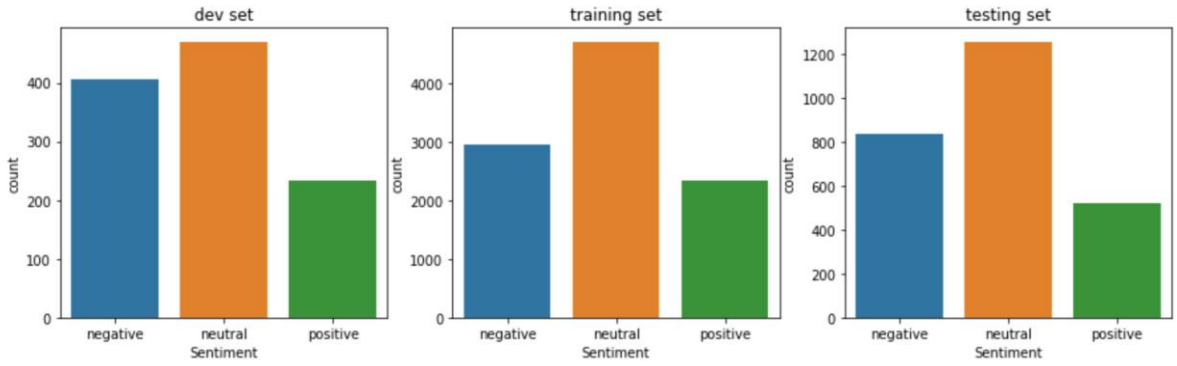
Sr No.	Utterance	Speaker	Emotion	Sentiment
43	Ugh, can you believe that guy!	Ross	disgust	negative
50	Oh my God! I overslept! I was supposed to be on the set a half an hour ago! I gotta get out of here!	Joey	fear	negative
51	Oh wait, Joey, you canÂ't go like that! You stink!	Monica	disgust	negative
44	Yeah. I really like his glasses.	Phoebe	neutral	neutral
46	What?	Monica	neutral	neutral
49	Oh no wait, oh no, the elastic on my underwear busted.	Phoebe	neutral	neutral
45	Ohh!	Phoebe	surprise	positive
47	It kicked! I think the baby kicked!	Phoebe	surprise	positive
48	Oh my God!	Monica	surprise	positive

(Fig 2: samples of the raw data)

Exploratory investigation shows that the neutral label has significantly dominated the dataset, which is more than twice as much as positive labels in training and testing sets (Table 1, Fig 3).

Sentiment	**Dev**	**Train**	**Test**
Negative	406	2945	833
Neutral	470	4710	1256
Positive	233	2334	521

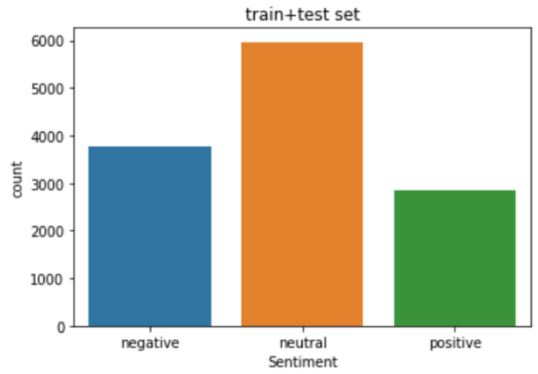
(Table 1: count of sentiment labels in all three sets)



(Fig 3: distribution of sentiment labels in all three sets)

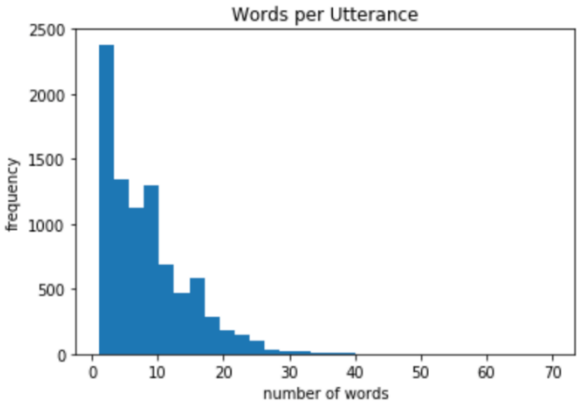
To gain more control on the ratio used for train-validate-test split, I have combined the training and testing sets. This enabled me to decide a splitting ratio that is more suitable to my need for my model development. The labels in the combined dataset follows the same imbalanced distribution (Fig 4).

[3] Github source: https://github.com/SenticNet/MELD1



(Fig 4: distribution of sentiment labels in the combined set)

A word count has been performed post-rebalancing. As expected from conversational data, the majority of utterances are short sentences (Fig 5). On average, there are 8 words per utterance.



(Fig 5: histogram shows the distribution of word count per utterance)

Algorithms and Techniques

Some of the key techniques applied in this project include:

- Lemmatization
- Tokenization
- Dimensionality reduction
- Word embedding
- Stratified K-fold cross validation

Lemmatization

Stemming and lemmatization are text normalization techniques in the field of natural language processing that are used to prepare text, words, and documents for further processing. Stemming and lemmatization helps us to achieve the root forms of inflected (derived) words.[4] For example, the word "better" has

[4] reference: https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

"good" as its lemma; the verb 'to walk' may appear as 'walk', 'walked', 'walks', 'walking'.[5] The difference between stemming and lemmatization is that stem might not be an actual word whereas, lemma is an actual language word. In this project, lemmatization has been applied to all utterances in the entire dataset.

Tokenization

Tokenization is the process of breaking up the given text into units called tokens. The tokens can be words, phrases or symbols. In NLP, instead of using the actual words as machine learning features, numeric scores that represent each word will be used. Hence, as part of feature extraction, each token will be assigned a weight after we obtained a list of tokens. The weights assigned depend on the chosen tokenization algorithm.

Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF) are the two most commonly considered algorithms. While BoW simply counts how many times a particular word appears in a document, TF-IDF measures the number of times that particular word appears in a given document. However, because words such as 'and' or 'the' appear frequently in all documents, those must be systematically discounted. The more documents a word appears in, the less valuable that word is as a signal to differentiate any given document (Fig 6). That is intended to leave only the frequent and distinctive words as markers.6 In this project, I have implemented TF-IDF as part of the feature extraction.

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \log\left(\frac{\mathrm{N}}{\mathrm{df}_i}\right)$$

$\mathrm{tf}_{i,j}$ = total number of occurences of i in j
df_i = total number of documents (speeches) containing i
N = total number of documents (speeches)

(Fig 6: formula for TF-IDF algorithm)

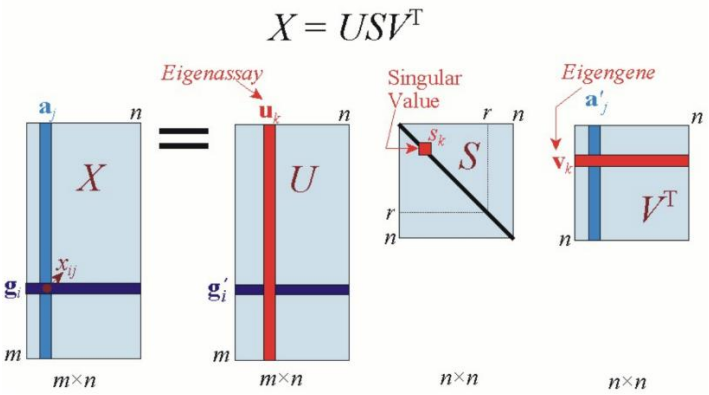
Dimensionality reduction

While we are adding as many features as possible to capture more useful indicators and obtain a more accurate result, the performance of the models might be diminishing after a certain point. This is because the sample density decreases exponentially with the increase of the dimensionality. When more features are added without increasing the number of training samples, the dimensionality of the feature space grows and becomes sparser and sparser. Due to this sparsity, it becomes much easier to find a 'perfect' solution for the model, which highly likely leads to overfitting.[7]

The truncated singular value decomposition (SVD) is one of the common algorithms used to perform linear dimensionality reduction in the feature space. In Mathematics, it involved decomposing a matrix X with high dimension to three matrices in which S is a singular matrix – only has non-zero values on the diagonal (Fig 7)[8]. This technique has been incorporated in the feature extraction stage in this project.

---

[5] reference: https://en.wikipedia.org/wiki/Lemmatisation
[6] reference: https://skymind.ai/wiki/bagofwords-tf-idf
[7] reference: https://medium.com/@cxu24/why-dimensionality-reduction-is-important-dd60b5611543
[8] reference: https://public.lanl.gov/mewall/kluwer2002.html



(Fig 7: Graphical depiction of SVD of a matrix X)

Word embedding

Normally when words are represented in a vector form, all the words are treated as independent of each other. However, the fact is some words are closely associated together when spoken. In a nutshell, word embeddings are vector representations of words that introduce some dependence of one word on the other words. Words with similar context occupy close spatial positions.

Word2Vec and GloVe are two of the most popular techniques in the word embedding space. Some more recent development in word embeddings includes language models such as ELMo and BERT. The main advantage of these models in comparison to Word2Vec and GloVe is that they have taken word order into account.

A word embedding layer is an essential building block of my convolutional LSTM deep learning model in this project. For simplicity, I have chosen the Global Vectors for Word Representation (GloVe[9]) as the embedding layer. By leveraging a pre-trained word embedding, it saves time from training my own embedding from scratch, and allows my model to take advantage of an embedding trained on a better and bigger dataset, which potentially could empower the LSTM model that I built.

Stratified K-fold cross validation

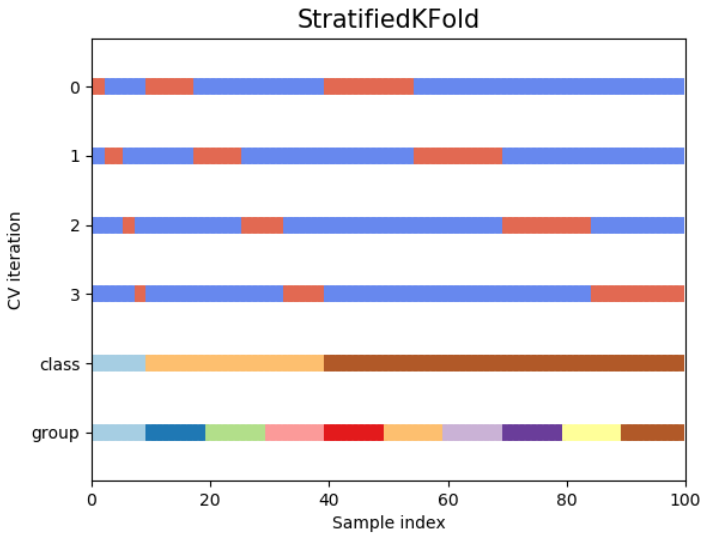
K-fold is a common type of cross validation used in machine learning. The parameter k is referring to the number of groups that a given dataset is to be split into. It partitions the training set into k equal subsets. It keeps a single fold as validation/testing set and the remaining k-1 folds as training set. The model was then trained with k-1 training set and calculate the accuracy of the accuracy by validating the predicted results against the validation set. On top of that, Stratified K-fold will make sure that for every split, the split sets will definitely contain all the distinct classes, and each set contains approximately the same percentage of samples of each target class as the complete set (Fig 8)[10].

---

[9] reference: Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. https://nlp.stanford.edu/projects/glove/

[10] reference:https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html#sklearn.model_selection.StratifiedKFold

This allows me to evaluate the accuracy of the model by averaging out the accuracies derived from all the k cases.

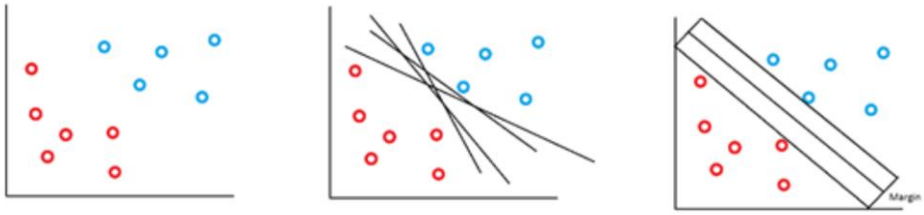


(Fig 8: Example of 4 splits with Stratified K-Fold)

Models

Support Vector Machine

SVM is based on the idea that we can separate classes in a p-dimensional feature space by means of a hyperplane. The SVM algorithm uses a hyperplane and a margin to create a decision boundary for different classes[11]. In a simplest example (Fig 9) where we have two features (distinguished by axes) namely a 2-dimensional feature space and two classes (distinguished by colours). There are more than one way to draw a boundary which will separate the two classes. This is where SVM comes into play. It will determine the best position of the boundary by maximising the distance margin between classes.

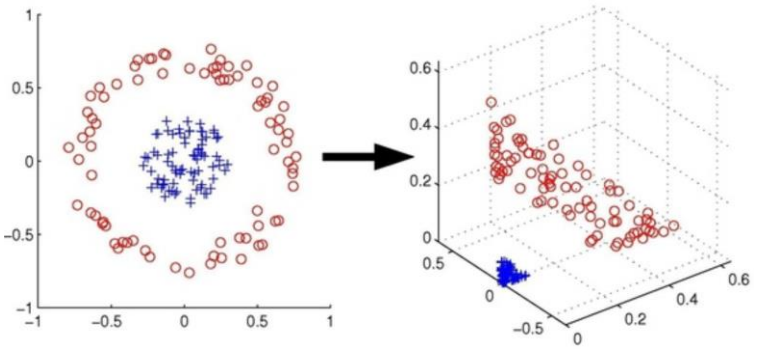


(Fig 9: illustration of the SVM algorithm)

In cases where the classification cannot be simply done linearly (Fig 10), the kernel trick will be used to solve the separation problem. The kernel is a way of computing the dot product of two vectors in feature space. It is essentially a mapping function that transforms a given space to another space. The dimension of the transformed space is often higher than the original. The logic behind this is that sometimes it is much easier to separate the classes in a higher dimensional space. Then by applying an inverse mapping, one can obtain a non-linear decision boundary on the original input space. SVM has the advantage of being effective in the high dimensional feature space. In text analysis, the dimension of the feature space

[11] Reference: https://quantdare.com/svm-versus-a-monkey/

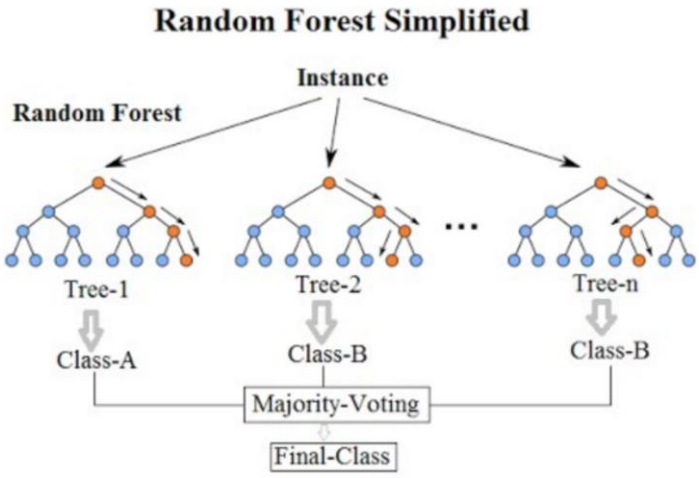
is usually determined by the number of tokens. A passage or corpus can be long and contains a great number of tokens. Therefore, such strength of the SVM makes it suitable for text classification.



(Fig 10: mapping 2D to 3D with the kernel trick)

Random Forest Classifier

Random Forest Classifier is an ensemble algorithm, which means it combines more than one algorithm for classifying objects and then take vote for final consideration of class for the test object. In Random Forest, these algorithms are multiple decision trees from randomly selected subset of training set. Every decision tree will make its own decision on which class the test object belongs to. The Random Forest classifier will then determine the final class of the test object after aggregating the results from all the decision trees (Fig 11)[12].



(Fig 11: illustration of the RF algorithm)

Taking a step up from using a single decision tree, the process of averaging the results of different decision trees helps the classifier to overcome the problem of overfitting. With Random Forest, a model can be trained with a relatively small number of samples but still obtain decent results. In contrast, a deep neural

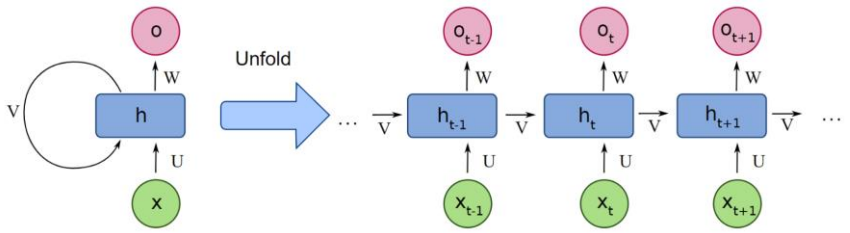
[12] Reference: https://medium.com/machine-learning-101/chapter-5-random-forest-classifier-56dc7425c3e1
Image source: https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

network needs more samples to deliver the same level of accuracy, but it will benefit from massive amounts of data, and continuously improve the accuracy[13].

Long Short-Term Memory

LSTM is a type of recurrent neural networks (RNN) which has the ability to learn and remember over long sequences of input data. Traditional RNNs (Fig 12) have the limitations on[14]:

- Short-term memory – discarding information from earlier time steps when moving to later ones, which result in the loss of important information. The inability to retain information when the sequence given is long is the biggest drawback of RNN.
- Vanishing gradient – the gradient shrinks as it back propagates through time. Gradient is the value used to update the weight used in a neural network. If a gradient value becomes extremely small, it does not contribute too much to learning.
- Exploding gradient – this occurs when the network assigns unreasonably high importance to the weights



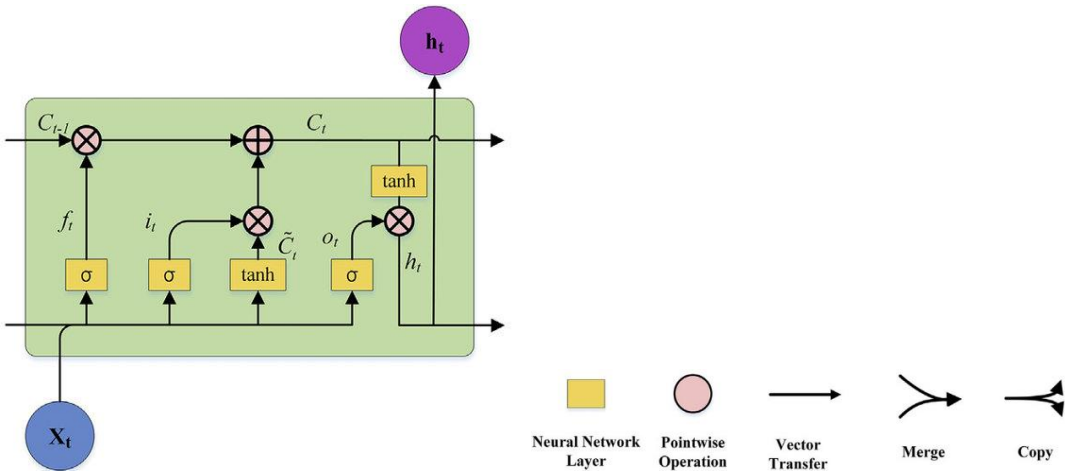
(Fig 12: illustration of RNN in folded and unfolded forms)

LSTM addresses these limitations by using 'gates' to decide if the information should be kept or forgot when data are being propagates forward. Usually the gates refer to an 'input gate', an 'output gate' and a 'forget gate'. The input gate controls the extent to which a new value flows into the cell, the forget gate controls the extent to which a value remains in the cell and the output gate controls the extent to which the value in the cell is used to compute the output activation of the LSTM unit[15]. Each of these gates is a neural network with a sigmoid activation function. The architecture of LSTM could vary – some might have fewer or different gates.

In a standard three gates scenario (Fig 13), the LSTM cell contains the following components:

- Forget gate denoted by 'f'
- Candidate layer denoted by 'C hat' – a neural network with Tanh activation function
- Input gate denoted by 'I'
- Output gate denoted by 'o'
- Hidden state denoted by 'h'
- Memory state denoted by 'C'

[13] Reference: Henrik Strøm - https://www.quora.com/What-are-the-advantages-and-disadvantages-for-a-random-forest-algorithm
[14] Reference: https://medium.com/datadriveninvestor/a-high-level-introduction-to-lstms-34f81bfa262d
[15] Reference: https://en.wikipedia.org/wiki/Long_short-term_memory



(Fig 13: example of a LSTM architecture with forget (f), input (i) and output (o) gates)

Benchmark

The benchmark model is a simplest model that will predict the same class for any given sentences/utterances. The class it will predict depends on the majority class seen in the training dataset. Since I have resampled the data to obtain a more balanced set of labels, the resulting ratio between labels are 36% negative, 31% neutral and 33% positive. As a consequence, the baseline model has an accuracy close to 36%. Any model that could achieve an accuracy higher than 36% would be considered as a 'good enough' model.

A simple baseline model has been constructed in the way that fits in the standard Sklearn pipeline (Fig 14).

```
class BaseModel():
    def __init__(self):
        self.pred=None

    def fit(self, X, y):
        self.pred = Counter(y).most_common()[0][0]

    def predict(self, x_test):
        return x_test.shape[0] * [self.pred]
```

(Fig 14: class definition for the baseline model)

III. Methodology

Data Preprocessing

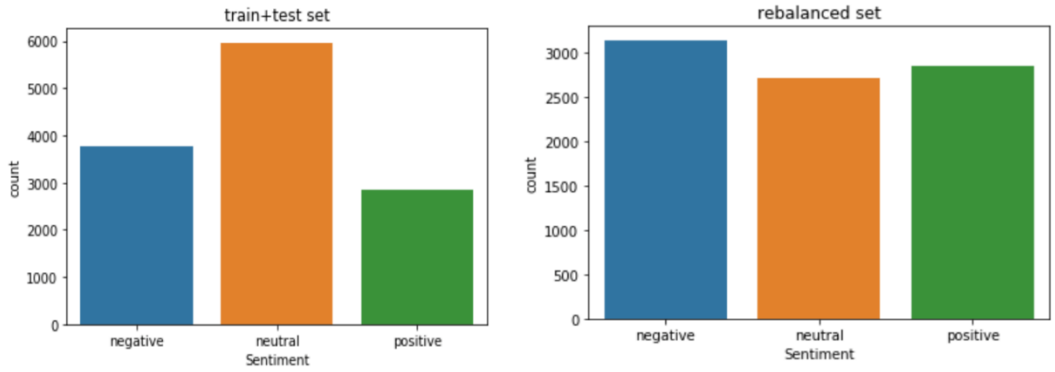
The data preprocessing consists of the following steps:

- Resampling
- Lemmatization

Resampling

Recall that the labels in the combined dataset follows an imbalanced distribution with the neutral sentiment label significantly out-numbered the negative and positive labels. Data rebalancing is therefore required, otherwise the accuracy-driven models trained with such dataset would choose to 'conveniently' output the most seen labels as a prediction to no matter what data they are fed. There are several different methodologies for rebalancing such as putting on class weights, over-sampling the minority classes or under-sampling the majority classes. Considering that over-sampling involves data generation, which might introduce the risk of changing data patterns in the original dataset, I have decided to under-sample the neutral class.

The resampled dataset has 3140 negative, 2712 neutral and 2855 positive labels, with a ratio of 36:31:33 for negative, neutral and positive (Fig 15). I have deliberately added slightly more negative data points for the models to learn, because the detection of negative sentiment is generally more of interest in real life applications.



(Fig 15: distribution of sentiment labels before [left] and after [right] rebalancing)

Lemmatization

The Snowball stemmer in the **nltk** package has been used to stem and lemmatize the utterances. By applying the stemmer to every utterance, each word in the utterance has been normalised to its root form.

Implementation

Sklearn's *train_test_split* has been applied on the resampled dataset to help splitting the data randomly into 67% for training and 33% for testing.

Feature extraction has been done in two slightly different approaches to adapt the input requirements for Sklearn's supervised learning models and Keras' deep learning models respectively.

<u>Supervised learning:</u>

Feature extraction

For tokenization, I have used the TfidfVectorizer available from Sklearn with *lowercase* parameter set to true and *stop_words* parameter set to 'english'. This would remove the common English stop words as

they are 'meaningless' from a features' perspective and provide little value in terms of improving predictive power. The output of the TfidfVectorizer, a large sparse matrix, was then fed into TruncatedSVD for dimensionality reduction.

Having prepared the features (aka. X), the labels (aka. y) also needed transformation so as to be easily fed into the models. I applied the *factorize* method offered by Pandas to the sentiment label set. It is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values[16]. The resulting mapping of labels is {'negative': 0, 'neutral': 1, 'positive': 2}.

Hyper-parameter tuning

One of the difficulties, and a crucial step, in training a supervised learning model is parameter tuning. The GridSearch available in Sklearn could be of great assistance to hyper-parameter tuning. The *GridSearchCV* algorithm exhaustive search over specified parameter values for a given estimator[17]. It has been used to find the best set of parameters that could maximise the model accuracy.

Model development

Support Vector Machine (SVM): the model was built with the SVC class from Sklearn's svm module. GridSearchCV was used to identify the most suitable C – the penalty parameter of the error term, and the kernel type. The best estimator yielded by GridSearch was used as my final SVM model (Fig 16).

Best model by GridSearchCV

```
SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,
decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
kernel='linear', max_iter=-1, probability=True, random_state=None,
shrinking=True, tol=0.001, verbose=False)
```

(Fig 16: definition of the best SVM returned by GridSearch)

Random Forest Classifier: the model was built with the RandomForestClassifier from Sklearn's ensemble module. Again, GridSearchCV was used to select the best estimator (Fig 17) by exploring different values for *n_estimators*, *min_samples_split*, *min_samples_leaf* and *max_features*.

Best model by GridSearchCV

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
max_depth=None, max_features='auto', max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=4, min_samples_split=4,
min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
oob_score=False, random_state=None, verbose=0,
warm_start=False)
```

(Fig 17: definition of the best Random Forest classifier returned by GridSearch)

[16] reference: https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.factorize.html
[17] reference: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Deep learning:

Feature extraction

For tokenization, I have used the Tokenizer offered by Keras with setting every word to lower case and striping out punctuations with the *filter* parameter. To find an appropriate value for the *num_words* parameter, I have investigated the vocabulary size of my resampled dataset, which found 4185 unique tokens. As a result, I have selected a *num_words* of 6000.

The tokenized words were then converted to sequences. Since sequences need to be of the same length to form a matrix, additional zeroes have been padded to shorter sequences. The *pad_sequences* method has been used to achieve this. The *maxlen* parameter of this method was set to 100, which should cover the longest utterance in the dataset.

For convolutional neural network (CNN) specially, the vector representations of features have been reshaped to accommodate the input shape for the model.

Transfer learning

The pre-trained GloVe word vectors 'glove.6B.100d.txt'[18] was downloaded and I have constructed a weight matrix out of this data. It was trained on articles from Wikipedia and newswire text data from Gigaword. To apply transfer learning, I have loaded these weights to the first layer, the embedding layer, of my convolutional LSTM model with *trainable* set to false.

Model development

Fully-Connected Neural Network: a simple model was built with only fully connected dense layers (Fig 18). Batch normalisation layer and dropout layer with 20% drop out rate have been added in-between dense layer. Each dense layer has a 'relu' activation function, except for the final layer which used 'softmax' instead. The output dimension of the last dense layer was set to 3 so as to align with the three sentiment outcomes. The reason for using 'softmax' in the output layer is for this classification problem the classes are mutually exclusive, that is, every utterance can only possess one sentiment at a time. If we were to treat the three outputs from the final dense layer as the likelihoods/probabilities of the corresponding sentiment associated with the given utterance, we would like to have the summation of all outputs to be equal to 1. This is exactly what the Softmax function can help to achieve.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                3232
_________________________________________________________________
dropout_1 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_2 (Dense)              (None, 64)                2112
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
dense_3 (Dense)              (None, 128)               8320
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 387
=================================================================
Total params: 14,819
Trainable params: 14,435
Non-trainable params: 384
```

(Fig 18: model architecture for fully-connected neural network)

[18] downloaded from: https://nlp.stanford.edu/projects/glove/

Convolutional Neural Network (CNN): the basic architecture of my CNN was designed to be alternating Cov1D layers and MaxPooling1D layers (Fig 20). Padding was applied, so was kernel regularisation with the L2 regulariser. The rationale behind the use of Cov1D layers is that it is arguably good for text in general, whereas Conv2 D is good for audio and images where spatial matter. The transaction from convolutional layers and dense layers was bridged by a GlobalAveragePooling1D layer. This pools the data by averaging them. It also serves as an alternative to a Flatten layer, yet does more than just simply converting a multi-dimensional object to a one-dimensional tensor.

One of the obstacles I encountered when fitting the CNN was finding the input shape that could be accepted by the first layer. Reshaping was required on the feature vectors (Fig 19).

```
n_length = x_train.shape[0]
n_features = x_train.shape[1]

x_train_reshaped = x_train.reshape(n_length, n_features, 1)
x_test_reshaped = x_test.reshape(x_test.shape[0], n_features, 1)
X_test_ex_reshaped = X_test_ex.reshape(X_test_ex.shape[0], n_features, 1)
```

(Fig 19: python code to reshape input variables for CNN)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_7 (Conv1D)            (None, 100, 16)           48
_________________________________________________________________
max_pooling1d_7 (MaxPooling1 (None, 50, 16)            0
_________________________________________________________________
conv1d_8 (Conv1D)            (None, 50, 32)            1056
_________________________________________________________________
max_pooling1d_8 (MaxPooling1 (None, 25, 32)            0
_________________________________________________________________
conv1d_9 (Conv1D)            (None, 25, 64)            4160
_________________________________________________________________
max_pooling1d_9 (MaxPooling1 (None, 12, 64)            0
_________________________________________________________________
global_average_pooling1d_3 ( (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 128)               8320
_________________________________________________________________
batch_normalization_3 (Batch (None, 128)               512
_________________________________________________________________
activation_3 (Activation)    (None, 128)               0
_________________________________________________________________
dense_6 (Dense)              (None, 64)                8256
_________________________________________________________________
activation_4 (Activation)    (None, 64)                0
_________________________________________________________________
dense_7 (Dense)              (None, 3)                 195
=================================================================
Total params: 22,547
Trainable params: 22,291
Non-trainable params: 256
```

(Fig 20: model architecture for convolutional neural network)

Long Short-Term Memory (LSTM): the model (Fig 21) involves doing transfer learning from a pre-trained GloVe word embedding. The weights have been loaded into the first layer of the model – an embedding layer with *input_dim* set to the vocabulary size (6000) and *input_length* set to the maximum sequence length (100) as outlined in the previous section of this report. The loaded weights of the embedding layer have been frozen by setting *trainable* to false.

Following the embedding layer, comes a Conv1D layers and three LSTM layers. Every LSTM layer were accompanied by a BatchNormalization layer, a TimeDistributed layer and a Dropout layer. The time distributed layer is a wrapper around LSTM. In essence, it reduces the number of outputs from a LSTM layer with a fully-connected layer. For instance, a LSTM layer with *unit* equal to 5 neurons will return a sequence of 5 outputs, one for each time step in the input data. By wrapping a TimeDistributed(Dense(1)) layer around it, it will output one time step from the sequence for each time step in the input, even though five time steps have been processed at a time[19]. At last, the model was concluded with a standard approach of adding a few fully-connected layers to obtain the final probability prediction on sentiment labels.

One of the challenges in fitting a LSTM (or any deep learning models) is deciding the number of neurons, the type, number and order of layers. Finding the optimal involves some experimental try and error. For instance, in the first few neural networks I constructed, I found that my models always took an unusually long time to train for even one single epoch. Later I came to the realisation that it was due to the fact that I have too many trainable parameters (in some cases, I have over 2 million of parameters). Having identified such issue, I started adding in BatchNormalization layers to my neural network architectures, which helps normalize the input layer by adjusting and scaling the activations. This reduces the amount of trainable parameters and essentially speed up the training time.

```
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 100, 100)          600000
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 99, 256)           51456
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 24, 256)           0
_________________________________________________________________
lstm_1 (LSTM)                (None, 24, 128)           197120
_________________________________________________________________
batch_normalization_1 (Batch (None, 24, 128)           512
_________________________________________________________________
time_distributed_1 (TimeDist (None, 24, 64)            8256
_________________________________________________________________
lstm_2 (LSTM)                (None, 24, 64)            33024
_________________________________________________________________
batch_normalization_2 (Batch (None, 24, 64)            256
_________________________________________________________________
time_distributed_2 (TimeDist (None, 24, 32)            2080
_________________________________________________________________
lstm_3 (LSTM)                (None, 24, 32)            8320
_________________________________________________________________
batch_normalization_3 (Batch (None, 24, 32)            128
_________________________________________________________________
time_distributed_3 (TimeDist (None, 24, 16)            528
_________________________________________________________________
global_average_pooling1d_1 ( (None, 16)                0
_________________________________________________________________
dense_4 (Dense)              (None, 16)                272
_________________________________________________________________
dense_5 (Dense)              (None, 3)                 51
=================================================================
Total params: 902,003
Trainable params: 301,555
Non-trainable params: 600,448
```

(Fig 21: model architecture for convolutional LSTM with GloVe)

[19] reference: https://machinelearningmastery.com/timedistributed-layer-for-long-short-term-memory-networks-in-python/

Refinement

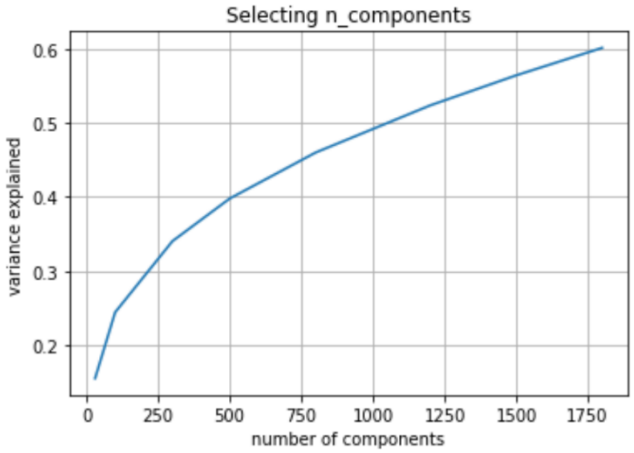
Refinements were made such as by tuning hyper-parameters, defining model architectures in greater detail and attempting different combinations of model components. Some of the refinements involved the use of:

- TruncatedSVD
- L1/L2 regularisers
- Dropout & BatchNormalization layers

TruncatedSVD

Usually, after transforming the text with TF-IDF tokeniser, the output matrix is sparse and in an enormous size. This could be problematic as it increases the computational cost and often lead to overfitting. Hence, reducing dimensionality became a step I took to improve my model training time and accuracy.

A key parameter used in TruncatedSVD is the number of components. If we choose a higher number, we get a closer approximation the original matrix. On the other hand, choosing a smaller number will save the computation and training time. Selecting an appropriate number is finding a balance between time and accuracy. From another perspective, what should be considered is the variance explained versus the actual uplift in accuracy. Obviously the with an increased number of components comes better variance explained (Fig 22). Eventually, by assessing the accuracies of my supervised models when different n_components were used (Table 2), I settled with 500 components, which would have 40% variance explained.



(Fig 22: relationship between the number of components used in SVD and variance explained)

	Average accuracy over 5 folds	
n_components in SVD	*SVM*	*Random Forest*
300	*49.04%*	*47.78%*
500	*49.628%*	*47.902%*
800	*48.25%*	*46.76%*

(Table 2: average model accuracies influenced by selecting different n_components in SVD)

L1/L2 regularisers

In some of my early attempts to neural networks training, I discovered that the models quite often tend to overfit to the training set quickly. Thus, I added L2 regulariser to most of the layers. This helps regularise the error function to punish high coefficients.

The reason for applying L2 is because it tends to maintain all the weights homogeneously small, whereas L1 regularisation usually ends up with small weights tend to go to zero and resulting in sparse vectors. In general, L2 is said to be good for model training and L1 is good for feature selection.

Dropout & BatchNormalization layers

The Dropout layer helps to prevent overfitting by ignoring randomly selected neurons during training, and hence reduces the sensitivity to the specific weights of individual neurons. Though after adding a few Dropout layers, the test accuracy tends to get slightly worse than without Dropout layers, the benefit of having Dropout layers can generalise the performance of the model.

Normalisation is often used as a pre-processing step to make the data comparable across features. It does so by shifting inputs to zero-mean and unit variance. This leads to higher learning rate and better speed. BatchNormalization is similar but applied the hidden layer to get improvement in the training speed. It also reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer's activations. Therefore, if we use batch normalisation, we will use less dropout, which is a good thing because we are not going to lose a lot of information[20].

IV. Results

Model Evaluation and Validation

First of all, the SVM returns the most accurate predictions out of all models (Table 3).

	SVM	Random Forest	Fully-Connected	CNN	LSTM with GloVe
Accuracy	49.63%	47.33%	36.88%	38.03%	42%

(Table 3: summary on model accuracies – testing set from rebalanced dataset)

Validation step was taken to ensure the models' robustness. For the supervised learning models: SVM and Random Forest, I have written a function to apply Stratified K-fold to cross-validate the results (Fig 23).

```
def predict_with_cv(clf, X, y, n_splits=5):
    skf = StratifiedKFold(n_splits=n_splits, random_state=42)
    acc_collector=[]
    nth_fold = 0
    for train_index, test_index in skf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf.fit(X=X_train, y=y_train)
        y_pred = clf.predict(X_test)
        acc = round(100*np.sum(y_pred==y_test)/len(y_pred),2)
        nth_fold += 1
        acc_collector.append(acc)
        print('Fold {}: F1 score = {}; Test ccuracy = {}%'.format(
            nth_fold, f1_score(y_pred=y_pred, y_true=y_test, average="weighted"), acc))
    print('Average accuracy over {} folds is {}%'.format(n_splits, sum(acc_collector)/len(acc_collector)))
```

(Fig 23: function to apply Stratified K-fold and obtain testing result from each fold)

[20] Reference: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

The accuracies I used to assess the performance and to compare with other models are the average accuracies I got from doing 5 folds.

On the other hand, for the deep learning models, I have used a validation split of 0.4 when calling the *fit* method. The neural networks were trained with 60-100 epochs, and check points were made by the end of each epoch to evaluate the improvement in *val_loss*. If it has improved from the last epoch, the model weights will be saved and overwrite the last best model.

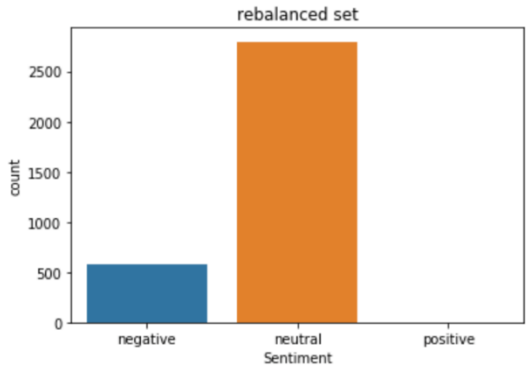
Early stopping

Early stopping has also been implemented in the training of deep learning models. It is usually applied to avoid overfitting by stopping the iterations early in the training process. Because without knowing the optimal number of times that we should pass training examples through backpropagation to get the best model, we usually will define a large enough epoch. The problem with this is that the number of epochs could be larger than enough. The models tend to overfit to the training set as it iterates through higher number of epochs. To prevent the final model from overfitting, the early stopping algorithm can terminate the training when it no longer shows improvement over a few consecutive epochs.

Both the checkpoints and early stopping are implemented to make sure the final models are the ones to make the best predictions.

Extra testing with imbalanced data

A robust model should be able to make a satisfactorily accurate prediction when deal with any unseen data. It should be predicting on a case by case basis, that is, not assuming any label distribution on the data to predict. Hence, to test the robustness of my models, I have created an extra testing set from the remaining samples in the original dataset. In other words, it is the delta between the original dataset and the rebalanced dataset. This new testing set is highly imbalanced (Fig 24).



(Fig 24: distribution of sentiment labels in the imbalanced testing set)

The supervised models are surprisingly good at predicting the additional testing set while the neural networks are struggling with the imbalance. Judging from the classes they predicted (Fig 25), the neural networks seem to still 'remember' the training set too well and not generalised enough.

	SVM	Random Forest	Fully-Connected	CNN	LSTM with GloVe
Accuracy	82.84%	83.2%	16.85%	13.03%	14.07%

(Table 4: summary on model accuracies - imbalanced testing set)

```
Fully-Connected: Counter({0: 2225, 2: 963, 1: 188})
      CNN      : Counter({0: 2482, 2: 893, 1: 1})
LSTM with GloVe: Counter({0: 2284, 2: 998, 1: 94})
where {'negative': 0, 'neutral': 1, 'positive': 2}
```

(Fig 25: class predicted by deep learning models for the imbalanced testing set)

Justification

According to the testing accuracy (Table 3), all models out-performed the benchmark models which only has 36% accuracy. However, having collected the results from using an extra unseen testing set (Table 4), the accuracies for the deep learning models have dropped significantly to a point that makes worst prediction than the benchmark model. One of the main causes is due to overfitting to the training set.

Though there are still rooms for improvement in the supervised models, they indeed performed better than the neural networks built so far in this project. Therefore, the SVM and the Random Forest classifier will be selected as the final models.

Robustness of supervised models

It was a close match between the SVM and the Random Forest classifier and they both achieved relatively satisfactory results. Nonetheless, it is worth drilling down further to the robustness of the two model. One of the useful measurement is the standard deviation over the 5-fold cross validation (Table 5). A small standard deviation would demonstrate that the models' performances were not fluctuating much. SVM was shown to have smaller standard deviation compared to Random Forest in both the original testing set and the imbalanced testing set. This implies the SVM model is slightly more robust against small perturbations in the training data.

	Original resampled testing set		*Extra imbalanced testing set*	
K-fold / Model	*SVM*	*Random Forest*	*SVM*	*Random Forest*
1st fold	*49.77*	*46.84*	*82.37*	*83.56*
2nd fold	*49.2*	*49.31*	*82.94*	*83.53*
3rd fold	*49.34*	*49.28*	*82.94*	*82.64*
4th fold	*50.03*	*47.04*	*82.76*	*83.06*
5th fold	*49.8*	*47.04*	*83.21*	*83.21*
Standard Deviation	*0.3455*	*1.27429*	*0.30989*	*0.37909*

(Table 5: fold-wise accuracies and standard deviation over 5 folds)

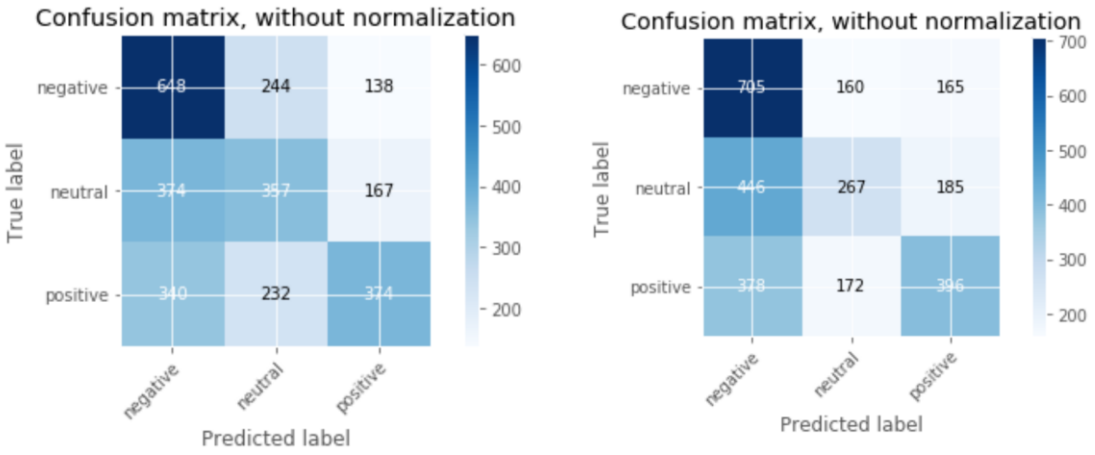
V. Conclusion

Free-Form Visualization

Supervised Learning:

After training a SVM and a Random Forest classifier, I have plotted the confusion matrix for both models to get more insight to their predictions (Fig 26). One significant observation is that both models did better in classifying the negative sentiment than the neutral and positive. Out of neutral and positive, the models did slightly better in predicting positive. This intriguing phenomena led my thought back to the distribution of the dataset. In the resampled dataset, the ratio between negative, neutral and positive is 36:31:33. It is possible that because the models have seen more negative data points so it is more capable to identify

negative sentiment. However, there might be other factors such as words used by speakers to express negative emotions have more distinct characteristics. Further analysis on the dataset or moving from sentiment analysis to emotion analysis would be helpful to understand such behaviour.

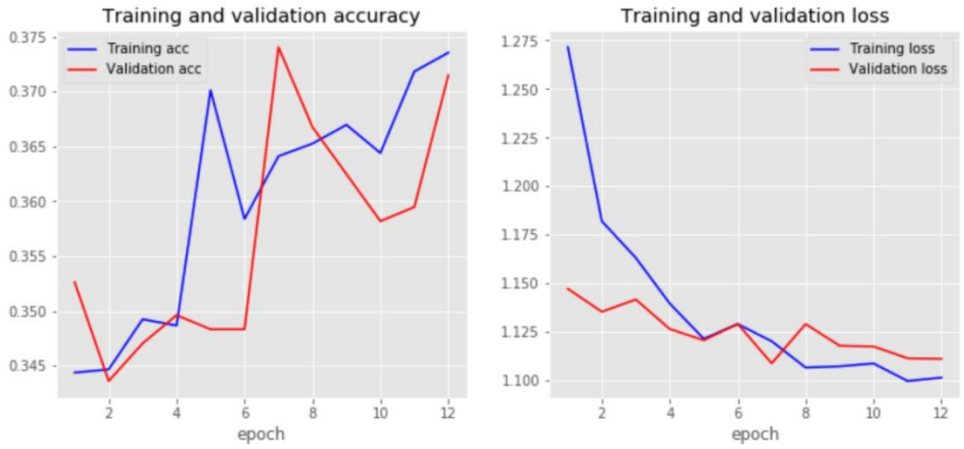


(Fig 26: confusion matrix for SVM [left] and Random Forest [right])

Deep learning:

One thing I would like to discuss is the training process of the neural networks in this project. The training histories were logged and illustrated with validation accuracy/loss plots for fully-connected neural network (Fig 27), CNN (Fig 28) and LSTM (Fig 29).

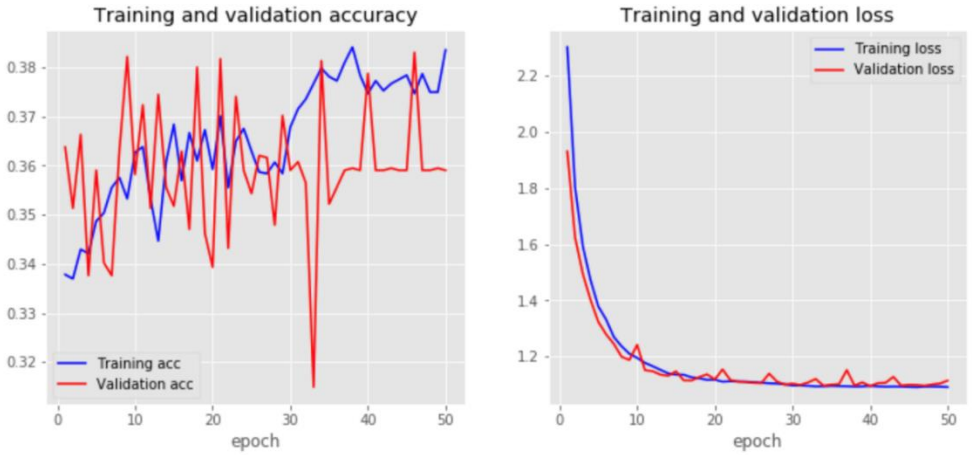
What time should the training be stopped at is one of the key questions to ask when fitting a neural network. The fully-connected model stopped the training after the 12th epoch. Judging from the validation loss plot, this is the time when training loss and validation loss just started to show convergence. The climbing trend of training and validation accuracies suggests that the model could still benefit from more backpropagations. Thus, the training should have been continued for a few more epochs.



(Fig 27: val_acc, val_loss plot for Fully-Connect neural network)

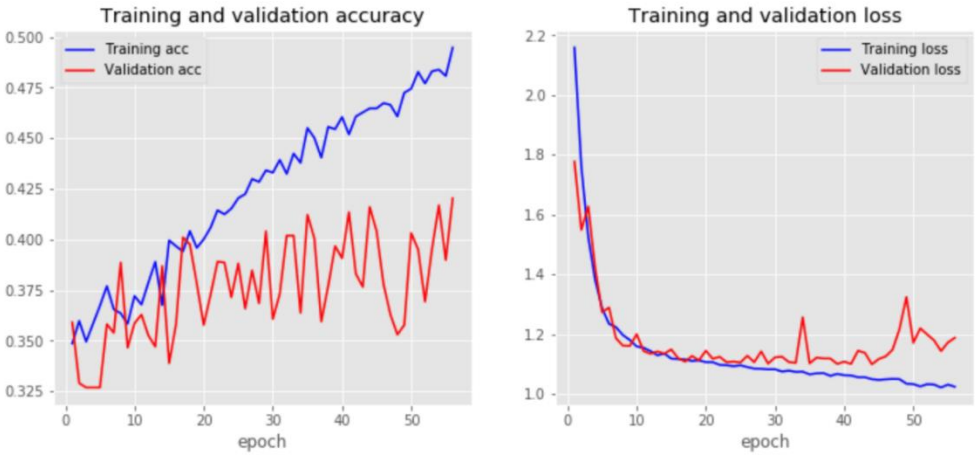
On the other hand, the validation loss plot for CNN provides a better view on when the training loss and validation loss started to converge and become flattened. We can choose to stop the training near the

20th epochs. Not only because this point is when both validation losses start to flatten out, but also because beyond the 25th epoch the training accuracy is starting to increase while the validation accuracy does not improve and has the risk of diminishing.



(Fig 28: val_acc, val_loss plot for CNN)

Last but not least, the validation loss plot for LSTM shows convergence in the first 25th epochs, but gradually starts to diverge. This validation loss continues increasing and moving away from the training loss. Similar shown in the validation accuracy plot in which the training accuracy is growing steeply and diverge from the validation accuracy. This suggests the LSTM model is seriously overfitted to the training set in the later epochs. The training process should have been stopped at the point around the 10th to 20th epoch.



(Fig 29: val_acc, val_loss plot for LSTM with GloVe)

Reflection

During my first few attempts to deep learning modelling I have encountered an issue where my neural network always predicted the same class regardless of what features it took. Having conducted further research, I was able to identify a few potential solutions:

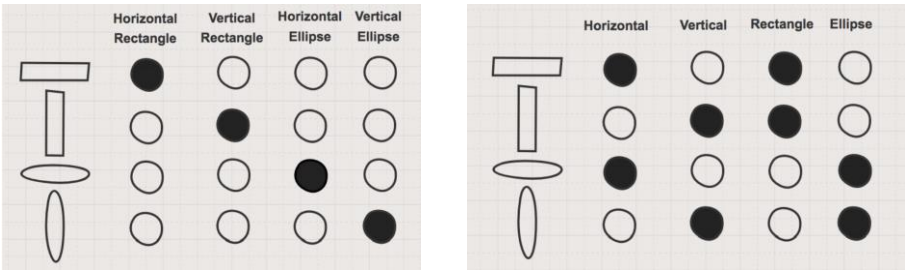
- Rebalance the dataset. This is to handle the unbalanced label distribution.
- Increase the number of epochs, so that the model goes through more iterations to update the weights and consequently more chances to learn.
- To solve the problem of LSTM always predicting the same class even when trained with balanced data, the 'return_sequences' param in the LSTM layer needs to be set to True and add a Flatten layer. In addition, 'return_sequences=True' must be set when stacking LSTM layers so that the second LSTM layer has a multi-dimensional sequence input.

By implementing these, the issue of predicting only one class was resolved.

Improvement

I have considered a couple of refinements, which include:

1. The problem with bad performing deep learning models might be caused by the lack of features because the conversational utterance are mostly short. The focus of building deep learning models should probably be utilising the idea of distributed representation (Fig 30)[21] of words where each input is represented by many features and each feature is involved in many possible inputs. It is worth try implementing embedding model such as Word2Vec.



(Fig 30: non-distributed [left] and distributed [right] representation)

2. Try a different evaluation metrics such as using precision and recall as measurement metrics (Fig 31)[22]. Evaluate the model performance with confusion matrix and ROC curve.
3. Adding more generalisation and regularisation methods when building a neural network to avoid overfitting.
4. For transfer learning, try to load weights from a different GloVe pre-trained word vectors that trained on a larger set of data.
5. In addition to text analysis, try extract audio features and develop a multi-modal model. It can start by building two separate models, one makes predictions on text features, the other on audio features. Eventually, developing an algorithm to fuse the multiple modality models together as one sentiment model.

[21] source: https://www.oreilly.com/ideas/how-neural-networks-learn-distributed-representations
[22] source: https://towardsdatascience.com/precision-vs-recall-386cf9f89488