Capstone Project 2

Sentiment Analysis of Amazon Reviews

Multilabel Classification with Neural Networks

Author: Vanita Kalaichelvan

May 2, 2020

Contents

Chapter 1

Introduction

1.1 Problem Statement

People are increasingly using social media to disseminate their views on products they have purchased and companies do not have a structured way of extracting this data and analysing it for sentiment. Most companies still rely on reviews being written to them or posted to their page to understand the impact of their product. This tends to be a small sample size and there are now many labelled datasets which can be used to train an in-house sentiment analysis tool for unlabelled data.

In this case, we will be creating a tool for a client selling electronic products using the Amazon electronics reviews dataset. The client will be able to use this tool to analyse the sentiment of unlabelled textual data about their products and hence, be able to make better decisions about their product.

1.2 Description of dataset

The project is based on the set of reviews provided by Amazon through their S3 service. More information on the dataset can be found here. The projects aims to use this labelled dataset to develop a tool that can perform textual sentimental analysis for other unlabelled data e.g. on twitter and other social media.

1.3 Data Extraction and Cleaning

The dataset is downloaded from the Amazon S3 bucket using **boto3**. Then, we process the dataset using the following steps:

- Check for and remove null and empty reviews and ratings
- Ensure rating labels are all integers between 1 and 5
- Remove all punctuation from the review text
- Lowercase all text
- One-hot encode target labels

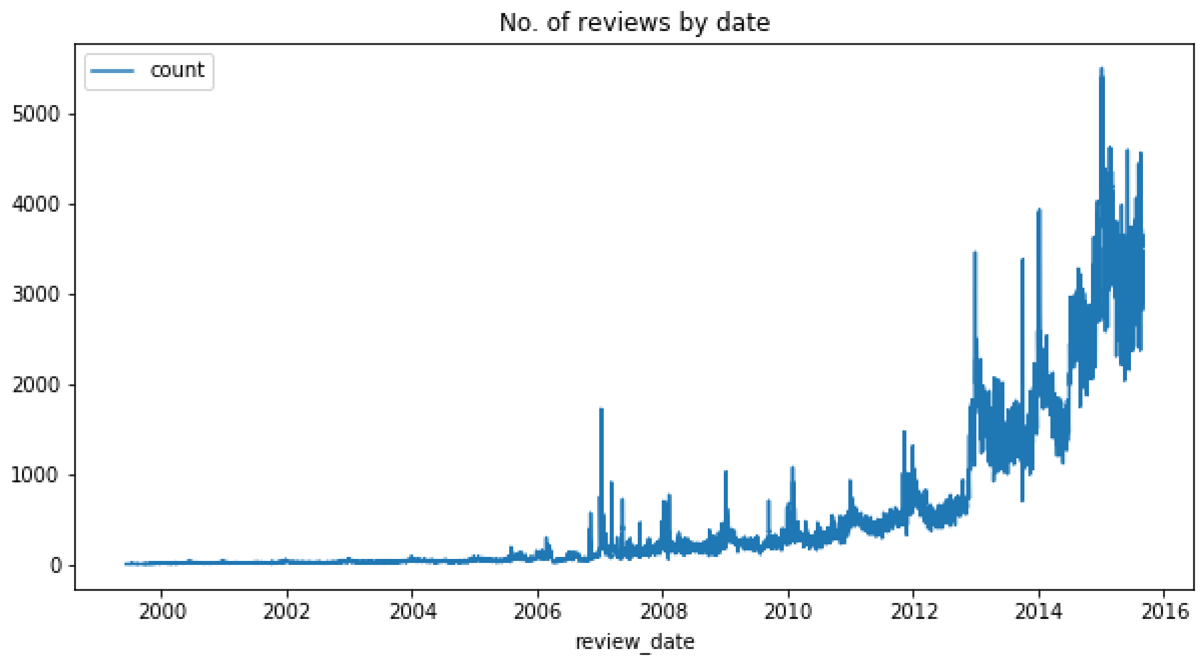
The sequences in the dataset will also have to be converted to numerical vectors. This is discussed in the next chapter, Feature Engineering.

Chapter 2

Exploratory Data Analysis

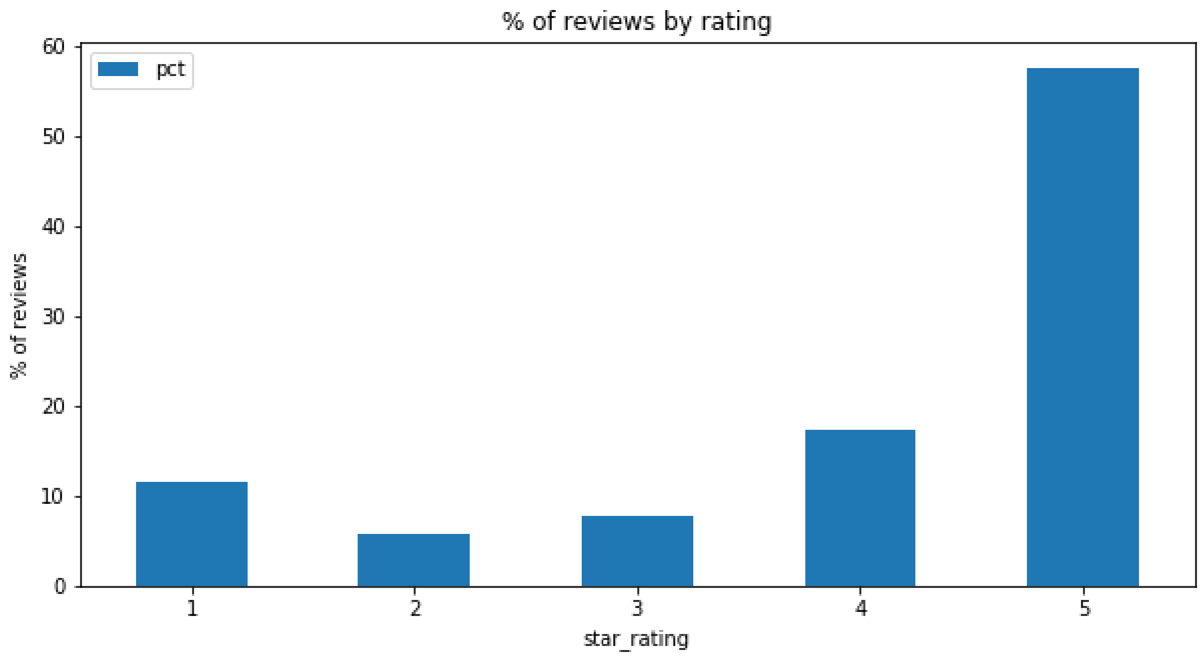
This data analysis is used to visualise the data before processing it for our model. We use this step to understand what our data constitutes of.

2.1 Number of reviews by date



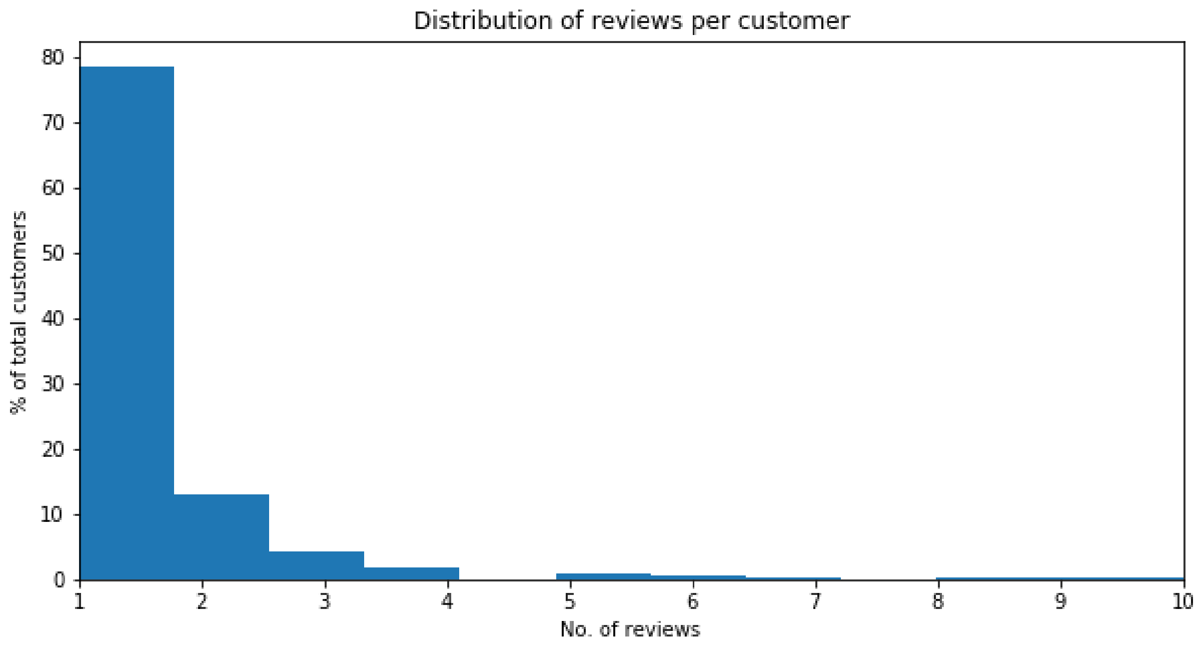
There are 3,093,660 reviews in this dataset. The dataset spans over 5,927 days from 09/06/1999 to 31/08/2015. Majority of the reviews are more recent as the use of the website increased exponentially with time.

2.2 Number of reviews by rating

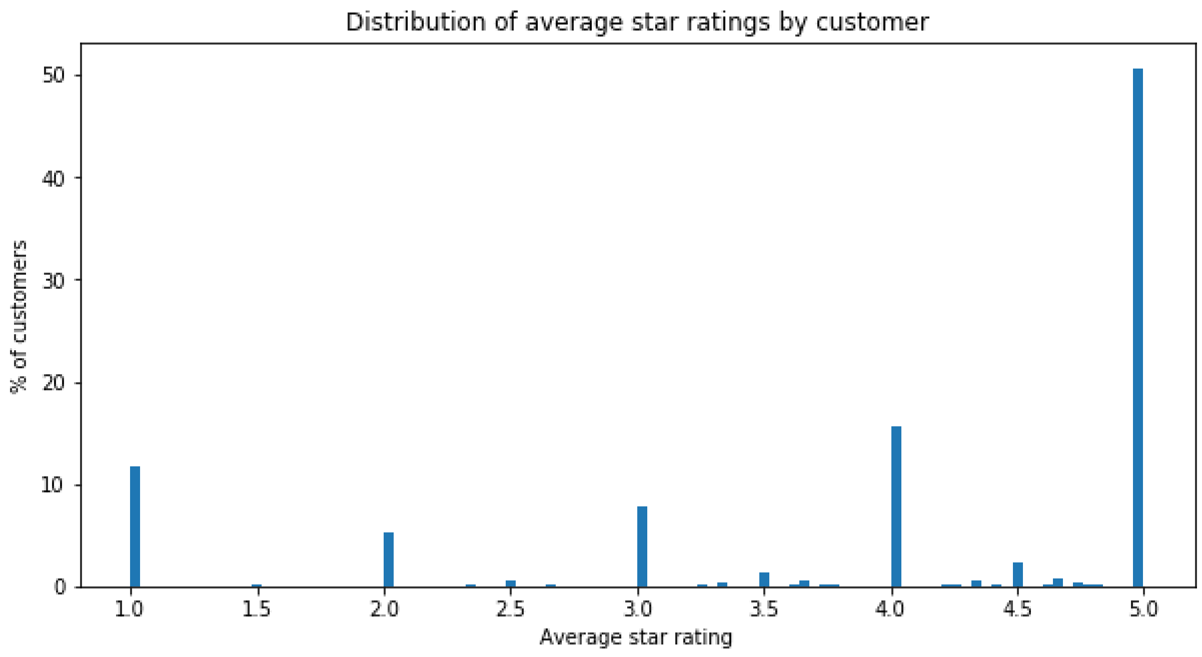


The dataset is very unbalanced with most of the reviews being positive. This is due to the way online marketplaces work where well rated products are more likely to be purchased and rated again while poorly rated products are rarely purchased.

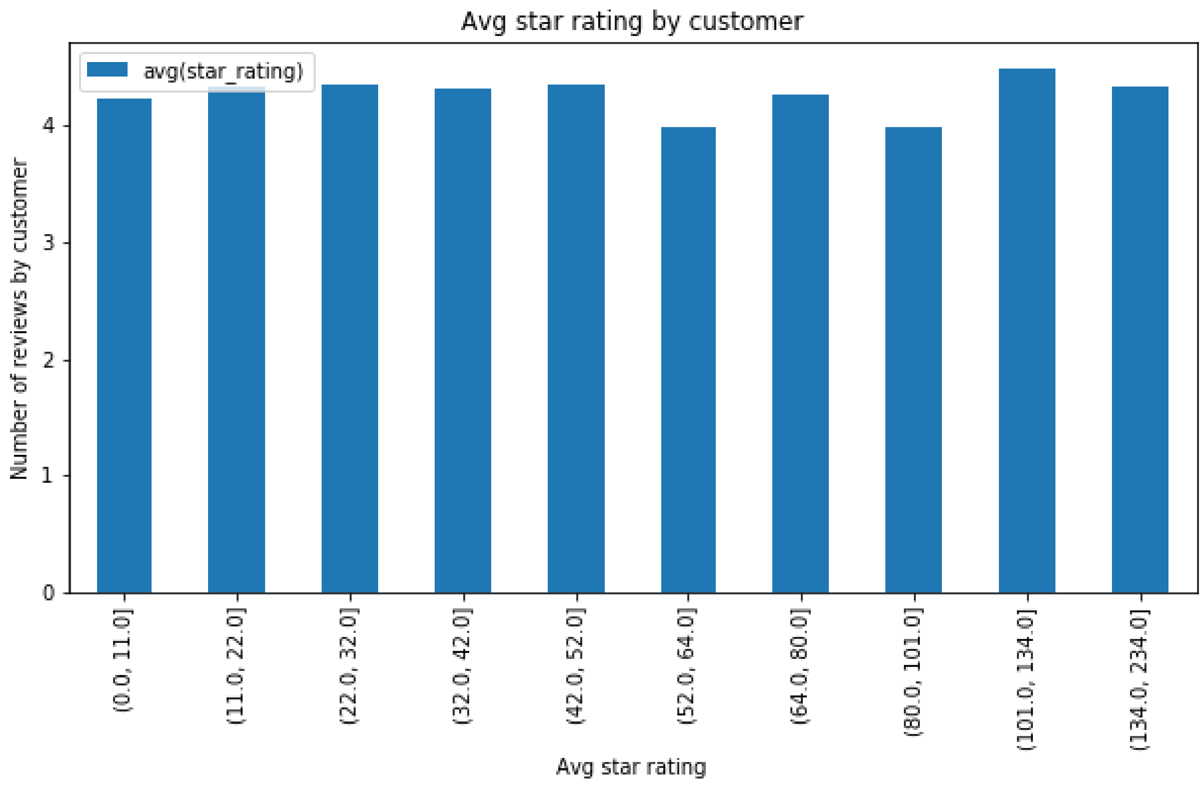
2.3 Distribution of reviews by customer



Most customers only post 1-2 reviews which is not surprising given that electronics products are low frequency purchases.

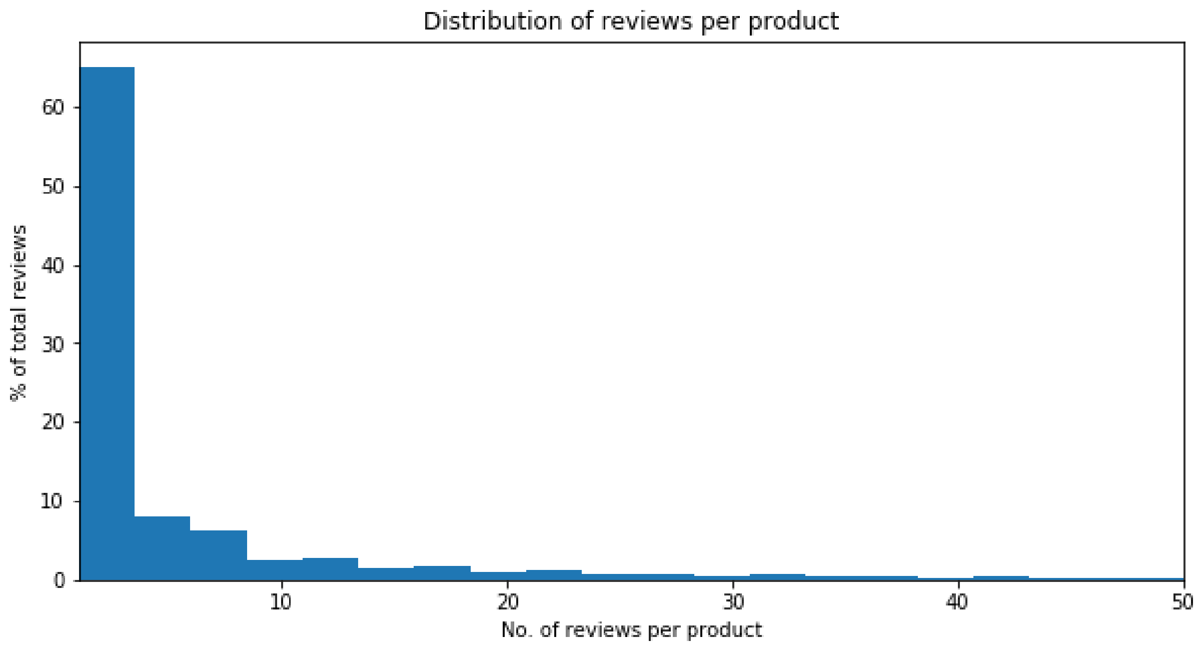


Most customers give positive ratings due to the nature of online marketplaces that operate in a circular manner. Customers are more likely to purchase products with good reviews.

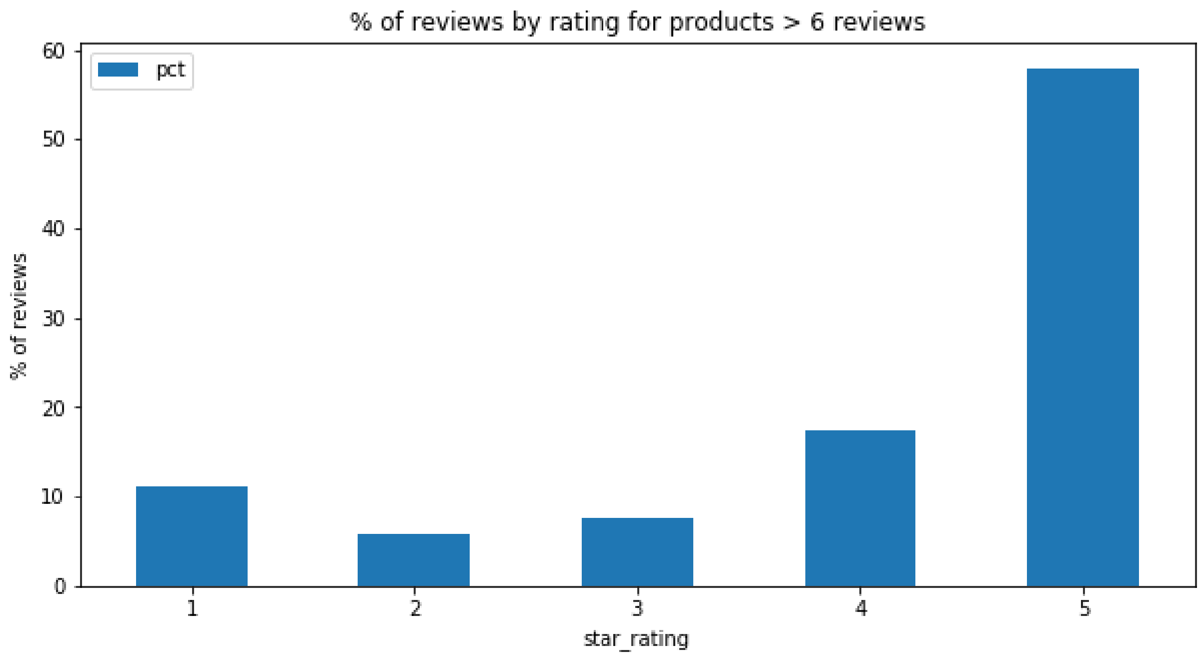


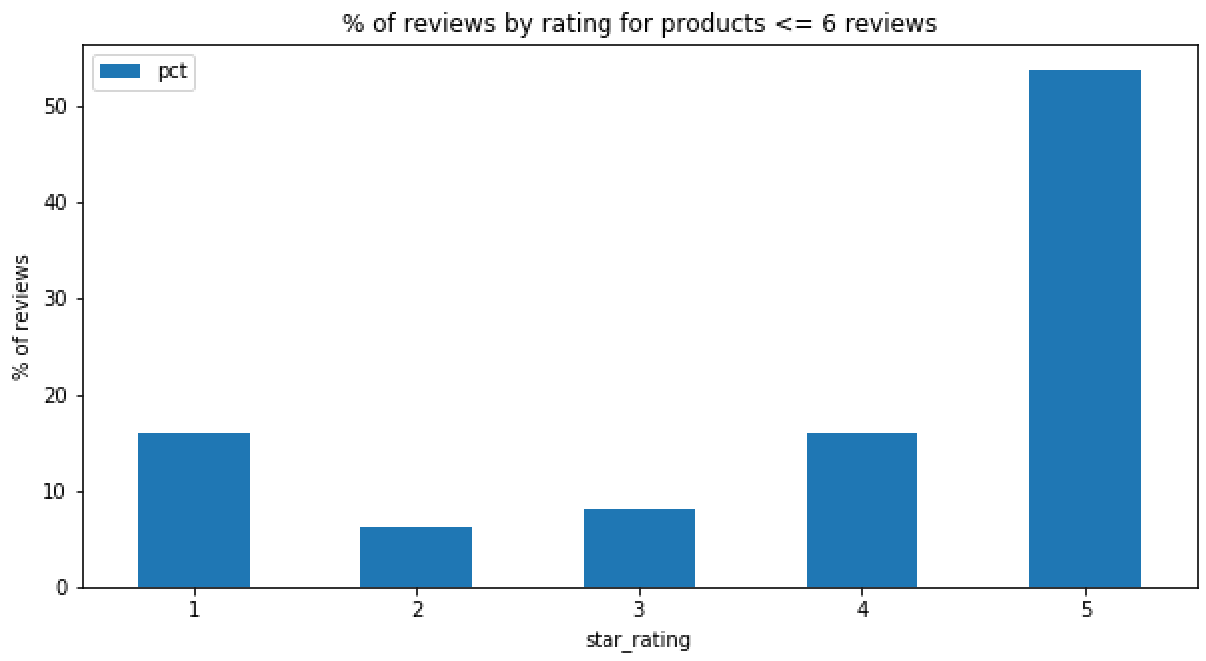
There does not seem to be a relationship between the number of reviews posted by a customer and the average rating provided.

2.4 Distribution of reviews by product

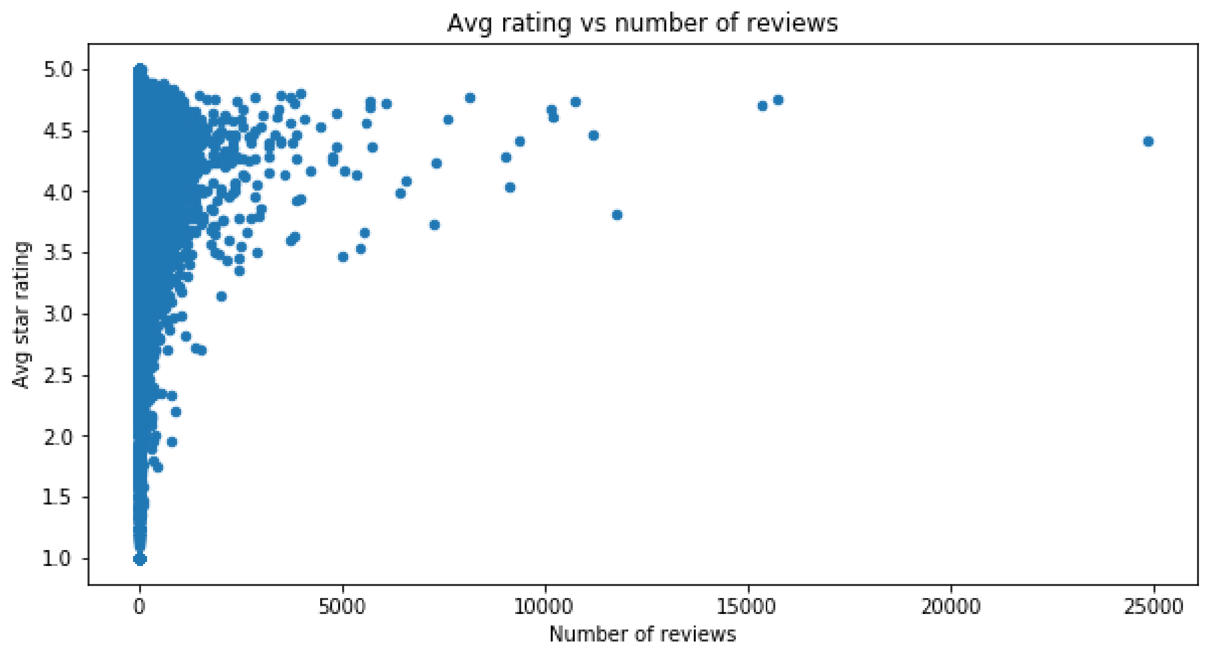


It is clear that most products only have a few reviews and a few products have a lot of reviews. This is because of how the marketplace works as explained before.

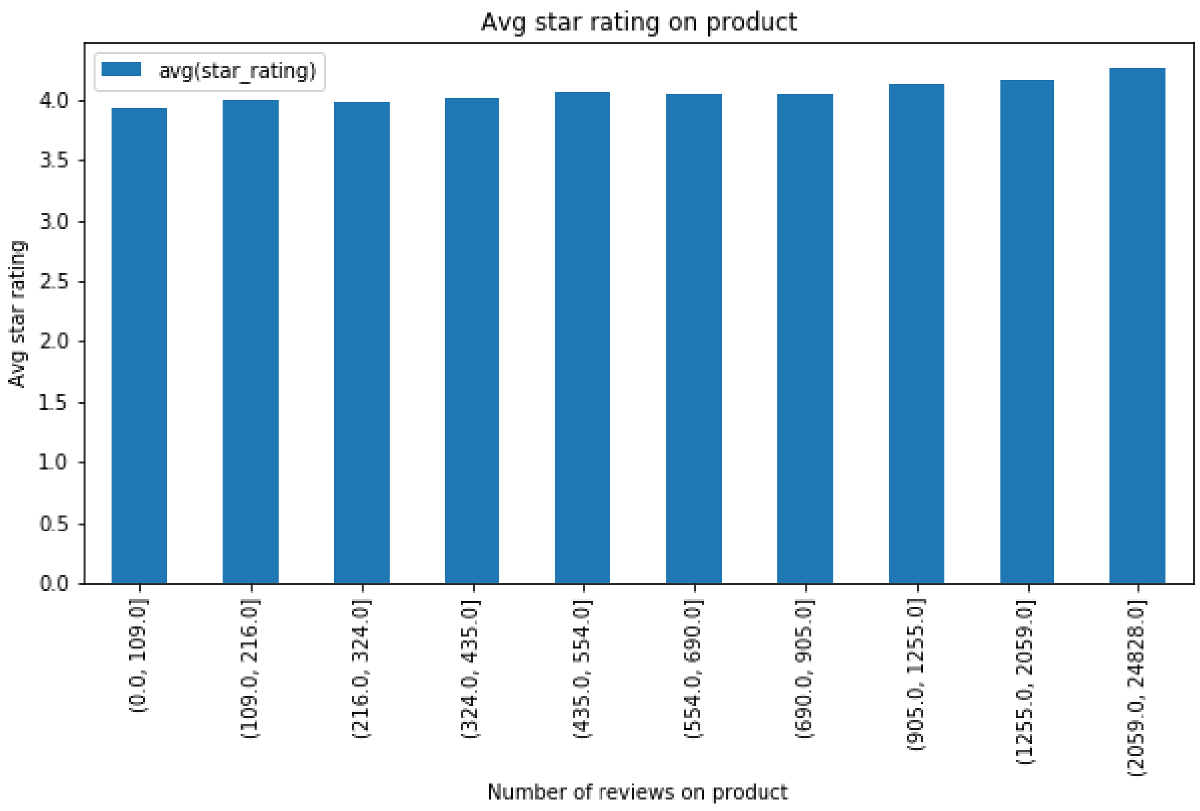




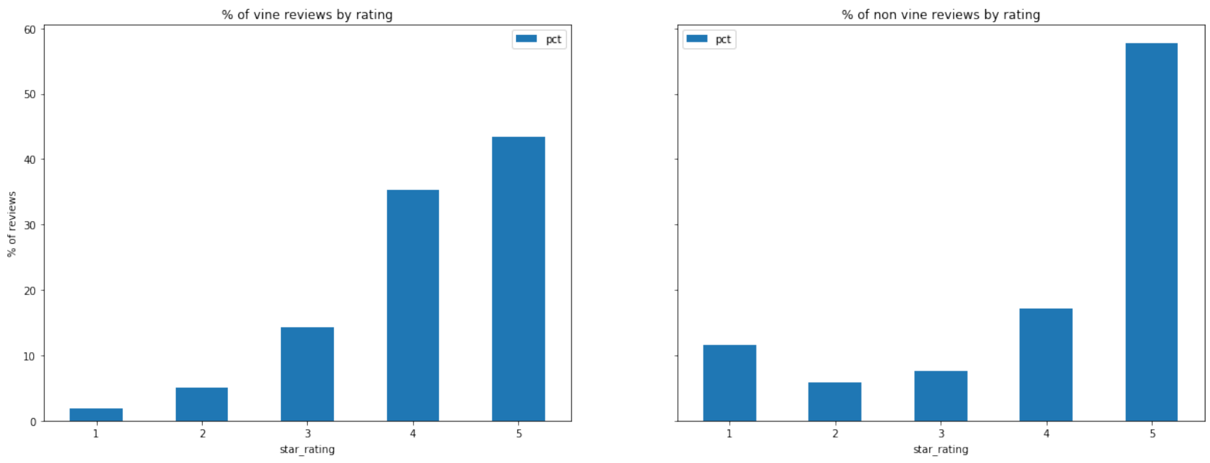
Popular products tend to have more 5 star ratings and fewer 1 star ratings which makes sense as the products are popular because they are good. This is a somewhat circular relationship. We can directly look at the correlation between average rating and number of reviews below. Although overall correlation is insignificant at 0.02, it is clear that the really popular products with lots of reviews have an average star rating of 4-5.

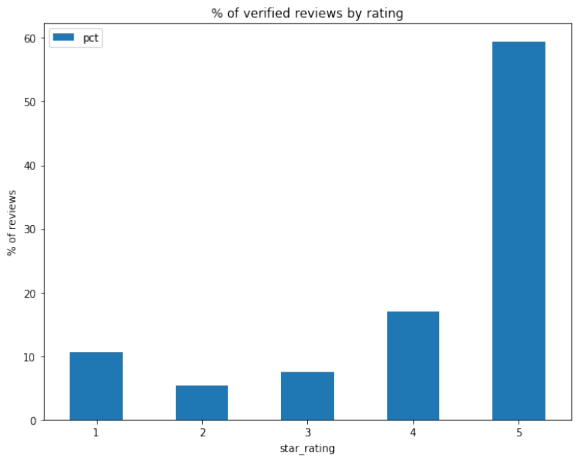


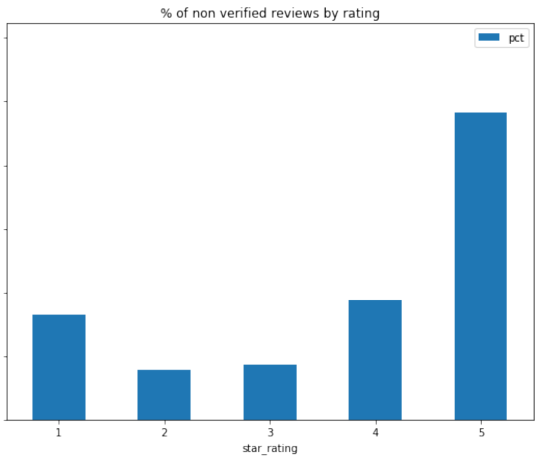
We can also look at average star rating of products by number of reviews on product. We see that this increases slightly, however, the differences in sample sizes create somewhat of a selection bias.



2.5 Distribution of ratings for vine reviews and verified purchases

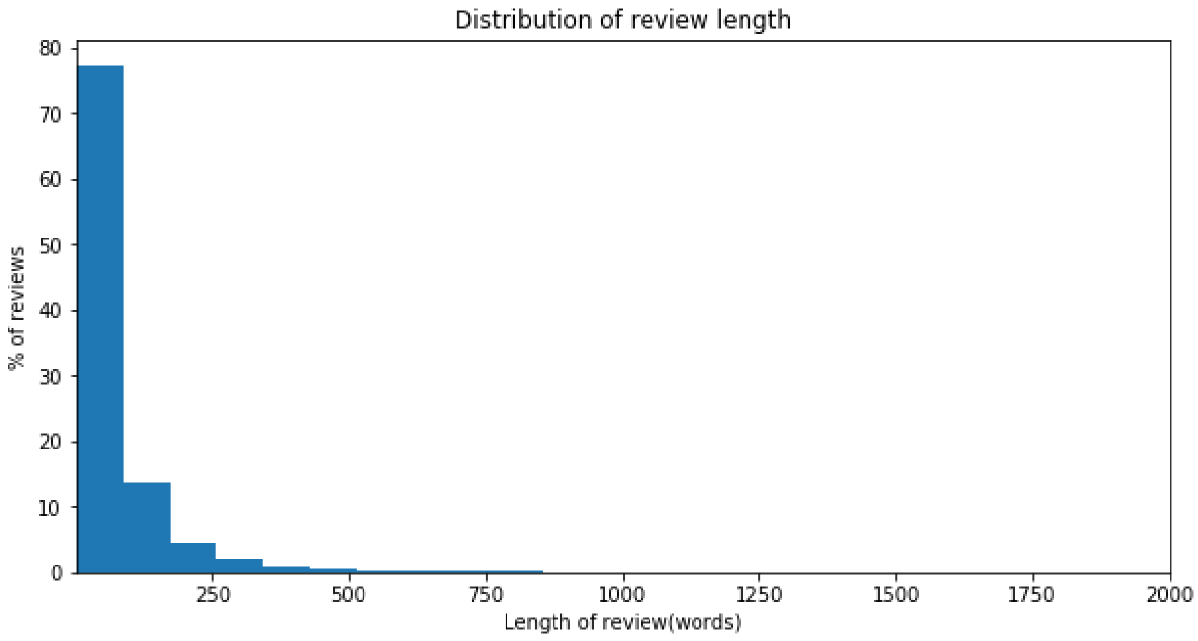


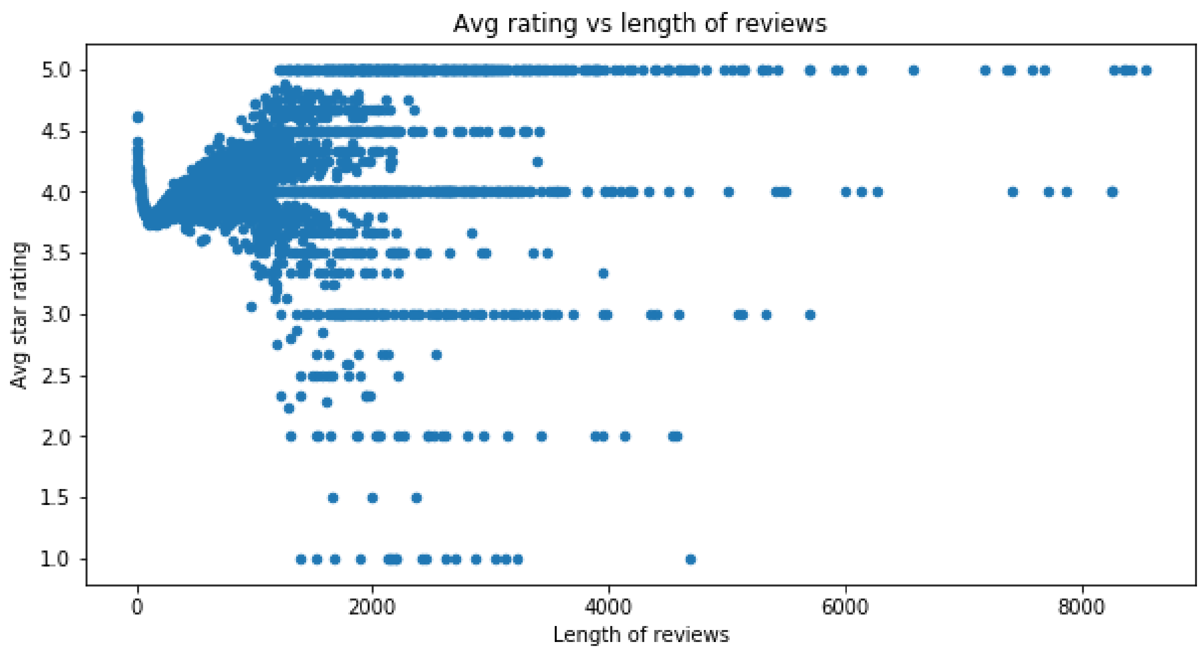




Looking at the results, it shows that vine reviews and verified reviews are a lot less negative. Again, in both cases, we are comparing non equal datasets which could affect the results through selection bias.

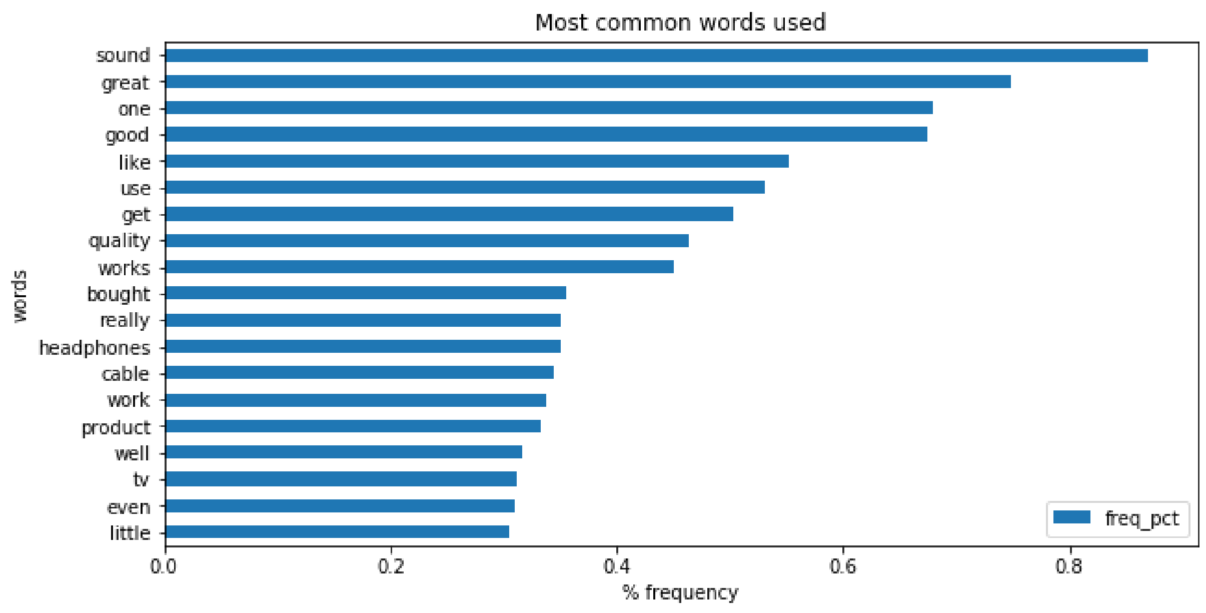
2.6 Distribution of ratings by length of review





Most reviews are less than 250 words long with a few extremely long reviews for 4-5 star ratings. This could be because there are lot more 4-5 star rated reviews, hence there is a larger variance in review length. No obvious correlation between length of review and rating given at 0.17.

2.7 Most frequently used words



The most popular words are mostly either descriptive of the product (e.g. good, great, like) or segment specific (e.g. sound-mostly for headphones/earphones, cable, headphones, tv).

2.8 Statistical Tests

We can do some statistical analysis to see if product ratings are independent of products. We will randomly choose 2 products and perform a chi-squared test on the ratings distribution. Here we use chi-squared as the ratings distribution is discrete and does not allow itself for other tests such as t-test or KS test. We will also limit the runs of the trial to 20 to limit execution time. This test rejects the null hypothesis that there is no difference in the 2 distributions with 95% confidence interval 17 times out of 20 random tests.

Based on the random sampling of products reviews, it is clear that the ratings provided to products are mostly independent of each other and it is unlikely that 2 different products have similar distribution in ratings.

Chapter 3

Feature Engineering

Since textual data cannot be fed directly into a model, we first have to transform the textual features into numerical features. This can be done either by tokenization based on different algorithms or using word embeddings which are learned representations of words where words with same meaning have similar representations.

3.1 Tokenization

3.1.1 TF-IDF

The first tokenization method used is known as TF-IDF(term frequency-inverse document frequency). The idea behind this algorithm is to weight frequently occurring terms in a sequence highly unless they occur frequently throughout the document. This enables the algorithms to give low importance to commonly used words that provide no meaning while capturing keywords that indicate sentiment.

Mathematically, the score for each word in a sequence is defined as

$$TFIDF(t,d,D) = TF(t,d) \times IDF(t,D)$$
$$IDF(t,D) = log\frac{|D|+1}{DF(t,D)+1}$$

where $|D|$ is the total number of documents, $TF(t,d)$ is the number of times a term, t appears in a sequence, d and $DF(t,D)$ is the number of times the term appears in the document, D. We can see an example below which shows how a document of sequences can be transformed using TF-IDF.

$$\begin{bmatrix} \text{This was a great product} \\ \text{This was a bad product} \\ \text{This was ok} \end{bmatrix} \overset{\text{TF-IDF}}{\Rightarrow} \begin{bmatrix} 0.15 & 0.15 & 0.15 & 0.30 & 0.20 \\ 0.15 & 0.15 & 0.15 & 0.30 & 0.20 \\ 0.15 & 0.15 & 0.30 & 0 & 0 \end{bmatrix}$$

3.1.2 N-grams

The second tokenization method used is a more advanced version of the first where we implement both TF-IDF and N-grams. This allows us to not just consider each word independently but also take into account the word before and after it.

	sentence	tokens
N = 1	This is a sentence	[This, is, a, sentence]
N = 2	This is a sentence	[This is, is a, a sentence]
N = 3	This is a sentence	[This is a, is a sentence]

Table 3.1: Table showing n-grams

Table 3.1 shows how tokens are formed for a sequence based on the n-gram value chosen. In a basic tokenization method, only the case N=1 is considered where each word becomes a token. This can be improved by using bi-grams, as seen in the example for N=2, and tri-grams, as seen in the example for N=3 as it allows the model to also analyse the relationship between each word and the words around it.

3.2 Word Embeddings

Word embeddings produce a dense representation of each word by mapping each word to a vector where vector values are learned based on the usage of words. For example, words with similar meanings will have vectors close to each other and relationships between words can be defined by the distances between vectors.

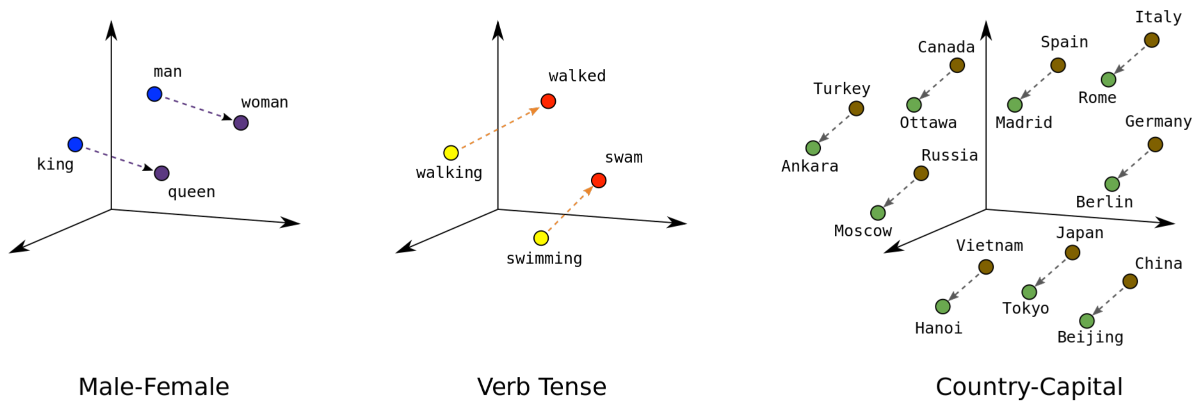


Figure 3.1: Vector representations of words in an embedding space[1]

We can implement word embeddings by using a pre-trained embedding or by training an embedding layer during model implementation. The embedding layer takes an input of a 2D vector of shape (samples, sequence length) and outputs a 3D vector of shape (samples, sequence length, embedding dimension). Using pre-trained embeddings significantly reduces training time but may not perform as well for the task.

3.2.1 GloVe

GloVe[2] is an unsupervised algorithm used to obtain word embeddings based on word to word co-occurrence statistics. Given a sentence, the training algorithm selects a word to be the target and a certain number of words around it(based on window size) to be the context. The training objective is to minimize the difference between the dot product of the target and context vectors and logarithm of the probability of co-occurrence between the target and context. This is similar to the Word2Vec model except that the GloVe model has global co-occurrence information and implements gradient descent instead of neural networks to train its model. In this project, having implemented models using GloVe word embeddings, it was seen that training the embedding layer(i.e not using pre-trained word embedding such as GloVe) had a performance advantage and a small time disadvantage. Hence, it was decided that the embedding layer will be trained with the model.

Chapter 4

Machine Learning

4.1 Neural Network Models

All the models implemented in this project are neural-network based due to the nature of the project being focused on the learning of deep learning methods.

4.1.1 Neural Networks

A neural network is a deep machine learning method made up of layers of nodes. The input layer takes in the features of the dataset and passes it on to the hidden layers of the model after multiplying it by a weight and passing it through an activation function which helps to bound the output from each node. The activation function also makes it possible for the model to be non-linear so as to handle more complex tasks. The final layer of the network, known as the output layer, sums the outputs from the hidden layer and passes it through an activation function for the final output of the model.

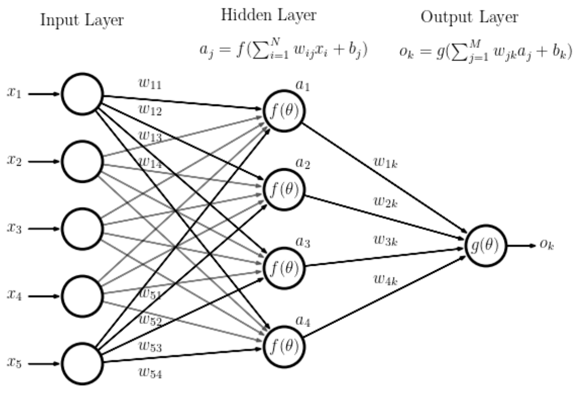


Figure 4.1: Neural Network Model[3]

During training, the model uses back propagation to update the weights such as to minimise the loss function. Neural networks are complex models with very low interpretability. However, they are very powerful and can learn many abstract features.

4.1.2 Recurrent Neural Networks

Simple neural networks tend to treat each feature independently which generally makes it less powerful for textual datasets where te relationships between the words in the sentences are equally important. Recurrent neural networks(RNN) have memory of words occurring previously as each node can be unrolled in time with the output at time, t is dependent on the input at time, t and the state at time, t. The state at time, t is a function of all the previous outputs till time, t-1.

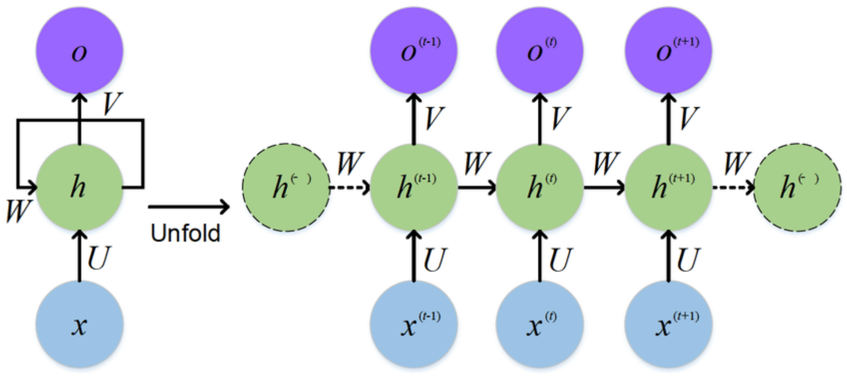


Figure 4.2: Unrolling RNN[4]

RNNs are very useful in sequence classification problems but are very susceptible to the vanishing/exploding gradient problem. With some derivation, we can show that the change in error term with respect to the weights is proportional to the change in the current state with respect to the previous state and is given by $W \cdot \sigma(.)$ where $\sigma(.)$ is bounded between 0 and 1. Over many time steps, the update becomes directly dependent on $\prod_{t=1}^{T} W$. This creates a problem when the weights are too small or too large causing the gradient to vanish or explode, hence making updating of the weights very difficult for large sequences. This problem is handled by the long-short term memory networks(LSTM).

Bidirectional RNNs

RNNs are usually trained on a sequence only in one direction. It is possible to train the sequence on both directions. The model might learn different representations of the words when training in different directions and combining them could improve performance of the model. For some sequence problems, the direction in which the sequence is processed can also significantly affect the results. This occurs when the chronological order of the sequence is important for the problem. In our case, the chronological order in which the words are processed does not matter. It has also been shown that the implementation of bidirectional RNNs has had no impact on performance.

4.1.3 LSTM Neural Networks

The LSTM architecture deals with the issue of vanishing gradients and allows for longer term memory using 3 gates, namely the input gate, output gate and forget gate. The forget gate controls what information can be forgotten based on previous information and new information and the input gate determines what information will be carried forward to the next cell. The forget gate allows for long-term memory which is usually not implemented well by the RNN. The use of multiple gating functions also means that the network has more flexibility to adjust the functions and its corresponding weights to handle the vanishing gradient problem.

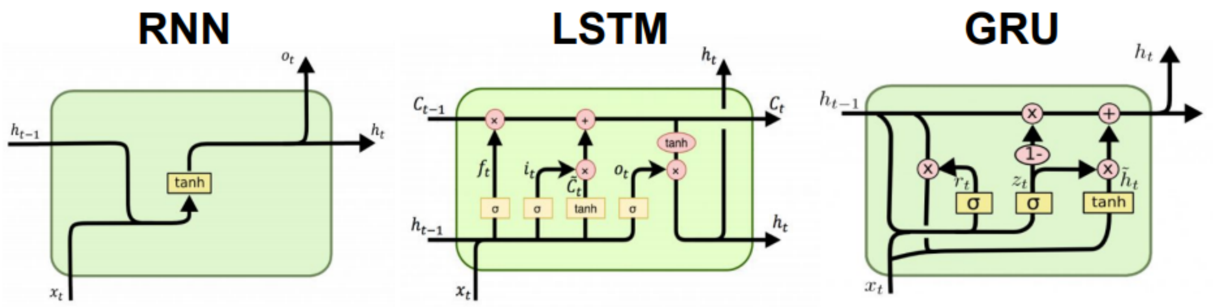


Figure 4.3: Cell Architecture of RNN, LSTM and GRU[5]

A less complex implementation of the LSTM can be done using the gated recurrent unit(GRU). The GRU only has 2 gates, namely the reset gate and update gate. The reset gate controls both how much memory to forget and how much new information to add. The GRU also does not use a seperate carry state to retain long term information and combines this with the hidden state, h_t. GRUs generally perform on par with LSTMs and train much more quickly but can be less effective on problems with long sequences. For the dataset in the project, it was shown that LSTMs generally performed better than GRUs.

4.1.4 Convolutional Neural Networks

Convolutional neural networks(CNNs) are very popular for image classificaiton problems but 1D CNNs can also be used for sequence classification. CNNs learn representations of a sequence by convolving it with a filter of a pre-determined size. The output from this stage is based on how similar the input is to the filter. The output is then pooled to reduce the dimensionality. CNNs can be used independently in sentiment classification or with RNNs and LSTMs. Since CNNs learn the representation of the words independently, they incur the same problem as simple neural network in that they cannot understand the relationships between the words. However, since they are less computationally exhaustive, they can be used as a pre-processing step to shorten long sequence problems before being fed into a RNN/LSTM. The performance from a neural network with a 1D-CNN layer followed by a LSTM layer was lower than that of one with stacked LSTMs. Hence, we did not explore this opportunity further.

4.2 Model Evaluation

4.2.1 Model Implementation

For our dataset, we have chosen the following models for our final discussion.

1. TF-IDF tokenization with Neural Network
2. TF-IDF and n-grams tokenization with Neural Network
3. Word Embeddings with LSTM Neural Network

The training data is limited to 70K rows and our testing data to 30K rows. The model is trained on Google Colab due to the access to a GPU which allows the model to train much faster. Tensorflow libraries are used to implement all the models. The models are trained using the following variables:

- number of labels: 5
- maximum words for tokenization: 10K
- embedding dimension: 100
- optimizer function:Adam
- loss function: Categorical Crossentropy
- validation metrics: Accuracy
- batch size: 512
- validation split: 10%
- epochs: 50

Early stopping is also used to stop the model from training when validation loss has stalled for a number of epochs. Regularization methods such as kernel regularizer and dropout have also been utilised to prevent overfitting issues.

4.2.2 Model Architecture

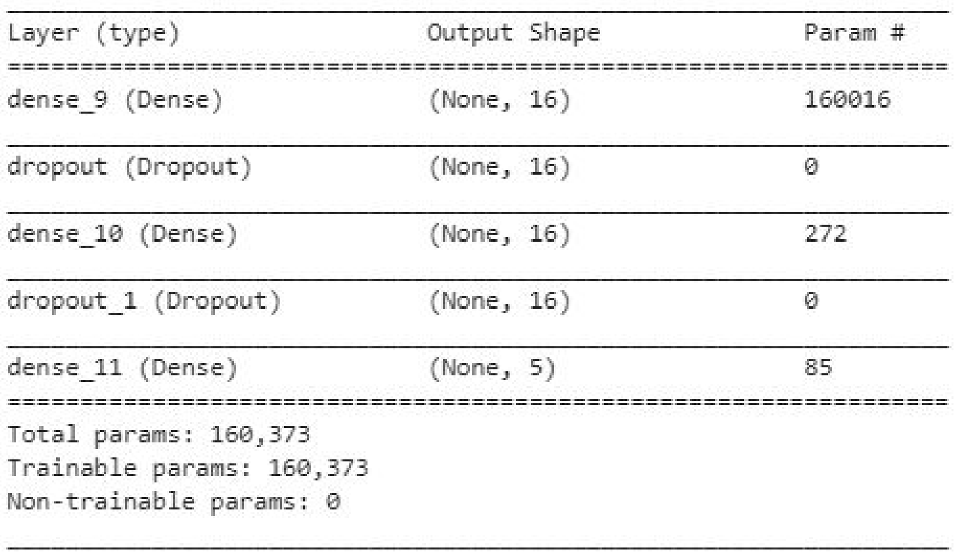
Model 1: TF-IDF tokenization with Simple NN

Listing 4.1: Code showing implementation of model

```
tokenizer, X_tokenized = tokenize_words(max_words, X) # uses TF-IDF tokenization
y_encoded = to_categorical(y-1, num_classes=5)
X_train, X_test, y_train, y_test = train_test_split(X_tokenized, y_encoded, test_size
    =0.3, random_state=0)

model_dout = models.Sequential()
model_dout.add(layers.Dense(16, activation='relu', input_shape=(max_words,)))
model_dout.add(layers.Dropout(0.4))
model_dout.add(layers.Dense(16, activation='relu'))
model_dout.add(layers.Dropout(0.4))
model_dout.add(layers.Dense(num_labels, activation='softmax'))
model_dout.compile(optimizer=optimizer_function, loss=loss_function, metrics=
    model_metrics)
```

The first model is implemented using the code shown above. There are 2 hidden layers with 16 nodes each and an output layer with 5 nodes to predict the probability of each class. The dropout layers are placed between the hidden layers to prevent overfitting. The dropout layers work by randomly setting some fraction of the node outputs to 0 at each update during training time. The activation function for the hidden layers is chosen to be a rectified linear unit (ReLU) due to its simple nature which makes it much quicker to implement than other activation functions. The activation function for the output layer is chosen to be the softmax function which outputs a probability of occurrence such that the values of all outputs sum to 1. The model architecture with number of trainable parameters is outlined below.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_9 (Dense)              (None, 16)                160016
_________________________________________________________________
dropout (Dropout)            (None, 16)                0
_________________________________________________________________
dense_10 (Dense)             (None, 16)                272
_________________________________________________________________
dropout_1 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_11 (Dense)             (None, 5)                 85
=================================================================
Total params: 160,373
Trainable params: 160,373
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.4: Model architecture

Model 2: TF-IDF and N-grams tokenization with Simple NN

Listing 4.2: Code showing implementation of model

```
vectorizer = TfidfVectorizer(ngram_range=(1, 2), min_df=0, max_features=max_words)
X_ngrams = vectorizer.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X_ngrams.toarray(), y_encoded,
    test_size=0.3, random_state=0)

model_ngrams = models.Sequential()
model_ngrams.add(layers.Dense(16, activation='relu', input_shape=(max_words,)))
model_ngrams.add(layers.Dropout(0.4))
model_ngrams.add(layers.Dense(16, activation='relu'))
model_ngrams.add(layers.Dropout(0.4))
model_ngrams.add(layers.Dense(num_labels, activation='softmax'))
model_ngrams.compile(optimizer=optimizer_function, loss=loss_function, metrics=
    model_metrics)
```

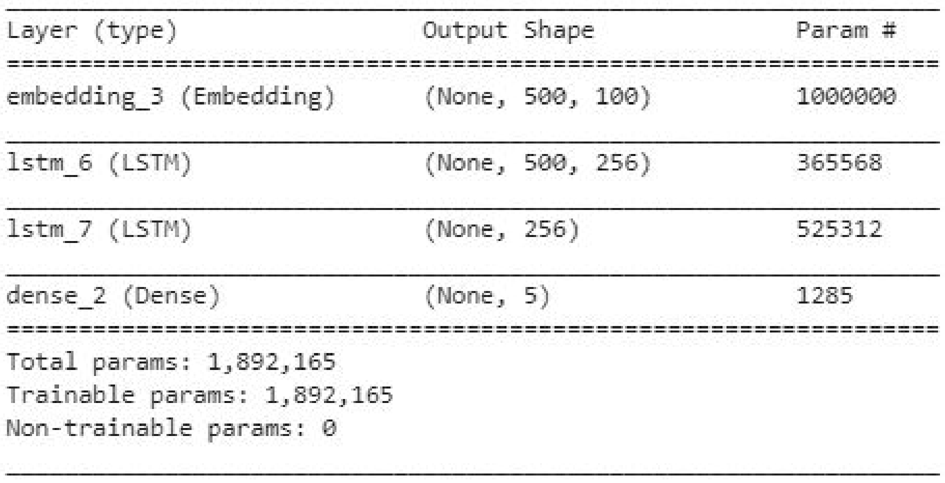
The 2nd model is implemented with the same model architecture as the first model as seen in Figure 4.4. The only difference from the 1st model is the way the tokenization is done. Here, we define the TF-IDF vectorizer with unigrams and bigrams. We can also add trigrams, however, this will increase the complexity and training time of the model. During implementation, the use of trigrams did not have a significant enough impact on the results to warrant it.

Model 3: Word embeddings with LSTM NN

Listing 4.3: Code showing implementation of model

```
model_complex = models.Sequential()
model_complex.add(layers.Embedding(max_words, embedding_dim, input_length=max_len))
model_complex.add(layers.LSTM(256, kernel_regularizer=regularizers.l2(0.002),
    return_sequences=True))
model_complex.add(layers.LSTM(256, kernel_regularizer=regularizers.l2(0.002)))
model_complex.add(layers.Dense(num_labels, activation='softmax'))
model_complex.compile(optimizer=optimizer_function, loss=loss_function, metrics=
    model_metrics)
```

The final model implemented uses an embedding layer, followed by 2 hidden LSTM layers with 256 nodes. As seen from the model summary below, the number of trainable parameters has significantly increased with the number of nodes and the use of LSTM layers where each unit has multiple trainable parameters that control the ouptut and the flow of information through time.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 500, 100)          1000000
_________________________________________________________________
lstm_6 (LSTM)                (None, 500, 256)          365568
_________________________________________________________________
lstm_7 (LSTM)                (None, 256)               525312
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 1285
=================================================================
Total params: 1,892,165
Trainable params: 1,892,165
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.5: Model architecture

4.2.3 Training Results

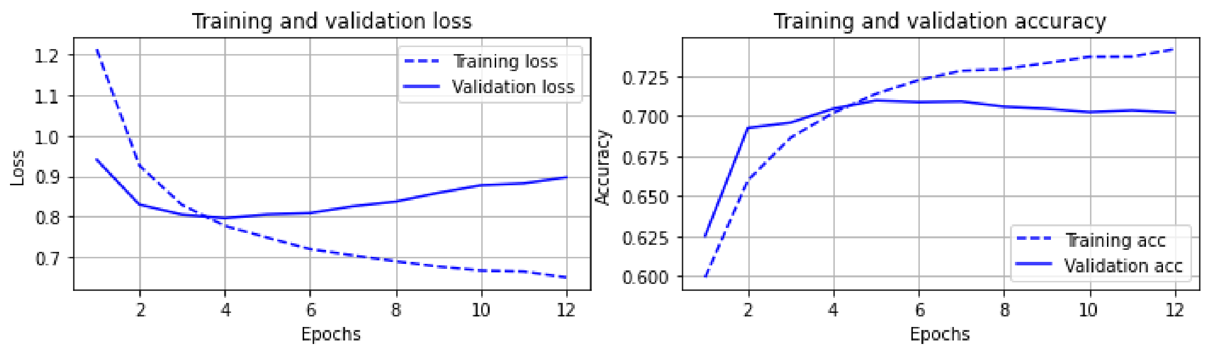


Figure 4.6: Training results for TF-IDF model

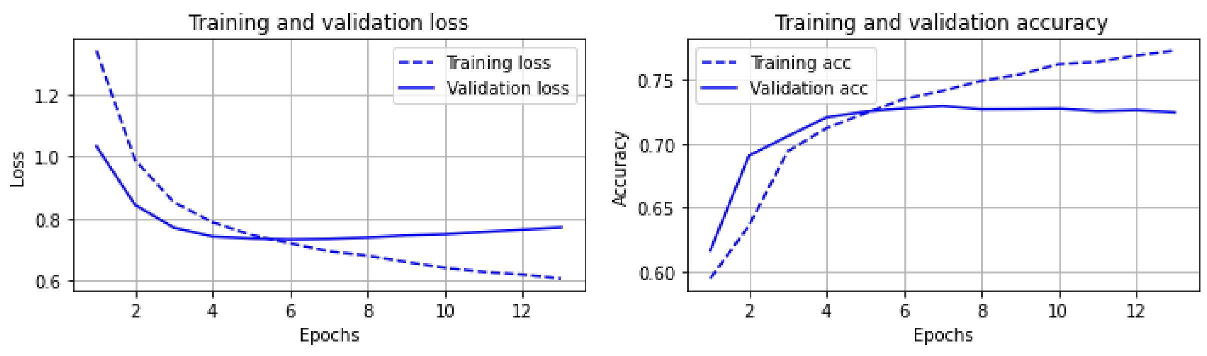


Figure 4.7: Training results for TF-IDF with N-grams model

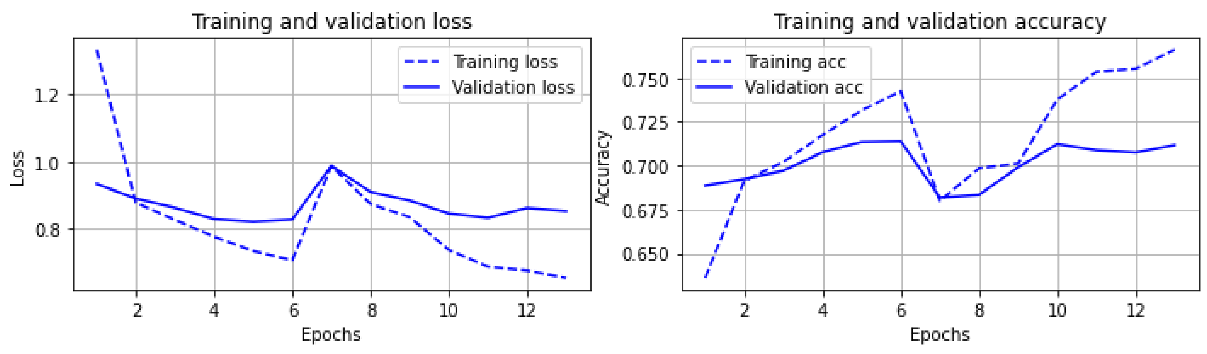
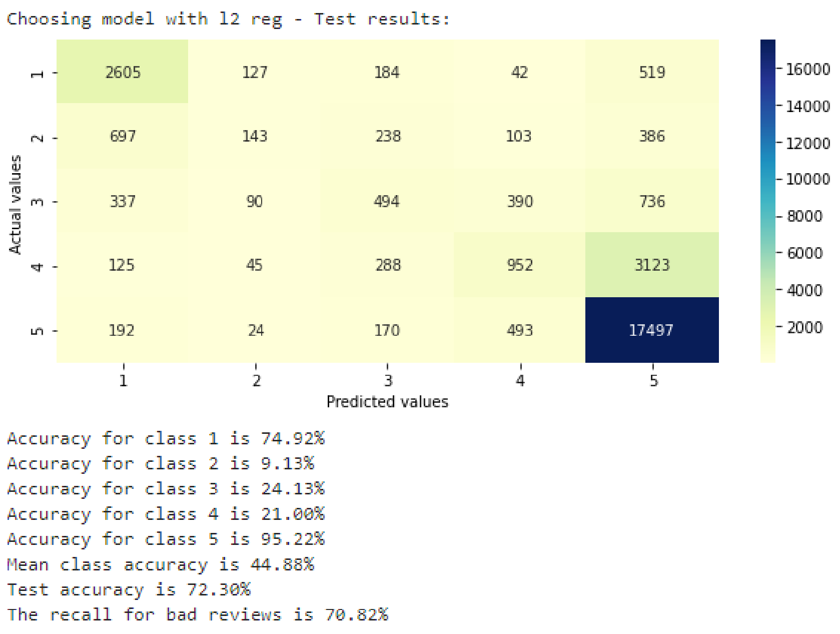


Figure 4.8: Training results for LSTM model

As seen from the figures above, the use of n-grams improves performance. However, the LSTM model performs on par with the other models despite having a more complex implementation.

4.2.4 Testing Results



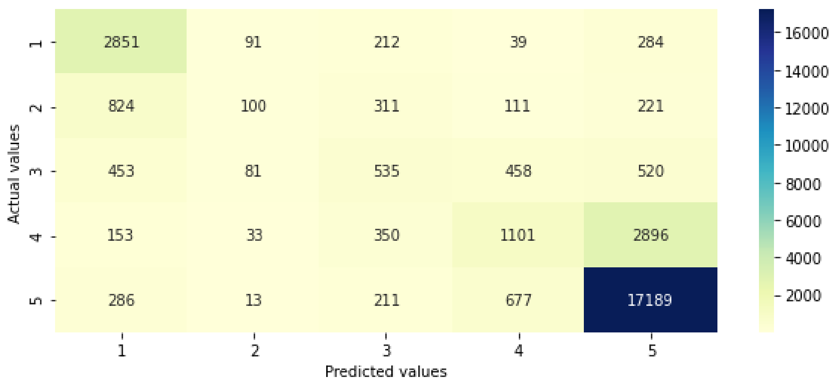
```
Choosing model with l2 reg - Test results:
```

Actual values \ Predicted values	1	2	3	4	5
1	2605	127	184	42	519
2	697	143	238	103	386
3	337	90	494	390	736
4	125	45	288	952	3123
5	192	24	170	493	17497

```
Accuracy for class 1 is 74.92%
Accuracy for class 2 is 9.13%
Accuracy for class 3 is 24.13%
Accuracy for class 4 is 21.00%
Accuracy for class 5 is 95.22%
Mean class accuracy is 44.88%
Test accuracy is 72.30%
The recall for bad reviews is 70.82%
```

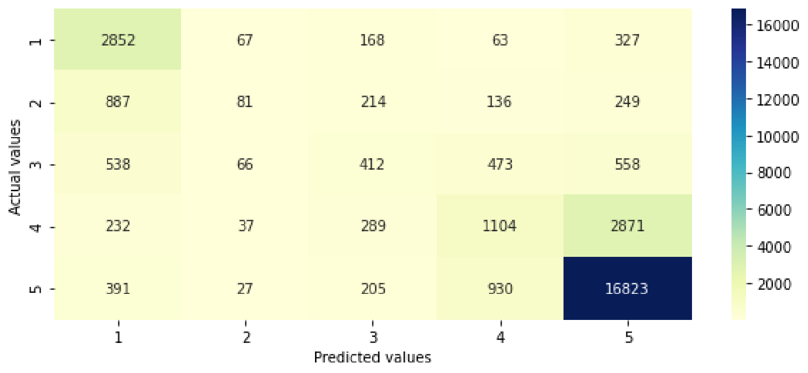
Figure 4.9: Test results for TF-IDF model

```
Test results:
```



```
Accuracy for class 1 is 82.00%
Accuracy for class 2 is 6.38%
Accuracy for class 3 is 26.14%
Accuracy for class 4 is 24.29%
Accuracy for class 5 is 93.54%
Mean class accuracy is 46.47%
Test accuracy is 72.59%
The recall for bad reviews is 76.65%
```

Figure 4.10: Test results for TF-IDF with N-grams model



```
Accuracy for class 1 is 82.02%
Accuracy for class 2 is 5.17%
Accuracy for class 3 is 20.13%
Accuracy for class 4 is 24.35%
Accuracy for class 5 is 91.55%
Mean class accuracy is 44.64%
Test accuracy is 70.91%
The recall for bad reviews is 77.06%
```

Figure 4.11: Test results for LSTM model

Based on the results on the test set, we can see that the best performing model is the TF-IDF with N-grams model. The LSTM model has the worst accuracy but has the best recall on bad reviews. This means it is the best model at not classifying bad reviews as good ones. For our project case, this is the most important metric as we want the business to be responsive to bad reviews. However, the TF-IDF with N-grams model is not far behind and trains much faster.

Chapter 5

Conclusion & Future Work

The project has shown that more complex implementation is not necessarily always better. It is possible that the more complex models would have outperformed if the problem was a binary one. It is also possible that not enough has been done to tune the hyper parameters. Tuning hyper parameters of neural networks can be a very iterative and exhaustive process especially when it is difficult to understand how the features are influencing the model.

Possible future work to undertake:

- Implementation of BERT model
- Fine tuning of existing models
- Using other machine learning methods such as Naive Bayes, SVM for text classification
- Reintroducing the problem as a binary one

Bibliography

[1] Embeddings: Translating to a lower-dimensional space. https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[3] Convolutional neural netowrks: Training an image classifier with keras. http://www.datastuff.tech/machine-learning/convolutional-neural-networks-an-introduction-tensorflow-eager/.

[4] Weijiang Feng, Naiyang Guan, Yuan Li, Xiang Zhang, and Zhigang Luo. Audio visual speech recognition with multimodal recurrent neural networks. pages 681–688, 05 2017.

[5] Rnn, lstm and gru. http://dprogrammer.org/rnn-lstm-gru.