Sentiment Analysis on Tweets
Team No. 34
Sebastian Muir-Smith, n10199721, Contribution: 25%
Joseph Grech, n9734902, Contribution: 25%
Angus Nobbs, n9736077, Contribution: 25%
Matthew Lord, n9743031, Contribution: 25%

Introduction/Motivation:

Sentiment analysis of statements allows an algorithm to determine whether a given sentence or in this case, a short statement, expresses a positive or negative sentiment. There are several advantages to selecting this project, including the lower training time of algorithms using text based data as opposed to images and the relative ease of finding suitable datasets. Additionally, sentiment analysis has multiple real world applications, ranging from brand name monitoring on social media to using the sentiments of financial news to inform stock trading strategies.

This project attempts to implement three different algorithms to conduct sentiment analysis, Naive Bayes, Support-Vector Machine and a Long-Short Term Memory Neural Network. This report will then detail the results of related works, how the dataset was prepared, the methodology used, results recorded and possible future improvements.

Related Works:

Sentiment140 is a sentiment analysis toolset developed by Stanford computer science graduates A. Go, R. Bhayani, L. Huang. In 2009. Similar to our project, Sentiment140 implemented Naive Bayes and Support-Vector Machine algorithms, however, Maximum Entropy was implemented in Sentiment140 while this project used Long-Short Term Memory. Sentiment 140 was able to achieve reasonably high accuracy of approximately 80% for all three algorithms. These results provide a baseline accuracy to which we can compare the success of our own implementations.

Data:

The chosen dataset was sourced from the github repo https://github.com/HamidNE/sentiment-analysis-with-lexicon. The dataset was given in a text document form, with each row containing an unsplit string of the statement's content, followed by a binary label (0/1) indicating the statement's sentiment. There are 6918 rows or statements in the dataset. If each row is split on whitespace characters, and each discrete string outputted by that split is considered as a word, then there are initially 2584 unique words in the dataset.

The dataset is pulled from a person's personal text analysis project, which is focused on using lexicon analysis to predict the sentiment of statements. Lexicon analysis is quite a simple tool, and thus it relies on simple training and testing data, which is evident in this project's dataset. The data is quite clean, with little complexity in statements and low variety in vocabulary. This is evident in comparing n to d, as the number of unique words post-processing is only 2046, much lower than the n of 6918. From other studies, to obtain a number of unique words less than number of sentences, n must usually be in the millions, due to the immense size of an average person's vocabulary (Pennington, 2019). This abnormally low vocabulary size is both beneficial and detrimental.

Having the number of words much lower than the number of statements is essential to train algorithms on, as algorithms struggle to meaningfully separate classes if inputs have insufficient aliasing and repetitions. E.G. In a categorical factor, if a factor level only appears once in the dataset, then an effect cannot be reliably attributed to it. Dimensionality reduction techniques such as word embedding can somewhat alleviate this need for n >> d, however, word2vec also suffers from high-dimensionality problems in training. In this regard, the chosen dataset is useful to produce meaningful results whilst still ensuring that data processing and model training times are kept reasonable for a report of this scope, thanks to not having to use millions of data points.

However, this low dimensionality implies serious data homogeneity problems. This issue is confirmed when exploring the dataset; many sentences are quite similar in their language, structure and topics. The data's homogeneity is likely a result of overfiltering by the initial dataset creator. This implies that models can quite easily make extremely accurate predictions on data points within the dataset, but as the dataset does not capture the size and complexity of the english vocabulary correctly, it might suffer on outside of dataset evaluation. This is not to say the models trained are prone to overfitting, rather that the dataset is so "easy" that high test accuracy rates are likely, but when applied to more complex dataset, these models may not be complex enough to perform well. Even so, due to the simplicity and low size of the dataset, it is great to practice models and techniques on. Perhaps the fundamental techniques and lessons learnt in this analysis can be applied to a larger, more real-world dataset in following research, if real-world performance is desired. With this limitation in mind, data processing can be performed

The data is initially ordered with the top 3943 statements only comprising of positive sentiment labels, and the bottom 2973 statements are only negative. This is problematic for two reasons: The ordering of the data points is problematic when algorithms are trained, as stochastic gradient descent and similar optimisation methods are severely inhibited when repeatedly given data of only one class, as they cannot learn how to differentiate non-aliasing class labels if they do not appear together in mini-batches. Additionally, the imbalance in numbers of each of the two classes is problematic, especially in frequency-based methods such as naive bayes, and thus this limitation in the dataset must be considered. To fix the first problem, the rows of the dataset are completely randomised in order before analysis or further cleaning.

After this point, data cleaning can be performed using the MatLab Text Analytics Toolbox, in order to reduce the number of unique words and improve the overall quality of the data. Firstly, all words are converted to lowercase form, to ensure that capitalised and non-capitalised forms of the same words are considered as one word. After this, all urls are removed from statements using regular expressions, as they are often not seen more than once in the dataset and provide little value as predictors. The same justification is used to justify removing all words of less than 1 and more than 15 characters long from the dataset. At this point punctuation is also removed, to reduce the size of the vocabulary. Additionally, using the MatLab-supplied lexicon of “stop” words, a number of likely useless words are removed from the dataset. At this point, words could be normalised to further reduce dimensionality and improve frequency-based methods, however, this would inhibit the usefulness of the word cloud visualisation methods used later in this report. The data processing stage is now complete, returning a cell array of 6918 statements, each comprised of a subset of 2046 unique words, a reduction of 538 unique words from pre-processing.

Methodology:

Dimensionality Reduction

One of the main problems in text analytics the curse of dimensionality. If each feature is taken as a binary existence or lack of a word in a given statement, then even with the relatively low-vocabulary data used in this report, there would be a dimensionality of over 2000 features. Using this many features in models would likely result in an undefined system in logistic regression, improper alisaising and interaction to fit an accurate svm, too many parameters in a neural network, and inaccurate estimates for priors when creating a naive bayes classifier. Therefore, the dimensionality of data must first be reduced before fitting models.

MatLab’s relieff (similar to multivariate relative discrimination criterion) feature selection algorithm was first employed to try to only use features (words) that explain a large amount of variation and seperate the two classes. However, using this approach the svm model only reached a maximum test accuracy of 70%, and the algorithm struggled with computational efficiency with the large number of features and observations, and thus it was not suitable.

A much better solution is the concept of word embedding. Word embedding (Specifically google’s Word2Vec algorithm) is a machine learning feature extraction technique similar to Principal Component Analysis (PCA), but specifically focused on reducing the dimensionality of textual data. Word2Vec works under the distributional hypothesis, which posits that “linguistic items with similar distributions have similar meanings” (TensorFlow, 2019). This essentially means that Word2Vec learns the relationships between words by noting which words commonly appear together and what context they appear in.

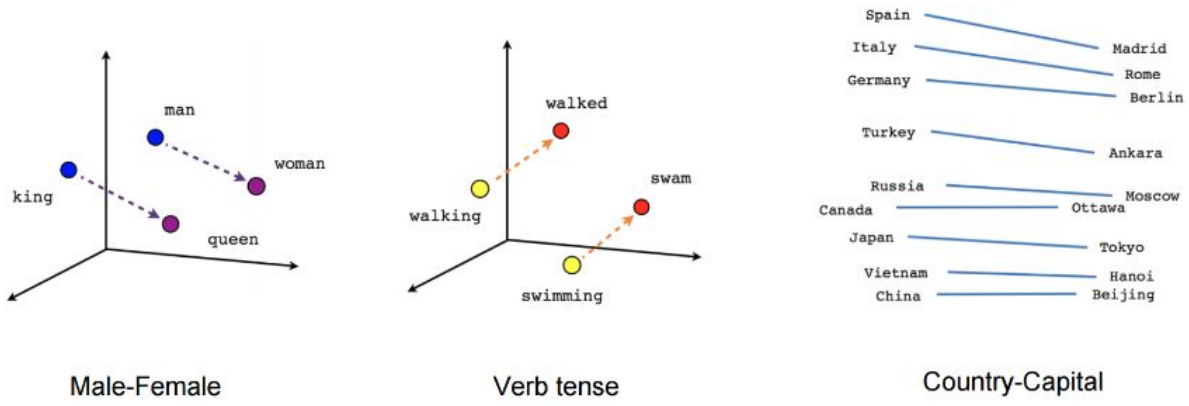


Figure 1: Relationships between words learnt by a Word2Vec model
(TensorFlow, 2019)

A trained word embedding model takes an input of a word, matches it to its pre-trained vocabulary, and outputs a numerical vector representation of the meaning of the word. For example, if a word embedding was trained and learnt two embedded meanings, size and speed, then it might convert the word “elephant” into [0.8,0.1], and the word “rabbit” into [0.3,0.7] (see figure 1 for another example). However, in this application, the embedding dimension is 100, and in reality the true meaning behind each dimension is often much less human-interpretable. With this known, how the Word2Vec model is trained can be investigated.

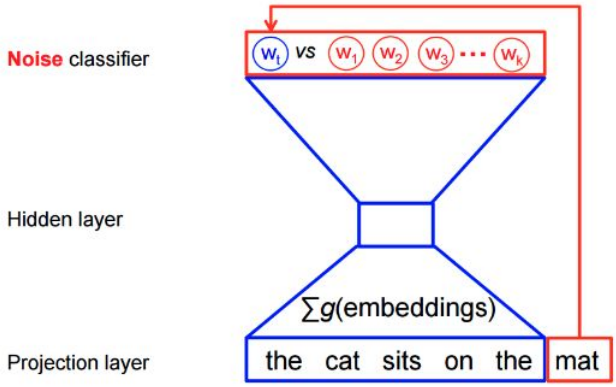


Figure 2: Word2Vec Noise-Contrastive training method
(TensorFlow, 2019)

The Word2Vec algorithm utilises a training method dubbed noise-contrastive training. It works by constructing many binary classification logistic regression problems for each sentence. An incomplete sentence is constructed from the training data, and then a “true” output, the next word in the training data’s sentence is assigned as one class for the binary output. Then, another random word that is known to not come next in the sentence is added as the other label, the “noise”. The model is then trained by minimising the probability of predicting the

“noise” word, and maximising the probability of predicting the “true” word, hence a “noise-contrastive” cost function. Over many iterations this results in a model that can accurately model the differences and “essence” of words in a much lower dimension.

A commonly used pre-trained word embedding model is the twitter GloVe (Global Vector) model. It contains a pre-trained model that was trained on two billion tweets, with a vocabulary of over 1.2 million words (Pennington, 2019). Theoretically, this model should outperform a model trained on this report’s relatively tiny vocabulary and observation size. However, in practice, the pre-trained model’s test accuracies are significantly worse than a custom-trained model. This is likely due to the pre-trained model not being able to understand the dataset-specific relationships and correlations present, in order to reliably encapsulate the data, agreeing with the conclusions of sentiment140. With this information, a Word2Vec model is trained using MatLab’s inbuilt functionality, with an embedding dimension of 100. However, since the dataset-trained embedding only has a vocabulary of 2000, on real-word high-dimensionality data the pre-trained model might perform better.

Once the word embedding model is trained, each word is projected onto the 100-dimensional word embedding vector space. However, for order-independent models (I.E anything other than a RNN), this ordering and multiple numbers of words must be “squished” into a single input for each sentence for models. Simply taking the maximum of a one-hot encoding is no longer suitable, as each word embedding is now a 100-dimensional numerical vector. There are multiple approaches to solving this problem, including evaluating the model on every word individually and taking an average of the outputted class, or taking the maximum of all the words and constructing a new vector. The method employed to perform dimensionality reduction on the data is to average the embedded vectors for each word into a new vector $\in \mathbb{R}^{100}$. This method employs a relatively low amount of information loss, however, it is by no means perfect, and its implementation may reduce the effectiveness of distribution-based methods like naive bayes. These 100-dimensional representations were spread into disjoint sets of proportions 80/10/10 for Training/Validation/Testing (Validation was only used in the LSTM model).

Naive Bayes

Naive Bayes is a simple model that is based on applying Bayes’ theorem by assuming independence between features. A gaussian Naive Bayes model was applied to our statement data. The probability of a class (*y*) given features (*x_i*) is given by

$$P(y = c|x_1, ..., x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i|y=k)}{P(x_1,...,x_n)}$$

$$P(x_i|y) = 1/\sqrt{2\pi\sigma^2{}_y}\, exp(-\frac{(x_i-\mu_y)^2}{2\sigma^2{}_y})$$

Where σ_y & μ_y are estimated using maximum likelihood.
An alternate method for treating continuous features is to bin the values to discretize the data, however this method was not employed.

SVM

Once the word embedding had been completed this lexicon was applied to the dataset using SVM. This resulted in a set of words that appear in the dataset as either positive or negative. This predicted sentiment of words in the dataset offers insight into what words in the dataset was considered positive or negative and if this was reasonable or if further investigation into the word embedding or dataset was required. In this case, the initial result was satisfactory.
After individual words were calculated, the total sentiment score for a whole statement was calculated and sorted into a binary classification of positive or negative.

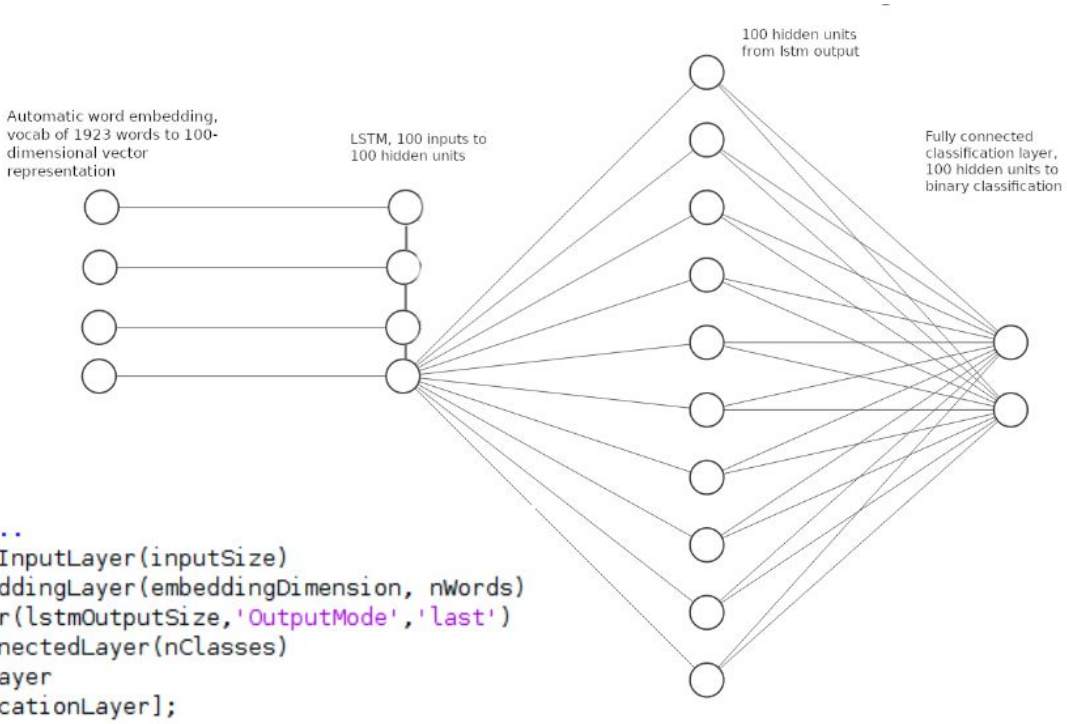
LSTM

The final model tested is a Long Short Term Memory neural network. In the context of a sequential written statement, the introduction of psuedo-memory makes sense. The memory cells of a LSTM allow a model to essentially “read” words in order, similar to how humans read text.

A LSTM is used over a traditional RNN, due to how long some sequences are. The longest sentence in the dataset is 28 words long; at this point, traditional RNNs may experience gradient explosion or vanishing with constant weight multiplication and no forget or other gates to regulate gradient flow. Additionally, the introduction of hidden states allows the information from many words to be combined into one vector in a manner much less crude than simply taking an average. There are also other benefits to a neural network model for textual classification.

As previously discussed, word embedding is a form of dimensionality reduction that allows for maximum retainment of structure in sequential text data. This report utilises a custom-trained model for the previous two classification algorithms, however, a word embedding layer can be

added to the input stage of a neural network using MatLab. This allows the word embedding weights to receive adjustments from backpropagation, essentially transforming the embedding from a PCA-like method to one much more similar to Linear Discriminant Analysis (LDA), as it is now incentivised to embed words in a manner that ensures that the two output classes are separable, making the overall model very effective.




```
layers = [ ...
    sequenceInputLayer(inputSize)
    wordEmbeddingLayer(embeddingDimension, nWords)
    lstmLayer(lstmOutputSize,'OutputMode','last')
    fullyConnectedLayer(nClasses)
    softmaxLayer
    classificationLayer];
```

Figure 3: Graphical representation of LSTM architecture

Figure 3 shows the model architecture, both as it appears in MatLab and as an approximate graphical representation. After the statement is embedded and fed through the LSTM auto-encoder, then the output is fed through a simple fully-connected layer, which outputs into a binary softmax classification. The values of various hyper-parameters can be seen in Table 1. These values were sourced from a combination of research into similar projects, and fine-tuning using the partitioned validation set.

Table 1: Hyper-Parameters / Model Training Parameters	
Variable	Value
LSTM Input Size	1 (word)
Embedding Dimension	100
Number of hidden states (LSTM output size)	100

Maximum Epochs	10
Mini-Batch Size	20 (statements)
Initial Learning Rate	0.01

Evaluation

Results and Discussion

As can be seen in the table below, the results for the three algorithm implementations vary significantly.

Results			
	SVM	Naive Bayes	LSTM
Accuracy	86.56%	61.71%	99.57%

While the SVM and LSTM algorithms appear to have achieved a high level of accuracy, well above the ~80% accuracy of Sentiment140, Naive Bayes shows significant inaccuracy. Given that the training and test data was comprised of a ~60/40 spread of positive to negative sentiments, Naive Bayes achieved an accuracy only 1% better than simply guessing positive every time. This indicates significant problems with either the data set or methodology when compared to Sentiment140's ~80% accuracy for Naive Bayes.

An explanation for the poor performance of Naive Bayes may be found in a 2003 article by J. Rennie, L. Shih, J. Teevan and D. Karger, "*One systemic problem is that when one class has more training examples than another, Naive Bayes selects poor weights for the decision boundary. This is due to an under-studied bias effect that shrinks weights for classes with few training examples,*" - given that our training data was spread approx 60/40 for positive over negative sentiments, this may explain our poor results with naive bayes. Additionally, the feature reduction method implemented may also severely inhibit the accuracy of this method.

The Long-Short Term Memory Neural Net has achieved an accuracy of 99.57% this is significantly above the ~80% accuracy achieved by Sentiment140. Such high accuracy is unexpected and it suggests that the training and test data might have very high similarity. As such it is unlikely that the model could be used in real life applications with complex vocabulary without further refinement to the dataset.

Finally, the implemented SVM's accuracy was more in line with expectations at 86.56%, however it is still slightly higher than the ~80% accuracy reported by Sentiment140. This suggests that the model may also be affected by a dataset that has high similarity, a conclusion that appears to be supported by the results of Naive Bayes and LSTM. It is again unlikely that this model could be used in real life applications without further refinement to the training data.

Conclusions and Future Works:

We have effectively shown that using short statements as training data for supervised learning it is possible to classify sentiments. Two of the implemented machine learning techniques (SVM, and LSTM) achieved high accuracy on the test data. However, Naive Bayes performed poorly due to the distribution of the training data, and LSTM may have slightly limited utility outside of its vocabulary/dataset. It is clear that word embedding is a powerful technique for manipulating text and provides a great basis for sentiment analysis.

Future work should aim to repeat the same algorithms and methodology but with improved training data with a more even distribution of positive and negative sentiments. Furthermore, the training data could be improved by including a greater variety of vocabulary and the introduction of an additional neutral sentiment class. In particular, by introducing literary devices such as negations and sarcasm to the dataset, the effective use of classification is suspected to increase greatly, even if our accuracy on the test set were to decrease.

References:

Go, A., Bhayani, R., & Huang, L. (2009). Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, *1*(12), 2009.

Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive bayes text classifiers. In *Proceedings of the 20th international conference on machine learning (ICML-03)* (pp. 616-623). http://people.csail.mit.edu/jrennie/papers/icml03-nb.pdf

TensorFlow (2019). Vector Representations of Words | TensorFlow Core | TensorFlow. [online] TensorFlow. Available at: https://www.tensorflow.org/tutorials/representation/word2vec [Accessed 7 Jun. 2019].

Pennington, J. (2019). GloVe: Global Vectors for Word Representation. [online] Nlp.stanford.edu. Available at: https://nlp.stanford.edu/projects/glove/ [Accessed 7 Jun. 2019].