GOLDSMITHS, UNIVERSITY OF LONDON

FINAL YEAR PROJECT

Spoiler Alert: Deep Learning with Natural Language Processing to identify spoilers

---

Author:
Yasmin PAKSOY

Supervisor:
Dr. Daniel STAMATE

May 2022

Abstract

Social media has become an endless sea of information; some you want to see and others you don't. This project uses deep learning on a dataset of IMDb movie reviews to train neural network models to detect spoilers in text. A spoiler is information that will ruin a viewer's sense of surprise during a motion picture; this paper focuses on movie spoilers in particular, hence the use of the IMDb dataset. While other research into this area does exist, they mainly use book review data, which does not generalize well to movies. The three types of models developed for this task are presented: bag-of-words, recurrent neural networks, and transformers. Final models are evaluated using accuracy, recall, precision, and F1 scores. The best results are achieved by the transformer model, which uses DistilBERT. When compared to previous research, this model achieves a higher F1 score.

Keywords

Spoiler Detection, Text Classification, Deep Learning, LSTM, Transformers

Acknowledgements

I would like to thank my supervisor, Daniel Stamate, for his support and reassurance throughout this project, which was an immense help. I would like to take this opportunity to express my gratitude to my friends, Davids, thank you for making this final year so enjoyable. My appreciation also goes out to my family for their encouragement, support and unwavering belief in me while I completed my degree.

Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
Colab	Google Collaboratory
DL	Deep Learning
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
RNN	Recurrent Neural Network

Chapter 1

Introduction

1.1 Motivation

Spoilers, defined as information about the plot of a motion picture that can spoil a viewer's sense of surprise or suspense [1], have the potential to ruin any entertainment experience: Dumbledore dies, Darth Vader is Luke's father, Jon kills Daenerys.

In recent years, multiple psychological studies have concluded that spoilers cause a decrease in enjoyment [2] [3]. Following this, a poll done by the American media company Vox [4] has found that 50% of participants find it "very important" that a movie, book, or TV show is not spoiled for them. Only 15% of participants expressed that this was "not important" for them, while the remaining found it "somewhat important". Even though these statistics may suggest that society agrees that spoilers are unpleasant, they are still massively prevalent. Although 85% of people find avoiding spoilers at least somewhat important, they do not share the same concern for not spoiling movies or TV shows for other people. A study [5] found that participants can take pleasure in spoiling movies or TV shows for others. Given reasons include: "Just for fun", "It's nice to see your friends angry" and "Because it's good to see people suffering!".

With the rise of social media platforms, it has become almost impossible for those with a social media presence, 78% of the UK's population [6], to avoid spoilers, making them a large first-world cause for concern. The study done by Vox reported that 50% of participants found social media platforms, namely Facebook and Twitter, to be responsible for most spoilers they've seen. In a poll about the appropriate amount of time to wait before discussing major movie plot twists on social media [7], New York Magazine found that 22% of respondents thought it was okay to discuss immediately and only 33% of respondents they'd wait two weeks or longer. In recent years, users have taken to 'live-tweeting' while watching TV shows and movies also. This is where users tweet their opinions and reactions during the event itself, and do not even wait for the end before taking to social media [8].

Along with the rise of social media platforms, online streaming services have also now become common. In 2021, over sixteen thousand households in the UK were subscribed to Netflix [9]. This means that when Netflix drops a new movie, millions of people in the UK have immediate access to it without even having to go to the cinema or spend extra money. In regards to TV shows, Netflix drops a season at a time, rather than the conventional episode per week. This means that viewers are now watching TV shows at the time and speed they'd like, and so it is much more common for two people to start a season at the same time but finish days or weeks apart. This gives people access to spoilers on a much larger and faster scale than before.

These two aspects combined has made avoiding spoilers online almost impossible. Some platforms have put in processes to help users avoid spoilers, which will be discussed in the next section, but overall, there is not a reliable solution to this problem. That is the main motivation for this project.

1.2 Approach

The advances in artificial intelligence (AI) and natural language processing (NLP) in recent years can help eliminate this issue. This paper looks at how these advances can be used to identify spoilers in text, which can then be blocked from social media users.

In the past decade, the field of AI, "the effort to automate intellectual tasks normally performed by humans" [10], has been reinvigorated. This is due to the newfound access to high powered central and graphics processing units (CPU/GPU) [10] and wide internet access, which has meant that it is now much easier to collate large datasets that can be used to train machine learning (ML) models. Machine learning is an AI approach which uses data to generate rules, which can then be applied to new data to produce original answers. ML consists of multiple subsets, of which this paper will focus on deep learning (DL). DL puts emphasis on learning successive layers of increasingly meaningful representations, while other ML approaches only focuses on learning one or two layers of representations, thus shallow learning [10]. The successive layers in DL allow it to learn much more high level features from data, resulting in better models for NLP tasks. NLP is a branch of AI that focuses on a computer's ability to understand text and spoken words in the same way human beings can [11], meaning this project is a NLP binary classification task.

In the current state of this field, there has not been concrete development in the context of spoiler detection in regard to movies. While there has been similar research to that done in this project, it has either been executed using spoiler data for books or does not use state of the art methods for detection. Due to this, algorithms have not yet yielded high accuracy scores for movie spoilers.

1.3 Objectives

The objective of this project is to systematically train several types of DL models to find the best model possible for classifying text as either contains spoilers or not, aiming to develop a model that preforms better than the current industry best. I will build and discuss bag-of-words, recurrent neural network, and transformer models. While bag-of-words and recurrent neural network models have previously been developed in the context of avoiding spoilers, transformers models are a state of the art solution that have not yet been explored in this context. These models will be trained on a dataset of over 500,000 movie reviews from IMDb, each with a classifying label. This paper will compare results of these different models and discuss which is best suited for this task.

Chapter 2

Background

2.1 Existing Solutions

As previously mentioned, some platforms have put in processes to help users avoid spoilers, while others have systems in place for unrelated reasons that can be used to avoid spoilers. For example, some websites allow users to "tag" whether their post contains spoilers or not. If the post contains a spoiler, it will be made apparent to users. For example, the social media platform Reddit includes a small red tag next to the title of text posts and blurs images that have been marked as containing spoilers. While this solution seems like it would work perfectly, human-error can mean that some posts are not tagged correctly, or as discussed previously, users may knowingly not tag their posts in order to feel pleasure out of spoiling movies for others.

The social media platform Twitter has a mute function where users are able to add words or phrases that they do not want to see on their timeline. However, for each movie they'd like to avoid spoilers for, users have to manually add each word and phrase that may relate to it, as not every Tweet containing spoilers will necessarily contain the movies name. This is a long and tedious process, and it is impossible for users to hit every possibility if they have not yet seen the movie and only know a few key names and phrases, so they may see spoilers anyway. These are the only two popular social media platforms that have the ability to somewhat block spoilers. Other popular platforms, Facebook, Instagram, TikTok, have no built-in functionality to aid in blocking spoilers.

Other solutions, which are not platform specific, include browser extensions, such as Spoiler Slayer on Chrome and Firefox [12]. This uses the same concept as Twitter's mute function; users manually enter the words and phrases they want to block, making it again impossible for this to work all the time. This browser extension is not unique, and others exist exactly like it, however not one has a big user base. The most popular one is Spoiler Shield [13] on Chrome, with a little over one thousand users and only two out of five stars. Considering how many people use social media today, this is an extremely small number.

Due to these issues, most people have taken to fully avoiding social media on release days of new movies. When searching the web on how to avoid spoilers, the number one solution given by most posts is to simply avoid social media at all costs [14] until you have seen the movie, and some even suggest staying off your phone completely [15].

2.2 Similar Work

As mentioned previously, there has been some research pursuing possible solutions to this problem, but not in the context of movies. Regardless of this, I will be using this research as a starting point as many of the papers use similar methods and have the same data format. There are two main papers that I will be discussing as they are the most up-to-date and represent the current state of spoiler detection best.

2.2.1 SpoilerNet

SpoilerNet [16], developed by the computer science department at the University of California San Diego in 2019, was trained to identify spoilers at a sentence level from two datasets: book reviews from Good Reads and single-sentence comments about TV shows from TVTropes.org. Each sentence was labelled as either containing or not containing a spoiler. They used Gated Recurrent Networks (GRUs) to train the models and reported 74% accuracy for TV spoilers. The models were trained not only on the review data, but the metadata for each review as well. This includes book titles, descriptions, etc.

2.2.2 "Killing Me" Is Not a Spoiler

This paper [17], by Chang et al. at Korea University, proposed a spoiler detection model using Graph Neural Networks to better capture the semantics in text. This was suggested after the authors found that attention-based models, such as SpoilerNet, regard the phrase "killing me" as a spoiler, even though it is a metaphor. This paper also used the same datasets as SpoilerNet, Good Reads book reviews and TVTropes.org comments. The project used pretrained word embeddings and LSTM to encode contextualized word representations. They trained using a relation-aware attention mechanism and achieved an F1 score of 0.210 on the Good Reads dataset and 0.801 on the TVTropes.org dataset.

2.2.3 Review

I have decided to focus on these two papers in particular as the projects are very similar in nature; all three are attempting to classifying spoilers from review data, while other projects use data from various sources. While the existing research has achieved decent results in spoiler detection, the data used has not made them ideal for spoiler blocking on social media platforms.

Both SpoilerNet and Chang et al. classify on the sentence level. When discussing movies on social media, users rarely write a substantial piece of text where only a particular section contains a spoiler. While there are no statistics into how long an average post on social media is, many sources agree that posts with 50-150 characters receive the most engagement [18]. Due to this, it is not necessary to classify each sentence individually for movie spoilers on social media. Additionally, SpoilerNet being trained on the metadata means that the model had much more information about the book/TV show when making predictions. Since this will not be the case on social media, the models presented in this paper are only trained on review text.

2.2.4 Other Research

There have been other papers published addressing this subject using a range of different methods. For example, a paper named “Spoiler detection in TV program tweets” [19] in 2015 was also aimed at detecting spoilers in tweets. This project used a support vector machine (SVM) based prediction model and was then trained on tweets collected about the TV show Dancing with the Stars season 13 and the 2014 World-Cup final.

2.3 Research into the Field

In Deep Learning, representations are learned via artificial neural networks (ANNs), a computer system initially inspired by human brains [20], but now regarded as a mathematical framework [10]. ANNs are built using artificial neurons and are ideal for NLP tasks as they are powerful, versatile, and scalable, and thus can learn from large and complex data [21].

2.3.1 History

The first artificial neurons were suggested by Walter Pitts and Warren McCulloch in 1943 [21]. This neuron has one or more binary input, one binary output, and was activated when a certain number of its inputs were active. Building a network of these neurons made it possible to compute any logical proposition.

Following this, Frank Rosenblatt invented the Perceptron in 1957 [21]. This is one of the simplest ANN architectures; it uses numbers instead of the binary values suggested by Pitt and McCulloch and has an associated weight for each input. It uses a threshold logic unit or a linear threshold unit to calculate the weighted sum of inputs and applies a step function to that sum to calculate the output value. This architecture cannot solve complex problems, as highlighted by Minsky and Papert in their 1969 book Perceptrons [22].

To overcome the issues present in Perceptrons, multi-layer Perceptrons (MLPs) were invented. This consists of stacking Perceptrons together to include not only input and output layers, but several hidden layers of nodes also, called a deep neural network. MLPs can be trained using the backpropagation algorithm, popularized by Rumelhart, Hilton, and Williams [23]. While other methods have been developed in the meantime, deep neural networks have been the go-to for many tasks since 2012, including natural language processing.

2.3.2 How Deep Learning Works

A deep neural network consists of the following aspects:

- Layers, which are linked into a model
- The loss function, which calculates the loss value used for learning
- The optimizer, which uses the loss value to update the networks weights

Layers

Layers are made up of a number of artificial neurons, each of which have a corresponding weight. The input data given to the layer is parameterized by its weights. The more layers there are in a model, the more transformations the input data goes through. A balance needs to be found between having too many or too little, different for every problem. The last layer will make the label prediction for the input data.

Loss Function

The loss function is a measure of distance. It measures how far the prediction of the model is from the actual label. The closer the prediction to the actual value, the smaller the loss function will be. The goal of training is to minimise the loss function as much as possible. There are many different loss functions to choose from, but the decision should be made based on the type of task. In the context of spoilers, the default loss function is binary-crossentropy, as this is a binary classification task. This measure calculates the performance of the model by outputting a value between 0 and 1. A lower value suggests that the model is improving.

Optimizer

The optimizer receives the loss function as input and uses the backpropagation algorithm to decide how much the weights will be adjusted in order to decrease the loss function. As more examples are processed by the model, the optimizer tunes the weights to produce the lowest possible loss. There are many different optimizers available; some are known to produce better results, such as Adam [21], but the results depend heavily on the data so not one is known as the best.

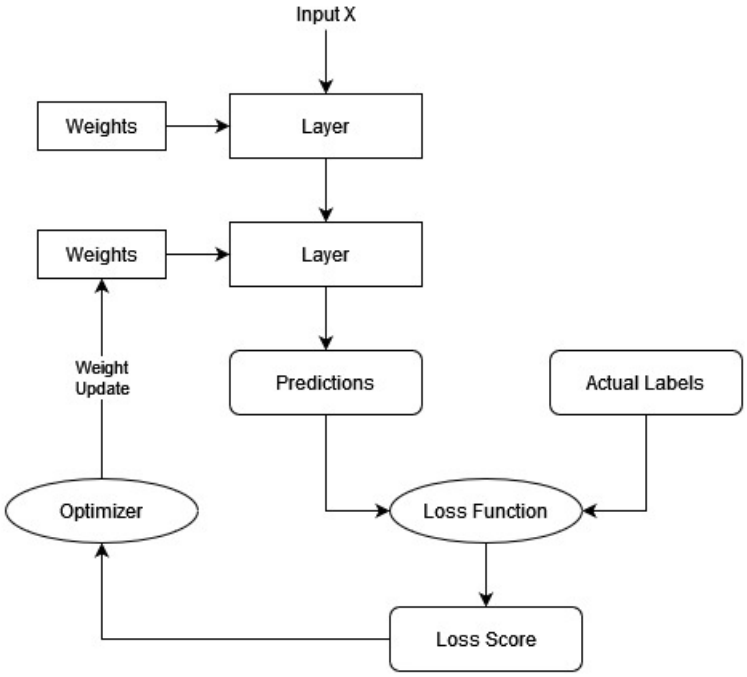


FIGURE 2.1: Process of training

2.3.3 Vectorizing for Natural Language Processing

Vectorization consists of transforming text into numeric representations [10]. For all DL models, the input data must be numeric, text cannot be used for training without vectorization. For text to be vectorized, it needs to be split into tokens. This process is known as tokenization; text is split either into words, characters, or n-grams. After tokenization, numeric vectors can then be associated with each token, which will be passed to a DL model. The two most common methods are one-hot encoding and word embedding.

In one-hot encoding, each word is represented by a unique integer i. A binary vector the size of the total vocabulary is made for each word, where all indexes are zero except for i, which is one. Where the size of the vocabulary is n, one-hot encoding is n-dimensional. For a vocabulary of 10,000 words, the tokens produced will 10,000-dimensional [10].

Word embedding are fewer dimensions and are learned from data. There are two ways to produce word embeddings. One method is learning them as you train the main task, i.e., classification, by starting with random word vectors which are adjusted the same way weights are. The other is to use pretrained word embeddings, which is where embedding vectors are loaded from an existing space that already captures the generic aspects of the language [10].

2.3.4 Feed-Forward Neural Networks

Classic DL networks are known as feed-forward networks; each input neuron is connected to each neuron in the first hidden layer, which is connected to each neuron in the second hidden layer and so forth [20]. The more hidden layers there are in a feed-forward network, the more complex features are extracted from the input. For such models, the input for NLP tasks consists of the whole text, as there is no memory allowing words to be processed separately. While this may cause problems for sequence data, it is beneficial to build these models anyway, as it will provide a baseline for comparison and justify the need for using more complex models [10].

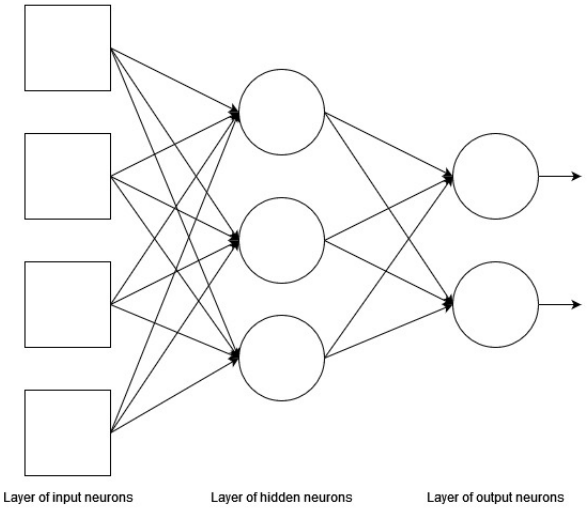


FIGURE 2.2: Feed-forward Network with 1 hidden layer

2.3.5 Recurrent Neural Networks

When processing text, humans read one word at a time; the previous words are remembered, creating context for the upcoming words. As mentioned, this is not possible in feed-forward networks, as the text is regarded as one whole, context and meaning can be difficult to discern by the model. To combat this issue, recurrent neural networks (RNNs) can be used. RNNs process sequences using an internal loop, so all sequence elements are processed separately, and the model can maintain a state representing the information processed so far [10]. There are two main RNN architectures used for text processing, long short-term memory (LSTM) and gated recurrent units (GRUs).

LSTM was introduced by Hochreiter and Schmidhuber in their 1997 paper with the same name [24]. While simple RNNs do remember previously processed elements, they only have a short-term memory, so incorrect predictions can still be made if the reverent data was processed a few too many sentences ago [25]. LSTM contains *cells* in hidden layers which have input, output, and forget gates that control the flow of information needed to predict the networks output.

GRUs are similar to LSTM in that they work to address the short-term memory problem faced by simple RNNs, but are only made up of two gates, rest, and update, which controls how much and which information to retain. They tend to be less computationally expensive and streamlined when compared to LSTM [26].

2.3.6 Transformers

Transformers were introduced by Vaswani et al. in the 2017 paper “Attention is all you need" [27], where they outlined that "neural attention" could be used to build models for sequence data that did not need to use any recurrent layers. This has revolutionised the NLP field; neural attention is now the state-of-the-art method for NLP tasks and beyond.

A different vector representation is needed for words when they are used in different contexts, e.g., to represent the difference between going on a ’date’ and buying a ’date’ at a shop [28]. Transformers use self-attention to find the correlation between the words in a sequence, to then compute a representation that indicates the syntactic and contextual meaning of the word.

Since the initial introduction, there have been many transformer-based techniques that have emerged. Most notably, Devlin et al. from Google released the BERT model [29], which then went on to be utilised in Google’s search engine. BERT consists of 12 encoders, to read input, and 12 attention heads, that are pretrained on 3,300 million words from various databases. A smaller and faster

variation of this model was then suggested in 2019 by Sanh et al. in the paper "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter" [30], which has gained popularity.

2.4 Required Tools

In order to implement the methods discussed, there are a range of technologies that will be necessary.

2.4.1 Programming Language

Python [31] is a high-level programming language that has become industry standard for AI implementations. Due to this, there have been many libraries developed to make AI implementations more streamlined, making Python the best choice for this task.

2.4.2 Libraries

There are 3 main libraries that are used in this project:

TensorFlow [32] is an open-source platform for AI, released by Google. It focuses on building and training deep neural networks and includes a range of features such as various optimizers.

Keras [33] is an open-source DL library used with TensorFlow. It provides implementations for the most common neural network components, such as different types of layers, activation functions and loss functions.

Hugging Face [34] is an AI and data science platform that provides multiple libraries to aid the building and training of transformer models. Relevant to this project, it includes a Datasets library that allows easy dataset building and formatting, a Tokenizers library that includes the most common tokenizers used in transformer models and a Transformers library that provides APIs to download and train pretrained transformer models.

Other libraries used include:

- **Scikit-Learn** [35], a Python library for ML.
- **Pandas** [36], a open-source library for data analysis and manipulation in Python.
- **Matplotlib** [37], a plotting library for data visualisation in Python.
- **NumPy** [38], a library supporting large, multi-dimensional arrays and matrices.

2.4.3 Environment

Current industry standard for training DL models is using Jupyter Notebooks, a web-based environment consisting of input/output cells containing code or text. Local Jupyter Notebooks face hardware limitations as they utilise the RAM and GPU of the computer. This project will be building computationally expensive models, so Jupyter Notebooks will be utilised on Google Collaboratory instead, a cloud-based Jupyter Notebook environment built to work with TensorFlow.

Chapter 3

Methods

3.1 Dataset

Most recent research has used the Good Reads dataset of book reviews or the TVTropes.org dataset of TV show comments. While these databases have the large amount of data required to train DL models, they are not specific enough to movie spoilers. As mentioned before, the Good Reads dataset in particular contains data samples much longer than social media platforms would have. Due to this, I will be using the IMDb Spoiler dataset [39], collated by Rishabh Misra, instead.

This dataset was collected from IMDb, an abbreviation of Internet Movie Database, a platform providing information about movies and TV shows where users can also share their reviews. While TV shows are also present on this platform, this dataset only contains movie information. The dataset consists of two files; movies, which contains metadata about movies, and reviews, which contains movie reviews along with metadata for each.

IMDb allows users to know which reviews contain spoilers by tagging them with "Warning: Spoilers". Users are able to indicate whether their review includes a spoiler before they post. Reviews which include spoilers but have not been indicated to so will be removed when reported under the "Spoilers without warning" option, allowing for confidence in the accuracy of the data. While this method does rely on crowd-sourcing, which can usually be unreliable, IMDb being the biggest movie review platform available suggests that this data will be reliable.

3.1.1 Movies File

The movies file contains 1572 records, all of which have reviews in the reviews file. There are 7 attributes in this dataset: movie ID, plot summary, genre, rating, release date and plot synopsis.

	movie_id	plot_summary	duration	genre	rating	release_date	plot_synopsis
0	tt0105112	Former CIA analyst, Jack Ryan is in England wi...	1h 57min	[Action, Thriller]	6.9	1992-06-05	Jack Ryan (Ford) is on a "working vacation" in...
1	tt1204975	Billy (Michael Douglas), Paddy (Robert De Niro...	1h 45min	[Comedy]	6.6	2013-11-01	Four boys around the age of 10 are friends in ...
2	tt0243655	The setting is Camp Firewood, the year 1981. I...	1h 37min	[Comedy, Romance]	6.7	2002-04-11	
3	tt0040897	Fred C. Dobbs and Bob Curtin, both down on the...	2h 6min	[Adventure, Drama, Western]	8.3	1948-01-24	Fred Dobbs (Humphrey Bogart) and Bob Curtin (T...
4	tt0126886	Tracy Flick is running unopposed for this year...	1h 43min	[Comedy, Drama, Romance]	7.3	1999-05-07	Jim McAllister (Matthew Broderick) is a much-a...
5	tt0286716	Bruce Banner, a brilliant scientist with a clo...	2h 18min	[Action, Sci-Fi]	5.7	2003-06-20	Bruce Banner (Eric Bana) is a research scienti...
6	tt0090605	57 years after Ellen Ripley had a close encoun...	2h 17min	[Action, Adventure, Sci-Fi]	8.4	1986-07-18	After the opening credits, we see a spacecraft...
7	tt0243155	Bridget Jones is an average woman struggling a...	1h 37min	[Comedy, Drama, Romance]	6.7	2001-04-13	Bridget Jones (adorably played by Renee Zellwe...
8	tt0121765	Ten years after the invasion of Naboo, the Gal...	2h 22min	[Action, Adventure, Fantasy]	6.6	2002-05-16	The opening crawl reveals that the Galactic Re...
9	tt0443453	Borat Sagdiyev is a TV reporter of a popular s...	1h 24min	[Comedy]	7.3	2006-11-03	Borat Sagdiyev is a TV reporter of a popular s...

FIGURE 3.1: First 10 records of movies file

3.1.2 Reviews File

The reviews file contains 573,913 records, all reviews for the movies in the movies file. There are 7 attributes in this dataset also: review date, movie ID, user ID, spoiler tag, review text, rating and review summary.

	review_date	movie_id	user_id	is_spoiler	review_text	rating	review_summary
0	10 February 2006	tt0111161	ur1898687	True	In its Oscar year, Shawshank Redemption (writt...	10	A classic piece of unforgettable film-making.
1	6 September 2000	tt0111161	ur0842118	True	The Shawshank Redemption is without a doubt on...	10	Simply amazing. The best film of the 90's.
2	3 August 2001	tt0111161	ur1285640	True	I believe that this film is the best story eve...	8	The best story ever told on film
3	1 September 2002	tt0111161	ur1003471	True	**Yes, there are SPOILERS here**This film has ...	10	Busy dying or busy living?
4	20 May 2004	tt0111161	ur0226855	True	At the heart of this extraordinary movie is a ...	8	Great story, wondrously told and acted
5	12 August 2004	tt0111161	ur1532177	True	In recent years the IMDB top 250 movies has ha...	8	Good , But It Is Overrated By Some
6	9 October 2005	tt0111161	ur6574726	True	I have been a fan of this movie for a long tim...	9	This Movie Saved My Life.
7	4 February 2012	tt0111161	ur31182745	True	I made my account on IMDb Just to Rate this mo...	10	Movie you can see 1000 times
8	24 October 2008	tt0111161	ur9871443	True	A friend of mine listed "The Shawshank Redempt...	10	The Shawshank Redemption
9	30 July 2011	tt0111161	ur2707735	True	Well I guess I'm a little late to the party as...	10	"I'm a convicted murderer who provides sound f...

FIGURE 3.2: First 10 records of reviews file

This project will only make use of the review text and spoiler tag. As the models are attempting to classify if text includes spoilers or not, they will be trained on the review text to predict the spoiler tag, which can then be checked against the actual labels to calculate metrics. While other information could improve the results of any models built, such as the movie plot summary and synopsis, they would not be present on social media and would thus produce a model that is not suitable for the real world applications of this project.

Of the 573,913 total reviews, only 150,924 have a spoiler tag of true, i.e., contain spoilers, giving this dataset an approximate ratio of 4 to 1. This is highly imbalanced, which was an initial cause for concern. Imbalanced datasets result in models showing a bias towards improving performance on the majority class, which is not the class of interest [40]. This means a model may not be able to detect instances of the positive class, spoilers. There are two methods used to overcome this problem; balancing the dataset and evaluating with metrics other than accuracy, both which have been used in this project.

3.2 Evaluation

3.2.1 Metrics

In order to evaluate how well a model is performing, metrics need to be used. For binary classification tasks, the traditional evaluation metric is accuracy. This does not however provide an accurate representation of a model's performance with imbalanced datasets. For a class split ratio of 4 to 1, the model can achieve 80% accuracy by just predicting false for every data sample. While 80% would be a decent score for most NLP tasks, it does not portray any success for this problem as it has not necessarily detected any of the positive class, spoilers. Due to this, accuracy will only be used as an evaluation metric when the dataset has been manually balanced, and in other cases, recall, precision and F1 scores will be used.

$$\text{Accuracy (a)} = \frac{\textit{True Positives} + \textit{True Negatives}}{\textit{True Positives} + \textit{True Negatives} + \textit{False Positives} + \textit{False Negatives}}$$

FIGURE 3.3: Accuracy Formula

Recall

Recall represents the proportion of predicted positive classes that are actually positive. It can be defined as the fraction of the correctly predicted positive records over all correct predictions. A high recall score means the model is better at detecting the spoiler class.

$$\text{Recall (r)} = \frac{\textit{True Positives}}{\textit{True Positives} + \textit{False Negatives}}$$

FIGURE 3.4: Recall Formula

Precision

Precision represents the proportion of correct positive class predictions out of all of them. It can be defined as the fraction of correct predictions of the positive class over the total number of positive predictions. When precision is low, it means that the model is predicting many records as positive (spoilers) when they are not.

$$\text{Precision (p)} = \frac{\textit{True Positives}}{\textit{True Positives} + \textit{False Positives}}$$

FIGURE 3.5: Precision Formula

F1 Score

F1 is a harmonic mean of both precision and recall. For recall to be high, precision must be low and vice versa, so it can become difficult to evaluate models on both, however both are still reverent as they measure different types of errors. F1 represents both precision and recall in a way that is easier to evaluate.

$$\text{F1 Score (f)} = 2 \times \frac{(\textit{precision} \times \textit{recall})}{(\textit{precision} + \textit{recall}}$$

FIGURE 3.6: F1 Score Formula

3.2.2 Evaluation Protocol

The evaluation protocol determines how data will be split for training, validation and testing. The most important aspects when evaluating DL models is to ensure the validation or testing data is unseen: the model has not used it for training. This represents how generalisable the model is, which is the ultimate goal.

The choice of evaluation protocol heavily depends on the dataset. When using a dataset with a very limited number of samples, K-Fold validation is a good choice as it doesn't require further splitting the training set into partial training and validation. While the IMDb Spoilers dataset is imbalanced, it does not have a limited number of records; even after manually balancing the dataset

to a 1 to 1 ration, there would be over 300,000 data samples. Due to this, K-Fold validation is not necessary.

Instead, the evaluation protocol used is hold-out validation; the dataset gets split into partial train, validation and test sets. This allows for an error rate to be computed on the validation data that can be used when attempting to find the best configuration of hyper-parameters, while still testing on unseen data, which is the most important aspect.

3.3 Optimisation & Activation

3.3.1 Loss

As previously mentioned, a loss function is necessary to measure the distance between the prediction of a deep learning model and the actual label. All models developed use binary crossentropy as the loss function, the most common for binary classification tasks.

$$\text{Binary Crossentropy} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \times \log(y_{pred}) + (1 - y_i) \times \log(1 - y_{pred})]$$

FIGURE 3.7: Binary Crossentropy Formula

3.3.2 Optimizer

The choice of optimizer depends on the task. The job of the optimizer is to optimize, i.e., decrease, the loss function. There are a range of optimizers used in this project, but each one is an adaptive learning rate method, meaning the learning rate does not remain constant during training. This ensures large weight updates will only take place when the output of the model varies significantly, decreasing the chances of models not progressing past a local optimum.

RMSProp

RMSProp, standing for Root Mean Square Propagation, is used for most models in this paper. This optimizer maintains a moving average of the square of gradients and divides the gradients by the root of this average [41].

Adam

Adam, based on adaptive estimation of first-order and second-order moments, is a stochastic gradient descent method [42]. Adam was developed by Kingma et al., who described it as:

> "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters"

NAdam

NAdam, developed by Dozat in 2015 [43], is a variation of Adam that uses Nesterov momentum instead of plain momentum [44].

3.3.3 Activation

Activation functions are used in order to gain access to a larger hypothesis space (the predefined set of operations used to make predictions). The larger the hypothesis space is, the more representations a model can learn. Without an activation function, a layer could only learn linear transformations of the data, so the multiple layers of representations would not add to the model. There are multiple difference activation functions used in this project; the choices were made by considering the model and layer type.

3.4 Bag-of-Words Models

Bag-of-words refers to tokenized text being passed to a model as a set rather than a sequence, meaning the words have no particular order. These models were built using Keras Dense layers and all other elements were also implemented using Keras.

3.4.1 Data Pre-Processing

Reviews were one-hot encoded in order to be accepted as input by the models using the Keras module `Tokenizer`. During this process, data was cut down by only considering the first 500 words of each review and the top 15,000 words in the dataset. All available data was used.

3.4.2 Activation

Hidden Layers

Rectified Liner Unit activation (ReLU) was the activation function of choice for the hidden layers. ReLU maps everything less than 0 to 0 and everything more than 0 to itself. This is linear behaviour (for any number larger than 0), which simplifies the learning of a model, making it the preferred choice for most ANN models, as described in chapter 6 of Introduction to Data Mining [40].

$$\text{ReLU}(x) = max(0, x)$$

FIGURE 3.8: ReLU Activation Formula

Output Layer

Sigmoid is a logistic function, meaning the output will always be between 0 and 1. Due to this, it is the activation function of the last layer for every bag-of-words model, as the number it returns can be considered as the probability of the review containing a spoiler or not.

3.4.3 Hyper-parameter tuning

After producing an initial model that had statistical power, different configurations were tested by altering the following hyper-parameters. Once the optimal value was found for one hyper-parameter, it was kept constant in all following models.

Learning Rate

The learning rate of an optimizer determines the rate at which the model updates its weights to find the minimum loss. The slower the value, the slower the model will find the minimum, however if the learning rate is too large the minimum could be skipped over altogether. The default learning rate for such models is 0.001. Learning rates larger (0.005, 0.003) and smaller (0.0005) were tried to find the optimum.

Weight Regularization

Weight regularization is where the network is forced to set weights to smaller values, making the distribution of them more regular. This is done by adding a cost to the loss function for larger weights. There are two types of cost:

1. **L1 Regularization**: Cost added is equivalent to the absolute value of the weight coefficients.
2. **L2 Regularization**: Cost added is equivalent to the square of the weight coefficients.

They can be added to a model individually or simultaneously using the `Regularizers` module from Keras.

Dropout

Adding dropout drops, i.e., sets to zero, a number of output features of a module during training. This is aimed to break coincidental patterns that are not significant, allowing the final model to be more generalisable. Dropout layers can be added to Keras models using the `Layers` module.

All Models

Below is a table showing all configurations tested of the bag-of-words model:

Model Number	Learning Rate	Weight Reg	Dropout
1	0.001	N/A	N/A
2	0.005	N/A	N/A
3	0.003	N/A	N/A
4	0.0005	N/A	N/A
5	0.001	L1	N/A
6	0.001	L2	N/A
7	0.001	L1 & L2	N/A
8	0.001	N/A	0.3
9	0.001	N/A	0.2
10	0.001	N/A	0.1

FIGURE 3.9: All Bag-of-Words configurations

3.5 LSTM & GRU Models

Both LSTM and GRU models were implemented using Keras.

3.5.1 Data Pre-Processing

Data was one-hot encoded in the same way it was for the bag-of-words models, however the partial training data was then balanced to a 1:1 ratio. After splitting and balancing, the partial training set contained 249,126 records, half of which were spoilers.

3.5.2 Activation

RNN Layers

RNN models have two types of activation functions:

- **Output Activation**, the general activation function. Default is Hyperbolic tangent (Tanh), which has not been changed for these models. Tanh returns values ranging from -1 to 1; strong positive inputs are mapped closer to 1 and strong negative closer to -1, making it a good choice for binary classification [20].
- **Recurrent Activation**, the activation function used during recurrence. Default is Sigmoid, which has not been changed for these models.

Output Layer

As with the bag-of-words models, Sigmoid was used as the activation function for the output layers on all models in this set.

3.5.3 Hyper-parameter tuning

After initial LSTM and GRU models were built, the best performing one was tuned. This consisted of trying different configurations by altering the following hyper-parameters:

Dropout

Dropout was added also. When adding dropout to RNNs, two types have to be added [45]:

- **Input Dropout**, the fraction of the units to drop of the inputs.
- **Recurrent Dropout**, the fraction of the units to drop of the recurrent state.

Dropout rates of 20%, 40% and 30% were tested.

Learning Rate & Optimizer

As it was for the bag-of-words models, the learning rate was tuned. The default for RNNs in Keras is 0.001, four further learning rates were tested (0.003, 0.006, 0.004, 0.01). Different optimizers were also tested, in particular Adam and NAdam.

All Models

Below is a table showing all configurations tested of the LSTM and GRU models:

Model Number	Layer Type	Learning Rate	Recurrent/Dropout	Optimizer
1	LSTM	0.001	N/A	RMSProp
2	GRU	0.001	N/A	RMSProp
3	LSTM	0.001	0.2	RMSProp
4	LSTM	0.001	0.4	RMSProp
5	LSTM	0.001	0.3	RMSProp
6	LSTM	0.003	0.4	RMSProp
7	LSTM	0.006	0.4	RMSProp
8	LSTM	0.004	0.4	RMSProp
9	LSTM	0.01	0.4	RMSProp
10	LSTM	N/A	0.4	Adam
11	LSTM	N/A	0.4	Nadam

FIGURE 3.10: All LSTM/GRU configurations

3.6 DistilBERT Models

DistilBERT models were implemented using the Transformers, Tokenizer and Datasets library from Hugging Face. Note: Transformer models do not use activation functions.

3.6.1 Data Pre-Processing

Before processing, data was split into partial training, validation and testing sets. Due to the computational power required to train Transformer models, using all the data was not feasible. Due to this, the set sizes were decreased:

- Partial Train: 554 records, balanced.
- Validation: 250 records.
- Test: 250 records.

Text data was then tokenized using the DistilBERT pre-trained model vocabulary.

3.6.2 Hyper-parameter tuning

After producing an initial model that had statistical power, different configurations were tested by altering the following hyper-parameters:

Learning Rate

Transformer models have learning rates much smaller than others; the default is 2e-05. Learning rates 2e-06 and 2e-04 were also tested.

Weight Decay

Weight decay is a regularization technique that adds a small penalty to the loss function, to aid in avoiding overfitting. The smaller the penalty, the longer it will take for the model to fit the data. If the weight decay is too large, the model will not fit the data well [46].

$$\text{Loss} = \text{loss} + \text{weight decay rate} \times \text{sum of weights}^2$$

FIGURE 3.11: Loss calculation with weight decay

Default models in Hugging Face do not include weight decay but the industry standard is 0.01, which was used. Rates 0.02 and 0.04 were also tested.

Dropout

Transformer models also have two types of dropout:

- **Dropout**, for all layers in the embeddings and encoder.
- **Attention Dropout**, for all attention probabilities.

Dropout defaults to 10% for both. Dropout rates of 20% and 30% were also tested.

Number of Hidden Layers

As previously mentioned, more hidden layers are able to extract more complex features input. The number of hidden layers defaults to 8 in the DistilBERT model, 8 and 10 layers were also tested.

All Models

Below is a table showing all configurations tested of the DistilBERT model:

Model Number	Learning Rate	Weight Decay	Dropout	Number of Layers
1	2.00E-05	0.01	0.1	6
2	2.00E-06	0.01	0.1	6
3	2.00E-04	0.01	0.1	6
4	2.00E-04	0.04	0.1	6
5	2.00E-04	0.02	0.1	6
6	2.00E-04	0.02	0.2	6
7	2.00E-04	0.02	0.3	6
8	2.00E-04	0.02	0.2	10
9	2.00E-04	0.02	0.2	8

FIGURE 3.12: All DistilBERT configurations

Chapter 4

Results

4.1 Validation Results

Each model is trained for a certain number of epochs: the number of iterations over the training data. The optimal number of epochs is the one with the lowest validation loss, as explained previously, before the model starts overfitting. All results given are for the optimal number of epochs.

4.1.1 Bag-of-Words

Figure 4.1 shows the configuration and results for every bag-of-words model developed. A total of 10 models were developed. The results consist of the accuracy and F1 score for this set as the training data was not balanced, making accuracy alone an unreliable metric.

Model Number	Learning Rate	Weight Reg	Dropout	Optimim Epochs	Val Loss	Val Accuracy	Val F1
1	0.001	N/A	N/A	1	0.4994	0.7647	0.3825
2	0.005	N/A	N/A	1	0.5065	0.7555	0.3966
3	0.003	N/A	N/A	2	0.5046	0.7617	0.3455
4	0.0005	N/A	N/A	2	0.5019	0.7617	0.2560
5	0.001	L1	N/A	4	0.7946	0.7615	0.3175
6	0.001	L2	N/A	3	0.5114	0.7624	0.3098
7	0.001	L1 & L2	N/A	4	0.7969	0.7615	0.3508
8	0.001	N/A	0.3	3	0.5040	0.7677	0.3548
9	0.001	N/A	0.2	3	0.5062	0.7662	0.3996
10	0.001	N/A	0.1	2	0.4992	0.7658	0.4158

FIGURE 4.1: Bag-of-Words Results

As the table shows, hyper-parameters were tuned in segments after the development of the initial model. First, learning rate was tuned, developing models 2, 3 and 4. Of the first 4 models, model 1 had the lowest validation loss, so 0.001 was fixed as the learning rate in the later models. Next, weight regularization was added, and all three models (numbers 5, 6 and 7) achieved an optimum validation loss high than model 1, so weight regularization was removed. Lastly, dropout was added, resulting in models 8, 9 and 10. Model 10 achieved the lowest validation loss, which was also lower than the previous best, model 1, and thus became the best of this set.

The configuration of this model and its results on the validation set can be seen below.

Model Number	Learning Rate	Weight Reg	Dropout	Optimim Epc	Val Loss	Val Accurac	Val F1
10	0.001	N/A	0.1	2	0.4992	0.7658	0.4158

FIGURE 4.2: Bag-of-Words Best Model Configuration & Results

4.1.2 LSTM/GRU

Figure 4.3 shows the configurations and results for every LSTM/GRU model developed. A total of 11 models were developed, and the results also consist of the accuracy and F1 score. While these models

were trained using a balanced dataset, keeping the F1 score will allow for more accurate comparison to bag-of-words models.

Model Number	Layer Type	Learning Rate	Recurrent/Dropout	Optimizer	Optimum Epochs	Val Loss	Val Accuracy	Val F1
1	LSTM	0.001	N/A	RMSProp	7	0.5309	0.7176	0.5524
2	GRU	0.001	N/A	RMSProp	7	0.5340	0.7367	0.5497
3	LSTM	0.001	0.2	RMSProp	1	0.5147	0.7549	0.5096
4	LSTM	0.001	0.4	RMSProp	12	0.4960	0.7621	0.5359
5	LSTM	0.001	0.3	RMSProp	5	0.5263	0.7334	0.5461
6	LSTM	0.003	0.4	RMSProp	6	0.5134	0.7558	0.5329
7	LSTM	0.006	0.4	RMSProp	5	0.5219	0.7342	0.5531
8	LSTM	0.004	0.4	RMSProp	2	0.5098	0.7461	0.5302
9	LSTM	0.01	0.4	RMSProp	2	0.5112	0.7426	0.5405
10	LSTM	N/A	0.4	Adam	12	0.5112	0.7551	0.5405
11	LSTM	N/A	0.4	Nadam	10	0.5394	0.7298	0.5466

FIGURE 4.3: LSTM/GRU Results

The first two models were developed to test which type of RNN would perform best with this set, the validation loss showing that LSTM achieved slightly better results. Other hyper-parameters were tuned using LSTM layers due to this. First, input and recurrent dropout rates of 0.2, 0.3 and 0.4 were tested (models 3, 4 and 5), which found a rate of 0.4 performed the best. Next, different learning rates were tested (models 6, 7, 8 and 9). As none of these models performed better than model 4, different optimizers were tested with their default learning rates. Neither Adam nor NAdam achieved a lower validation loss than model 4, thus model 4 became the best of this set.

The configuration of this model and its results can be seen below.

Model Number	Layer Type	Learning Rate	Recurrent/Dropout	Optimizer	Optimum Epochs	Val Loss	Val Accuracy	Val F1
4	LSTM	0.001	0.4	RMSProp	12	0.4960	0.7621	0.5359

FIGURE 4.4: LSTM Best Model Configuration & Results

4.1.3 DistilBERT

Figure 4.5 shows the configurations and results for every DistilBERT model developed. The results include precision and recall; as these models were trained on such a small set, other metrics would be necessary to accurately evaluate them. A total of 9 models were developed.

Model Number	Learning Rate	Weight Decay	Dropout	No. of Layers	Optimum Epoch	Val Loss	Val Accuracy	Val Recall	Val Precision	Val F1
1	2.00E-05	0.01	0.1	6	6	0.5516	0.7240	0.6032	0.4634	0.5241
2	2.00E-06	0.01	0.1	6	10	0.6912	0.4960	0.8696	0.3390	0.4878
3	2.00E-04	0.01	0.1	6	4	0.5368	0.6440	0.7826	0.4219	0.5482
4	2.00E-04	0.04	0.1	6	3	0.5359	0.7560	0.4366	0.5962	0.5041
5	2.00E-04	0.02	0.1	6	4	0.5350	0.7480	0.2857	0.5000	0.3636
6	2.00E-04	0.02	0.2	6	3	0.5330	0.7280	0.5238	0.4648	0.4925
7	2.00E-04	0.02	0.3	6	2	0.6745	0.7200	0.0282	0.6667	0.0541
8	2.00E-04	0.02	0.2	10	4	0.5793	0.6760	0.5070	0.4390	0.4706
9	2.00E-04	0.02	0.2	8	3	0.5364	0.6960	0.5077	0.4286	0.4648

FIGURE 4.5: Di8stilBERT Results

After developing the first model, different learning rates were tested (models 2 and 3). As model 3 had the lowest validation loss, the learning rate was permanently set at 2e-4. Next, the weight decay was altered (models 4 and 5), of which the rate 0.2 performed the best, thus was added to all future models. Dropout was tested next at rates 0.2 and 0.3 (models 6 and 7). Both kinds of dropout were altered during this process and model 6 performed the best, so a dropout rate of 0.2 was made permanent. Lastly, the number of layers was altered from the default 6 to 10 and 8, neither of which improved the mode, making model 6 the final one of this set.

The configuration of this model and its results can be seen below.

Model Number	Learning Rate	Weight Decay	Dropout	No. of Layers	Optimum Epoch	Val Loss	Val Accuracy	Val Recall	Val Precision	Val F1
6	2.00E-04	0.02	0.2	6	3	0.5330	0.7280	0.5238	0.4648	0.4925

FIGURE 4.6: DistilBERT Best Model Configuration & Results

4.2 Comparative Analysis

Figure 4.7 shows the validation accuracy and F1 score achieved by the best model in each set.

Model Set	Model Number	Optimim Epochs	Val Loss	Val Accuracy	Val F1
Set 1: Bag of Words	10	2	0.4992	0.7658	0.4158
Set 2: LSTM/GRU	4	12	0.4960	0.7621	0.5359
Set 3: Transformers	6	3	0.5330	0.7280	0.4925

FIGURE 4.7: Best Validation Results Comparison

These results make it clear that model 4 from set 2, LSTM/GRU, is the choice model for this task. Not only has it reached the highest accuracy score, but also the highest F1 score, showing that it's not only good at identifying the negative class, but the positive as well.

4.3 Final Results

In order to accept any one model as the best for this task, they all need to be tested on unseen data, i.e., the test set. The table below shows the results of this process. Recall and precision have also been included in the metrics for this process, to make in-depth analysis possible.

Model Set	Model Number	Accuracy	Recall	Precision	F1
Set 1: Bag of Words	10	0.76	0.55	0.43	0.48
Set 2: LSTM/GRU	4	0.78	0.27	0.74	0.40
Set 3: Transformers	6	0.56	0.74	0.34	0.47

FIGURE 4.8: Testing Results

The bag-of-words model's results are very similar to those achieved on its validation set, showing that the model is generalizable; it has performed as expected on unseen data, in regards to the accuracy and F1 score. While the accuracy is high, the recall shows that this model does not perform well in regards to identifying the positive class; only 55% of positive records are identified as such. This means that 45% of positive records are incorrectly identified as negative. The precision score shows that only 43% of positive predictions are correct, meaning 57% of records are incorrectly identified as positive. These scores are too high for this model to be utilised in real world settings.

The LSTM/GRU model achieves similar results to those achieved on the validation set also. The accuracy is quite high for such a task at 78%, however the recall score indicates issues. Only 27% of positive records are identified as such, which is too low to call this model a success. This means that 73% of the positive class is being identified as negative by the model, making this model unsuitable for real-world applications.

The transformer DistilBERT model achieves the lowest accuracy at 56% but can still be regarded as the best model. The recall score is 74%, meaning only 26% of positive class records are incorrectly identified. The precision score, 34%, does suggest that many negative classes are being predicted as positive, which explains why the accuracy is so low for a model with such a high recall. While the precision score does mean there will be many false alarms in regards to text being identified as spoilers, this model is still the most applicable to social media platforms, as it correctly identifies a majority of spoilers.

Chapter 5

Discussion

The initial objective of this project was to systematically train different types of DL models to find the best possible for classifying text as either contains spoilers or not. This objective has been met; 30 models were trained and evaluated systemically to find the optimal configuration of hyper-parameters for three types of models. In order to prevent overfitting, only a few hyper-parameters were trained in each set. This is where models are over-optimized on the training data, causing them to perform poorly on unseen data in comparison [10].

The results show the more complex models are necessary for classification to be possible of this data; while the accuracy alone may suggest bag-of-words and LSTM models performed the best, the recall and precision show this to be untrue. Using the transformer model was crucial to achieve decent results during this project. This is consistent with previous work; the most successful models for spoiler detection have been RNNs over NNs. Transformers are missing from this comparison due to the fact they are state-of-the-art and had not been implemented for this task until now. The general industry opinion is that transformers out-perform RNNs however, which this project has proved to be true.

This project aimed to develop a model that preforms better than the current industry best, which was presented in the paper "Killing Me" Is Not a Spoiler by Chang et al., discussed in chapter 2. This paper used both the Good Reads and TVTropes.org dataset. Due to the nature of the datasets, results will be compared to those achieved on the Good Reads dataset, as it is the most similar to the IMDb dataset; they both consist of review data, which is a format much different than the TVTropes.org comments. This paper reported an F1 score of 0.210 on the Good Reads dataset. The transformer model developed here, the best out of the final three, achieved an F1 score of 0.47 on the test data. This is much higher, suggesting that self-attention is a pivotal advancement for this task.

The second paper discussed in chapter 2 was SpoilerNet, who did not report the accuracy or F1 score on the Good Reads dataset. Accuracy was only reported on the TVTropes.org dataset, which was 74%. This is in fact much higher than the one achieved by the transformer model, 56%. One explanation for this could be the nature of the dataset; classifying one line of text should be much easier than multiple paragraphs, which explains why the accuracy achieved was significantly higher. Another explanation could be the size difference in training data. SpoilerNet was trained on all available data, while the transformer models here were trained on 554 records only, due to the lack of power. The other two models developed, bag-of-words and LSTM, did achieve an accuracy higher than 74%, 76% and 78% respectively. This is accredited to the fact they were trained on all available data, and not a minuscule amount like the transformer models. It can be expected that transformer accuracy would improve with training on more data.

Overall, the results show that it is possible to develop models to identify movies spoilers, which can be applied to social media platforms in different ways, e.g., browser extensions or built-in systems. This would block users from seeing a majority of spoilers, over and above any system available today.

Chapter 6

Limitations & Future Work

6.1 Limitations

6.1.1 Generalising to Social Media Platforms

These models were developed to be eventually applied to social media platforms, where the posts with most engagement tend to be 50-150 characters. This is much less than the average review on any review platform. Due to this, it may be difficult to apply this model to social media. While the results show the models are generalizable, the differences in the text length could cause the model to perform poorly. This is something that cannot be tested until integrations are built; there is no available dataset of spoilers from social media and building one would require a lot of manpower. Social media platforms would need to be scarped for posts, which would then need to be manually labelled as spoiler or not. While this is technically possible, it would not be feasible considering the time limitations of this project.

6.1.2 Limits of Data Pre-processing

For both the bag-of-words and LSTM models, during the one-hot encoding process, the reviews were cut down to the first 500 words only. Trimming the reviews this way could have meant the spoiler section of a few reviews were not actually included in the training data, causing the model to learn spoiler patterns incorrectly. Additionally, only the most common 15,000 words of the dataset were considered; while this would not cause issues in general NLP tasks, movies include many specific terms, e.g., character and location names, which could have been cut out due to this. This would have an effect on if a review is classed as a spoiler or not. While these two issues may seem insignificant alone, combined they may have a bigger effect than imagined. To circumvent these issues, the limits could be removed entirely, however this would increase the time and computational power needed to train these models, which is why the limits were put in in the first place.

6.1.3 Computational Power

The biggest limitation faced during this project was the lack of computational power. This problem was first faced when training a LSTM model with two layers. After training for approximately 5 hours, the model failed due to a Colab time-out, even though the most powerful Colab subscription was being used. Due to this, all further models only used one LSTM layer. Initial transformer models were cited sixty hours per epoch by Colab. This would mean a single model of 5 epochs would take three hundred hours to train, assuming that Colab did not time-out in the meantime. Running the model locally would have ensured there was not a time-out issue, but the model would take longer to train, which would not be feasible either. To bypass this issue, the training, validation, and testing sets were severely cut down as explained in chapter 3. This did allow the initial model and further hyper-parameter tuning models to be trained, but training with more data would have resulted in a better preforming model.

6.1.4 User Generated Labels

The label for each review has been decided by the sharer; as previously discussed, some people do enjoy sharing spoilers for malicious reasons, and so it is not possible to trust that all reviews have been labelled correctly. While the crowd-sourcing aspect does give more confidence, it is still not possible to know that there are no wrongly classified records. This leaves the integrity of the dataset at question, which is the most fundamental element of this project. If the dataset is not credible, the models are futile. To ensure the integrity of the dataset, records would have to be checked individually, although this could bring light to other issues.

6.1.5 Subjectivity of Spoilers

A spoiler is defined as information about the plot of a motion picture that can spoil a viewer's sense of surprise. While this may seem like a clear-cut definition viewers may not be able to agree on what information would in fact spoiler their "sense of surprise". This is something that is not necessarily a problem with this project, but in this field as a whole. For this issue to be minimised as much as possible, "spoilers" needed to be better defined.

6.2 Future Work

Moving forward, the most crucial thing to do would be to train the transformer model on all of the available data, as discussed above. There are other aspects that could be experimented with also, discussed below.

6.2.1 Other Transformer Models

So far, only one transformer model was used, and since it performed the best, the most logical next step would be to experiment with more transformer models. While BERT is the most common model, others have performed particularly well on NLP tasks also. In particular, XLNet has been found to out-perform BERT on certain NLP tasks [47]. XLNet was presented by Yang et al. in the 2019 paper "XLNet: Generalized Autoregressive Pretraining for Language Understanding" [48]. While XLNet is mainly used for generative NLP tasks, it does perform well on classification tasks.

6.2.2 Social Media Platform Integration

During the initial specification development, another objective was to integrate the resulting model into a platform that would make it accessible to users; while there has been previous research into this area, there has not been concrete development from a user perspective. It became clear early on that developing a model that would be suitable for this would be difficult; while models were reporting decent accuracy scores, precision, recall and F1 scores showed that it was the negative class than was mostly being correctly identified that drove the score up. Due to this, the focus was put solely on developing the best model possible.

Chapter 7

Conclusion

This research aimed to find the best DL model for detecting spoilers in text from three different types: bag-of-words, RNN, and Transformer. The IMDb spoiler dataset was used to train all three types of models, and they were all further tuned to find the most optimal configuration of hyper-parameters, resulting in 30 models in total. Data was pre-processed before training, using various methods. Models were evaluated during tuning using validation accuracy and F1 score, and a deeper understanding of the results was gained by adding recall and precision during final testing.

The results demonstrate that spoiler detection is text is possible, assuming that text is processed as a sequence. The order is important to achieve decent detection; the best recall score was achieved by the transformer model, at 76%, which has a self-attention mechanism that finds the relation between the processed word and previous ones. Bag-of-words and LSTM only achieved 55% and 27% recall, much too low for them to be classed as successful. These findings are in-line with the industry standards; transformers are accepted as preforming better when compared to RNNs etc.

While there are areas for improvement, the results shows that this project as a whole was successful. Different types of models were built, and their results provided clear answers on which DL models are best for this task. The resulting best model can be integrated into systems allowing social media users to avoid unwanted spoilers while browsing, so you won't find out that Bond dies from Facebook, but from watching No Time To Die.

Appendix A

Appendix

A.1 Repository

The repository containing all the code developed can be found at the following link:
`https://github.com/ysmnpksy/Final-Project`
The final version of each set is included below, previous versions for each set can be found on the repository only.

Final Project Prototype: Deep Learning Algorithm to Recognize Spoilers

In order to show proof of concept, I have developed a deep learning bag-of-words algorithm as my prototype. Deep learning allows for pattern recognition in words, sentences, and paragraphs, meaning it can be applied to this project in order to classify whether a piece of text includes a spoiler or not.

My development process can be examined through this notebook. I have included explanations and justification in text cells in-between the code to show my understand and demonstrate how I will continue the development phase moving forward.

Part 1: Kaggle & Dataset

As the first step, I am preparing the notebook by installing Kaggle and then the dataset.

1.1: Installing Kaggle

I have installed Kaggle using the following commands. While the output shows "already satisfied", thanks to Google Colab, I am leaving the code here for good practice, as it could be needed when running this notebook on a different platform.

I have generated an API login using my Kaggle account and uploaded the JSON file, `kaggle.json`, containing the username and key, to this notebook. If running this notebook, and not viewing via the HTML format, the API key, included in the submission folder, needs to be uploaded to the notebook before running the first code cell.

```
# making kaggle directory
! mkdir ~/.kaggle

# copying api login info into directory
! cp kaggle.json ~/.kaggle/

# allocating required permissions
! chmod 600 ~/.kaggle/kaggle.json
```

```
mkdir: cannot create directory ‘/root/.kaggle’: File exists
```

1.2: Installing Dataset

The dataset used for this prototype and any further iterations of this model is the IMDB Spoiler Dataset, downloaded from Kaggle. This dataset was developed using IMDB and contains two different databases: movies and reviews. The reviews dataset, which will be the one used for this model, contains 573,913 records.

IMDB allows users to know which reviews contain spoilers by tagging the reviews with "Warning: Spoilers".

Reviews which include spoilers but have not been indicated to do so by the reviewer will be removed when reported under the "Spoiler without warning" option. Due to this, I can be confident that this dataset includes accurate information, making this dataset was the best option for the project; the only downside is that the reviews are only taken from movies, and not a mix of movies and TV shows, which would have been more applicable to this project.

In the code cell below, I am downloading the dataset from Kaggle and unzipping it to make the data accessible.

```
! kaggle datasets download rmisra/imdb-spoiler-dataset

# unzipping dataset
! unzip imdb-spoiler-dataset.zip
```

```
Downloading imdb-spoiler-dataset.zip to /content
 97% 323M/331M [00:10<00:00, 37.3MB/s]
100% 331M/331M [00:10<00:00, 32.5MB/s]
Archive:  imdb-spoiler-dataset.zip
  inflating: IMDB_movie_details.json
  inflating: IMDB_reviews.json
```

1.2.1: Reviews Database

As mentioned above, the database I will be using from this dataset is the reviews database, which has over five hundred thousand reviews. This database contains information about each review, such as the date it was made, the movie ID of the movie it was about, the review text itself and a boolean field which indicates whether or not it is a spoiler.

The code cell below shows information about this database and a sample consisting of the first 5 reviews. As shown, this dataset consists of 573,913 records, 150,924 of them which contain spoilers. This is a significant number of records and should be enough to train this algorithm without overfitting issues.

```
# importing pandas to read the JSON files
import pandas as pd

# information regarding reviews file
all_reviews = pd.read_json('../content/IMDB_reviews.json', lines=True)

print('Total number of reviews:', all_reviews['review_date'].count())
print('Total number of reviews that contain spoilers:', all_reviews['is_spoiler'].sum())
print('User reviews shape:', all_reviews.shape)
print()

print('First 5 user reviews:')
all_reviews.head()
```

```
Total number of reviews: 573913
Total number of reviews that contain spoilers: 150924
User reviews shape: (573913, 7)
```

```
First 5 user reviews:
```

	review_date	movie_id	user_id	is_spoiler	review_text	rating	review_summary
0	10 February 2006	tt0111161	ur1898687	True	In its Oscar year, Shawshank Redemption (writt...	10	A classic piece of unforgettable film-making.
1	6 September 2000	tt0111161	ur0842118	True	The Shawshank Redemption is without a doubt on...	10	Simply amazing. The best film of the 90's.
2	3 August 2001	tt0111161	ur1285640	True	I believe that this film is the best story eve...	8	The best story ever told on film
3	1 September 2002	tt0111161	ur1003471	True	**Yes, there are SPOILERS here**This film has ...	10	Busy dying or busy living?
4	20 May 2004	tt0111161	ur0226855	True	At the heart of this extraordinary movie is a ...	8	Great story, wondrously told and acted

```
all_reviews.iloc[0].review_text
```

```
'In its Oscar year, Shawshank Redemption (written and directed by Frank Darabont, after the novella Ri
```

1.2.2: Movies Database

The second database included in this dataset is the movies database. This database contains all the information regarding each movie itself, such as the genre, duration, and release date. Information about this database and a sample of the first 5 movies can be seen below.

```
# information regarding movie details file
all_movies = pd.read_json('../content/IMDB_movie_details.json', lines=True)

print('Total number of movies:', all_movies['movie_id'].count())
print('Movie details shape:', all_movies.shape)
```

```
print()

print('First 5 movie details:')

all_movies.head()
```

```
Total number of movies: 1572
Movie details shape: (1572, 7)

First 5 movie details:
```

	movie_id	plot_summary	duration	genre	rating	release_date	plot_synopsis
0	tt0105112	Former CIA analyst, Jack Ryan is in England wi...	1h 57min	[Action, Thriller]	6.9	1992-06-05	Jack Ryan (Ford) is on a "working vacation" in...
1	tt1204975	Billy (Michael Douglas), Paddy (Robert De Niro...	1h 45min	[Comedy]	6.6	2013-11-01	Four boys around the age of 10 are friends in ...
2	tt0243655	The setting is Camp Firewood, the year 1981. I...	1h 37min	[Comedy, Romance]	6.7	2002-04-11	
3	tt0040897	Fred C. Dobbs and Bob Curtin, both down on the...	2h 6min	[Adventure, Drama, Western]	8.3	1948-01-24	Fred Dobbs (Humphrey Bogart) and Bob Curtin (T...
4	tt0126886	Tracy Flick is running unopposed for this year...	1h 43min	[Comedy, Drama, Romance]	7.3	1999-05-07	Jim McAllister (Matthew Broderick) is a much-a...

Part 2: Processing the data

The data needs to be processed before a model can be built. Here I will be loading the reviews and the spoiler labels into lists tokenizing using one-hot encoding, and splitting the dataset into training, validation and testing sets.

2.1: Loading the data

Here the labels are being added to a `labels` list while the review text is being added to a `texts` list.

```
import json

labels = []
texts = []

with open('IMDB_reviews.json', 'r') as json_file:
  for jsonObj in json_file:
    data = json.loads(jsonObj)
    if data['is_spoiler'] == True:
       labels.append(1)
    else:
       labels.append(0)
    texts.append(data['review_text'])
```

2.2: Tokenizing the text

Before splitting, I am tokenizing the text data using one-hot encoding. To do this I am using the `Tokenizer` module from Keras.

I have cut the reviews of after a maximum of 500 words and am only considering the most frequent 15000 words in the dataset.

After tokenizing, I have vectorized the data using the `pad_sequences` module from Keras, which converts the list of tokenized data, integers, into 2D tensors, which can then be fed into the neural network. I have also vectorized the list of labels by converting it to a list of floating point numbers.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 500 # cuts reviews off after 500 words
max_words = 15000 # considers only the top 15000 words in the dataset

# tokenizing texts
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

# vectorizing texts
data = pad_sequences(sequences, maxlen=maxlen)

# vectorizing labels
labels = np.asarray(labels).astype('float32')

print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
```

```
Found 379856 unique tokens.
Shape of data tensor: (573913, 500)
Shape of label tensor: (573913,)
```

2.3: Splitting Data

Initially, I am splitting the dataset into training and testing sets. This way I will be testing the model on unseen data only, ensuring information leaking does not affect the final results.

I am first shuffling the data, since the JSON file lists all reviews including spoilers first, then splitting it. I will be training on 200,000 samples and testing using 100.000 for this prototype.

```
# suffling data
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

# setting number of training and testing samples
training_samples = 200000
testing_samples = 100000

# splitting into training and testing sets
x_train = data[:training_samples]
y_train = labels[:training_samples]
x_test = data[training_samples: training_samples + testing_samples]
y_test = labels[training_samples: training_samples + testing_samples]
```

```
print('Training data shape:',x_train.shape)
print('Training labels shape:', y_train.shape)

print('Test data shape:',x_test.shape)
print('Test labels shape:',y_test.shape)
```

```
Training data shape: (200000, 500)
Training labels shape: (200000,)
Test data shape: (100000, 500)
Test labels shape: (100000,)
```

I am then further splitting the training set into a partial training and validation set. This way the data used to validate will not be the same as that used to train, which could cause the model to overfit. I will be training on 150,000 samples and validating on 50,000.

```
# setting number of training and validation samples
partial_training_samples = 150000
validation_samples = 50000

# splitting into training and validation sets
x_partial_train = data[:partial_training_samples]
```

```
y_partial_train = labels[:partial_training_samples]
x_val = data[partial_training_samples: partial_training_samples + validation_samples]
y_val = labels[partial_training_samples: partial_training_samples + validation_samples]
```

```
print('Partial training data shape:',x_partial_train.shape)
print('Partial training labels shape:', y_partial_train.shape)

print('Validation data shape:',x_val.shape)
print('Validation labels shape:',y_val.shape)
```

```
Partial training data shape: (150000, 500)
Partial training labels shape: (150000,)
Validation data shape: (50000, 500)
Validation labels shape: (50000,)
```

Part 3: The Initial Model

Here I am developing the initial model, which I will then tune.

```
import keras.backend as K
from sklearn.metrics import f1_score as f1

def f1_score(y_true, y_pred):
  tp = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
  posp = K.sum(K.round(K.clip(y_pred, 0, 1)))
  predp = K.sum(K.round(K.clip(y_true, 0, 1)))

  precision = tp / (posp + K.epsilon())
  recall = tp / (predp + K.epsilon())

  f1_score = 2 * (precision * recall) / (precision + recall + K.epsilon())
  return f1_score
```

3.1: Defining Initial Model

The initial model I've defined can be seen below. This model consists of 6 layers.

The first two layers are used for word embeddings, which is used to map human language into a geometric space. The `Embedding` layer takes integers as input, finds the integer in an internal dictionary, and returns corresponding word vectors, allowing for vectors to be associated with words.

The next four layers are classification layers, the last being an output layer.

```
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.keras import layers
from keras.layers import Embedding, Flatten
```

```
model = models.Sequential()

# Embedding layers
# network will learn 8-dimensional embeddings for each of the 15,000 words
model.add(Embedding(15000, 8, input_length=maxlen))
# flattens the 3D tensor output to a 2D tensor
model.add(Flatten())

# training single dense layer for classfication
model.add(layers.Dense(128, activation = 'relu'))
model.add(layers.Dense(64, activation = 'relu'))
model.add(layers.Dense(32, activation = 'relu'))

# output layer
model.add(layers.Dense(1, activation = 'sigmoid'))

model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

model.build()
model.summary()
```

```
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_13 (Embedding)    (None, 500, 8)            120000

 flatten_13 (Flatten)        (None, 4000)              0

 dense_50 (Dense)            (None, 128)               512128

 dense_51 (Dense)            (None, 64)                8256

 dense_52 (Dense)            (None, 32)                2080

 dense_53 (Dense)            (None, 1)                 33

=================================================================
Total params: 642,497
Trainable params: 642,497
Non-trainable params: 0
_________________________________________________________________
```

3.2: Training Model 1

Here I have trained the initial model using the partial training data and validated it on the validation data. I have trained for 10 epochs with a batch size of 32.

```
history = model.fit(x_partial_train, y_partial_train,
                    epochs = 10,
                    batch_size = 32,
```

```
                    validation_data = (x_val, y_val))
```

```
Epoch 1/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.5086 - accuracy: 0.7601 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 54s 11ms/step - loss: 0.4804 - accuracy: 0.7818 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4441 - accuracy: 0.8078 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4041 - accuracy: 0.8310 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.3671 - accuracy: 0.8493 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 60s 13ms/step - loss: 0.3338 - accuracy: 0.8657 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 54s 11ms/step - loss: 0.3065 - accuracy: 0.8779 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.2806 - accuracy: 0.8913 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.2587 - accuracy: 0.9021 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.2362 - accuracy: 0.9133 - f1_scor
```

```
history_dict = history.history
history_dict.keys()
```

Part 4: Tuning & Regualizing

Here I am tuning the learning rate and adding regularization to achieve the highest possible accuracy.

4.1: Learning Rate

I am tuning the learning rate of the optimiser. The learning rate determines how much the weights of the network are updated according to the accuracy achieved. If the learning rate is too small, the model may never find the optimal set of weights as they will be updated in very small amounts. If it is too large, the optimal set of weights could get skipped over during training.

As I will be building many iterations of this model, I have made a `build_model` and `train_model` function to make this process more efficient, shown below.

```
from tensorflow.keras import optimizers

def build_model(learning_rate):
  model = models.Sequential()

  # Embedding layers
  model.add(Embedding(15000, 8, input_length=maxlen))
  # flattens the 3D tensor output to a 2D tensor
  model.add(Flatten())
```

```
  # training single dense layer for classfication
  model.add(layers.Dense(128, activation = 'relu'))
  model.add(layers.Dense(64, activation = 'relu'))
  model.add(layers.Dense(32, activation = 'relu'))

  # outup layer
  model.add(layers.Dense(1, activation = 'sigmoid'))

  model.compile(optimizer=optimizers.RMSprop(learning_rate=learning_rate),
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

  return model

def train_model(learning_rate):
  model = build_model(learning_rate)
  history = model.fit(x_partial_train, y_partial_train,
                    epochs = 10,
                    batch_size = 32,
                    validation_data = (x_val, y_val))

  return history.history
```

Model 2: Learning Rate 0.005

In the initial model, I had the learning rate set to the default. In model 1, I am increasing it to 0.005 to see if this will help achieve a higher accuracy on the validation set. As shown below, this model starts degrading straight away, showing that this learning rate is too high.

```
model_2 = train_model(0.005)
```

```
Epoch 1/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.5171 - accuracy: 0.7565 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.5029 - accuracy: 0.7710 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 68s 15ms/step - loss: 0.4851 - accuracy: 0.7843 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.4621 - accuracy: 0.8029 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 68s 14ms/step - loss: 0.4302 - accuracy: 0.8208 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 68s 15ms/step - loss: 0.4006 - accuracy: 0.8396 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 68s 15ms/step - loss: 0.3712 - accuracy: 0.8531 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.3474 - accuracy: 0.8650 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.3258 - accuracy: 0.8754 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 66s 14ms/step - loss: 0.3087 - accuracy: 0.8834 - f1_scor
```

Model 3: Learning Rate 0.003

In this next model, I am decreasing the learning rate to 0.003. Again, the validation accuracy starts degrading straight away, showing this learning rate is also too high for this model.

```
model_2 = train_model(0.003)
```

```
Epoch 1/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.5108 - accuracy: 0.7593 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 68s 14ms/step - loss: 0.4766 - accuracy: 0.7832 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 68s 14ms/step - loss: 0.4302 - accuracy: 0.8139 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 68s 15ms/step - loss: 0.3810 - accuracy: 0.8425 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 75s 16ms/step - loss: 0.3389 - accuracy: 0.8617 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 68s 15ms/step - loss: 0.3055 - accuracy: 0.8809 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 66s 14ms/step - loss: 0.2771 - accuracy: 0.8936 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 65s 14ms/step - loss: 0.2509 - accuracy: 0.9055 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 66s 14ms/step - loss: 0.2270 - accuracy: 0.9151 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 65s 14ms/step - loss: 0.2083 - accuracy: 0.9253 - f1_scor
```

Model 4: Learning Rate 0.0005

In the next iteration I am decreasing the learning rate to 0.005 to see if this will improve on the validation accuracy compared to the initial model.

```
model_4 = train_model(0.0005)
```

```
Epoch 1/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.5061 - accuracy: 0.7607 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.4725 - accuracy: 0.7840 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 67s 14ms/step - loss: 0.4359 - accuracy: 0.8088 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.3969 - accuracy: 0.8313 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 71s 15ms/step - loss: 0.3615 - accuracy: 0.8504 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.3309 - accuracy: 0.8657 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 70s 15ms/step - loss: 0.3064 - accuracy: 0.8779 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.2825 - accuracy: 0.8889 - f1_scor
```

```
Epoch 9/10
4688/4688 [==============================] - 68s 14ms/step - loss: 0.2604 - accuracy: 0.9002 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 69s 15ms/step - loss: 0.2395 - accuracy: 0.9099 - f1_scor
```

4.2: Weight Regularization

I am adding weight regularization, where the network is forced to set weights to small values, making the distribution of them more regular. This is done by adding a cost to the loss function for larger weights. There are two types of costs:

1. L1 regularization: This is where the cost added to the weight is equivalent to the absolute value of the weight coefficients.
2. L2 regularization: This is where the cost added to the weight is equivalent to the square of the weight coefficients.

These two types of costs can be added to a network individually or simultaneously. I will be testing all three methods to find the one that works best for this model. In order to implement regularizers I have imported the regularizers module from Keras, shown below. I have also made a new build function to train the models.

```
from keras import regularizers

def train_reg_model(model):
  history = model.fit(x_partial_train, y_partial_train,
                      epochs = 10,
                      batch_size = 32,
                      validation_data = (x_val, y_val))

  return history.history
```

Model 5: L1 Regularization

The first model I am training includes L1 regularization. This model peaks at epoch 5, to 0.762, which is not higher than the initial model.

```
model = models.Sequential()
model.add(Embedding(15000, 8, input_length=maxlen))
model.add(Flatten())
model.add(layers.Dense(128, kernel_regularizer=regularizers.l1(0.001), activation = 'relu'))
model.add(layers.Dense(64, kernel_regularizer=regularizers.l1(0.001), activation = 'relu'))
model.add(layers.Dense(32, kernel_regularizer=regularizers.l1(0.001),activation = 'relu'))


model.add(layers.Dense(1, activation = 'sigmoid'))

model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

model.build()
```

```
model_4 = train_reg_model(model)
```

```
Epoch 1/10
4688/4688 [==============================] - 79s 17ms/step - loss: 0.8421 - accuracy: 0.7511 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 77s 16ms/step - loss: 0.7980 - accuracy: 0.7621 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 82s 18ms/step - loss: 0.7927 - accuracy: 0.7642 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 82s 18ms/step - loss: 0.7892 - accuracy: 0.7653 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 77s 17ms/step - loss: 0.7874 - accuracy: 0.7662 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 83s 18ms/step - loss: 0.7857 - accuracy: 0.7672 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.7845 - accuracy: 0.7683 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 79s 17ms/step - loss: 0.7833 - accuracy: 0.7696 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 77s 16ms/step - loss: 0.7824 - accuracy: 0.7702 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 77s 17ms/step - loss: 0.7817 - accuracy: 0.7710 - f1_scor
```

Model 6: L2 Regulaization

The second model I am training implements L2 regularization. This model peaks at epoch 3, to 0.766.

```
model = models.Sequential()
model.add(Embedding(15000, 8, input_length=maxlen))
model.add(Flatten())
model.add(layers.Dense(128, kernel_regularizer=regularizers.l2(0.001), activation = 'relu'))
model.add(layers.Dense(64, kernel_regularizer=regularizers.l2(0.001), activation = 'relu'))
model.add(layers.Dense(32, kernel_regularizer=regularizers.l2(0.001),activation = 'relu'))


model.add(layers.Dense(1, activation = 'sigmoid'))

model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

model.build()

model_5 = train_reg_model(model)
```

```
Epoch 1/10
4688/4688 [==============================] - 80s 17ms/step - loss: 0.5272 - accuracy: 0.7585 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.5080 - accuracy: 0.7667 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.4991 - accuracy: 0.7697 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 84s 18ms/step - loss: 0.4911 - accuracy: 0.7760 - f1_scor
```

```
Epoch 5/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.4827 - accuracy: 0.7829 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.4710 - accuracy: 0.7941 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 84s 18ms/step - loss: 0.4575 - accuracy: 0.8026 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 84s 18ms/step - loss: 0.4433 - accuracy: 0.8136 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 82s 18ms/step - loss: 0.4310 - accuracy: 0.8223 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 78s 17ms/step - loss: 0.4186 - accuracy: 0.8303 - f1_scor
```

Model 7: L1 & L2 Regulaization

The last model I am implementing includes L1 and L2 regularization simultaneously. This model achieved a highest validation accuracy of 0.765 at epoch 8.

```
model = models.Sequential()
model.add(Embedding(15000, 8, input_length=maxlen))
model.add(Flatten())
model.add(layers.Dense(128, kernel_regularizer=regularizers.l1_l2(l1=0.001, l2=0.001), activation = 'r
model.add(layers.Dense(64, kernel_regularizer=regularizers.l1_l2(l1=0.001, l2=0.001), activation = 're
model.add(layers.Dense(32, kernel_regularizer=regularizers.l1_l2(l1=0.001, l2=0.001),activation = 'rel


model.add(layers.Dense(1, activation = 'sigmoid'))

model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

model.build()

model_6 = train_reg_model(model)
```

```
Epoch 1/10
4688/4688 [==============================] - 92s 19ms/step - loss: 0.8503 - accuracy: 0.7492 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 90s 19ms/step - loss: 0.7976 - accuracy: 0.7618 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 91s 20ms/step - loss: 0.7923 - accuracy: 0.7643 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 90s 19ms/step - loss: 0.7899 - accuracy: 0.7651 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 92s 20ms/step - loss: 0.7880 - accuracy: 0.7659 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 87s 19ms/step - loss: 0.7871 - accuracy: 0.7667 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 73s 16ms/step - loss: 0.7864 - accuracy: 0.7672 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 73s 16ms/step - loss: 0.7857 - accuracy: 0.7688 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 73s 16ms/step - loss: 0.7849 - accuracy: 0.7688 - f1_scor
```

```
Epoch 10/10
4688/4688 [==============================] - 73s 16ms/step - loss: 0.7843 - accuracy: 0.7691 - f1_scor
```

4.4: Dropout

Dropout is another regularizing technique which I will be implementing. It drops, i.e., sets to zero, a number of output features of a model during training, breaking apart coincidental patterns that aren't significant, allowing the final model to be more generalisable. I have remade the build_model and train_model functions, shown below.

```
def build_drop_model(dropout):
  model = models.Sequential()
  model.add(Embedding(15000, 8, input_length=maxlen))
  model.add(Flatten())
  model.add(layers.Dense(128, activation = 'relu'))
  model.add(layers.Dropout(dropout)) # dropout layer
  model.add(layers.Dense(64, activation = 'relu'))
  model.add(layers.Dropout(dropout)) # dropout layer
  model.add(layers.Dense(32, kernel_regularizer=regularizers.l2(0.001),activation = 'relu'))
  model.add(layers.Dense(1, activation = 'sigmoid'))

  model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

  return model

def train_drop_model(dropout):
  model = build_drop_model(dropout)
  history = model.fit(x_partial_train, y_partial_train,
                    epochs = 10,
                    batch_size = 32,
                    validation_data = (x_val, y_val))

  return history.history
```

Model 8: Dropout 30%

I am stating by adding a dropout rate of 30% to two layers in the model. This has not significantly changed the highest validation accuracy, which is still at 0.765. This could be because the dropout rate was either too high or too low.

```
model_7 = train_drop_model(0.3)
```

```
Epoch 1/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.5214 - accuracy: 0.7582 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.4972 - accuracy: 0.7730 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.4765 - accuracy: 0.7875 - f1_scor
```

```
Epoch 4/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.4476 - accuracy: 0.8057 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4206 - accuracy: 0.8224 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 58s 12ms/step - loss: 0.3962 - accuracy: 0.8353 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 58s 12ms/step - loss: 0.3761 - accuracy: 0.8467 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.3569 - accuracy: 0.8581 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.3415 - accuracy: 0.8661 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.3257 - accuracy: 0.8745 - f1_scor
```

Model 9: Dropout 20%

I have dropped the dropout rate to 20% for the next model. This has increased the highest validation accuracy to 0.767 at epoch 2.

```
model_8 = train_drop_model(0.2)
```

```
Epoch 1/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.5173 - accuracy: 0.7595 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4920 - accuracy: 0.7752 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4686 - accuracy: 0.7921 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.4346 - accuracy: 0.8129 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.4017 - accuracy: 0.8318 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.3749 - accuracy: 0.8458 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.3511 - accuracy: 0.8582 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 50s 11ms/step - loss: 0.3316 - accuracy: 0.8684 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 52s 11ms/step - loss: 0.3132 - accuracy: 0.8769 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 53s 11ms/step - loss: 0.2987 - accuracy: 0.8844 - f1_scor
```

Model 10: Dropout 10%

I am now decreasing the dropout to 10%, hoping it caused validation accuracy to rise even more compared to model 8.

```
model_8 = train_drop_model(0.1)
```

```
Epoch 1/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.5126 - accuracy: 0.7612 - f1_scor
Epoch 2/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.4870 - accuracy: 0.7771 - f1_scor
Epoch 3/10
4688/4688 [==============================] - 54s 12ms/step - loss: 0.4614 - accuracy: 0.7952 - f1_scor
Epoch 4/10
4688/4688 [==============================] - 56s 12ms/step - loss: 0.4246 - accuracy: 0.8185 - f1_scor
Epoch 5/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.3878 - accuracy: 0.8379 - f1_scor
Epoch 6/10
4688/4688 [==============================] - 58s 12ms/step - loss: 0.3569 - accuracy: 0.8555 - f1_scor
Epoch 7/10
4688/4688 [==============================] - 55s 12ms/step - loss: 0.3295 - accuracy: 0.8683 - f1_scor
Epoch 8/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.3072 - accuracy: 0.8791 - f1_scor
Epoch 9/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.2857 - accuracy: 0.8903 - f1_scor
Epoch 10/10
4688/4688 [==============================] - 57s 12ms/step - loss: 0.2668 - accuracy: 0.8995 - f1_scor
```

5.1: Training Final Model

```
from tensorflow.keras import models
from tensorflow.keras import layers
from keras.layers import Embedding, Flatten

model = models.Sequential()
model.add(Embedding(15000, 8, input_length=maxlen))
model.add(Flatten())
model.add(layers.Dense(128,  activation = 'relu'))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(64,  activation = 'relu'))
model.add(layers.Dropout(0.1))
model.add(layers.Dense(32, activation = 'relu'))
model.add(layers.Dense(1, activation = 'sigmoid'))

model.compile(optimizer = 'rmsprop',
              loss = 'binary_crossentropy',
              metrics = ['accuracy', f1_score])

model.fit(x_train, y_train, epochs = 2, batch_size = 32)
```

```
Epoch 1/2
6250/6250 [==============================] - 67s 11ms/step - loss: 0.5092 - accuracy: 0.7618 - f1_scor
Epoch 2/2
6250/6250 [==============================] - 71s 11ms/step - loss: 0.4925 - accuracy: 0.7747 - f1_scor
```

```
<keras.callbacks.History at 0x7f84c689a0d0>
```

```
results = model.evaluate(x_test, y_test)
```

```
3125/3125 [==============================] - 9s 3ms/step - loss: 0.5217 - accuracy: 0.7571 - f1_score:
```

```
y_pred = (model.predict(x_test) > 0.5).astype("int32")
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
print('Accuracy: %.2f' % (accuracy_score(y_test, y_pred)))
print('Precision score: %.2f' % (precision_score(y_test, y_pred)))
print('Recall score: %.2f' % (recall_score(y_test, y_pred)))
print('F1 score: %.2f' % (f1_score(y_test, y_pred)))
```

```
Accuracy: 0.76
Precision score: 0.55
Recall score: 0.43
F1 score: 0.48
```

LSTM & GRU Layers

By Yasmin Paksoy

Table of Contents


```
from google.colab import files

files.upload()
```

Browse... No files selected.

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

```
{'kaggle.json': b'{"username":"yasminpaksoy","key":"9092d77ded0787db0dc92dec0c6c058c"}'}
```

```
# making kaggle directory
! mkdir ~/.kaggle

# copying api login info into directory
```

```
! cp kaggle.json ~/.kaggle/

# allocating required permissions
! chmod 600 ~/.kaggle/kaggle.json
```

```
! kaggle datasets download rmisra/imdb-spoiler-dataset

# unzipping dataset
! unzip imdb-spoiler-dataset.zip
```

```
Downloading imdb-spoiler-dataset.zip to /content
 98% 325M/331M [00:03<00:00, 111MB/s]
100% 331M/331M [00:04<00:00, 86.8MB/s]
Archive:  imdb-spoiler-dataset.zip
  inflating: IMDB_movie_details.json
  inflating: IMDB_reviews.json
```

```
# importing pandas to read the JSON files
import pandas as pd

# information regarding reviews file
all_reviews = pd.read_json('../content/IMDB_reviews.json', lines=True)

print('Total number of reviews:', all_reviews['review_date'].count())
print('Total number of reviews that contain spoilers:', all_reviews['is_spoiler'].sum())
print('User reviews shape:', all_reviews.shape)
print()

print('First 5 user reviews:')
all_reviews.head()
```

```
Total number of reviews: 573913
Total number of reviews that contain spoilers: 150924
User reviews shape: (573913, 7)

First 5 user reviews:
```

	review_date	movie_id	user_id	is_spoiler	review_text	rating	review_summary
0	10 February 2006	tt0111161	ur1898687	True	In its Oscar year, Shawshank Redemption (writt...	10	A classic piece of unforgettable film-making.
1	6 September 2000	tt0111161	ur0842118	True	The Shawshank Redemption is without a	10	Simply amazing. The best film of the 90's.

	review_date	movie_id	user_id	is_spoiler	review_text	rating	review_summary
					doubt on...		
2	3 August 2001	tt0111161	ur1285640	True	I believe that this film is the best story eve...	8	The best story ever told on film
3	1 September 2002	tt0111161	ur1003471	True	**Yes, there are SPOILERS here**This film has ...	10	Busy dying or busy living?
4	20 May 2004	tt0111161	ur0226855	True	At the heart of this extraordinary movie is a ...	8	Great story, wondrously told and acted

Preparing the Data

As shown in previous code cells, the reviews database contains 573,913 records, 150,924 of them which contain spoilers.

In the code cell below, I am converting the reviews and their labels into lists from the JSON file they were loaded in with. During this process, I am adding all reviews that contain spoilers to this list but only 150,924 reviews that don't. This leaves me with a fully balances dataset, which can be seen in the calculations below.

```
import json

labels = []
texts = []

with open('IMDB_reviews.json', 'r') as json_file:
  for jsonObj in json_file:
    data = json.loads(jsonObj)
    if data['is_spoiler'] == True:
       labels.append(1)
    else:
       labels.append(0)
    texts.append(data['review_text'])
```

Next, I am preparing the data. For this data to be inputted into a machine-learning model, it needs to be formatted into tensors which are small in value and normalized if heterogeneous. Heterogeneous data refers to the data being in different ranges, and while deep neural networks can adapt to this, it makes learning more difficult [1].

The data is currently not in tensors, so I will be tokenizing the text data using one-hot encoding. To do this I am using the `Tokenizer` module from Keras. I have cut the reviews of after a maximum of 500 words and am

only considering the most frequent 15000 words in the dataset.

After tokenizing, I have vectorized the data using the `pad_sequences` module from Keras, which converts the list of tokenized data, integers, into 2D tensors, which can then be fed into the neural network. I have also vectorized the list of labels by converting it to a list of floating point numbers.

This data is now ready to be inputted into a machine-learning model.

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 500 # cuts reviews off after 500 words
max_words = 15000 # considers only the top 15,000 words in the dataset

# tokenizing texts
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

# vectorizing texts
data = pad_sequences(sequences, maxlen=maxlen)

# vectorizing labels
labels = np.asarray(labels).astype('float32')

print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
```

```
Found 379856 unique tokens.
Shape of data tensor: (573913, 500)
Shape of label tensor: (573913,)
```

Lastly, to prepare the data, I am splitting it into training and testing sets. This way I will be testing the model on unseen data only, ensuring information leaking does not affect the final results. I will be training on 201,848 data points and testing on 100,00.

I am first shuffling the data, since the JSON file lists all reviews including spoilers first, then splitting it.

```
# suffling data
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

# setting number of training and testing samples
training_samples = 523913
testing_samples = 50000

# splitting into training and testing sets
x_train = data[:training_samples]
```

```
y_train = labels[:training_samples]
x_test = data[training_samples: training_samples + testing_samples]
y_test = labels[training_samples: training_samples + testing_samples]
```

```
print('Training data shape:',x_train.shape)
print('Training labels shape:', y_train.shape)

print('Test data shape:',x_test.shape)
print('Test labels shape:',y_test.shape)
```

```
Training data shape: (523913, 500)
Training labels shape: (523913,)
Test data shape: (50000, 500)
Test labels shape: (50000,)
```

Here I am further splitting the training set into partial training and validation. This ensures that the data used to validate will not be the same as that used to train, helping avoid overfitting issues. I will be training on 151,848 samples and validating on 50,000.

```
# setting number of training and validation samples
partial_training_samples = 473913
validation_samples = 50000

# splitting into training and validation sets
x_partial_train_unbal = data[:partial_training_samples]
y_partial_train_unbal = labels[:partial_training_samples]
x_val = data[partial_training_samples: partial_training_samples + validation_samples]
y_val = labels[partial_training_samples: partial_training_samples + validation_samples]
```

```
print('Partial training data shape:',x_partial_train_unbal.shape)
print('Partial training labels shape:', y_partial_train_unbal.shape)

print('Validation data shape:',x_val.shape)
print('Validation labels shape:',y_val.shape)
```

```
Partial training data shape: (473913, 500)
Partial training labels shape: (473913,)
Validation data shape: (50000, 500)
Validation labels shape: (50000,)
```

```
spoilers = 0;
nonSpoilers = 0;

for i in y_partial_train_unbal:
  if i == 1:
    spoilers += 1
  else:
    nonSpoilers += 1
```

```
print("Total number of reviews: " + str(len(labels)))
print("Total nummber with spoilers: " + str(spoilers))
print("Total number without spoilers " + str(nonSpoilers))
```

```
Total number of reviews: 573913
Total nummber with spoilers: 124563
Total number without spoilers 349350
```

```
x_partial_train = []
y_partial_train = []
sum = 0

for data in range(len(y_partial_train_unbal)):
  if y_partial_train_unbal[data] == 1:
    y_partial_train.append(1)
    x_partial_train.append(x_partial_train_unbal[data])
  else:
    if sum < spoilers:
      sum += 1
      y_partial_train.append(0)
      x_partial_train.append(x_partial_train_unbal[data])
```

```
spoilers = 0;
nonSpoilers = 0;

for i in y_partial_train:
  if i == 1:
    spoilers += 1
  else:
    nonSpoilers += 1

print("Total number of reviews: " + str(len(x_partial_train)))
print("Total nummber with spoilers: " + str(spoilers))
print("Total number without spoilers " + str(nonSpoilers))
```

```
Total number of reviews: 249126
Total nummber with spoilers: 124563
Total number without spoilers 124563
```

```
x_val = np.array(x_val)
y_val = np.array(y_val)
x_partial_train = np.array(x_partial_train)
y_partial_train = np.array(y_partial_train)
x_test = np.array(x_test)
y_test = np.array(y_test)
```

Build Functions

I have included all the functions I will use below.

The code functions I have included are:

- Build baseline model.
- Build scaled up model.
- Build model with dropout and recurrent dropout.
- Plot loss function.
- Plot final loss function.
- Plot accuracy function.
- Plot final accuracy function.
- Train function.
- Train final model function.
- Evaluate function.

```
# Importing all necessary libraries
import tensorflow as tf
import keras
import matplotlib.pyplot as plt

from gc import callbacks
from keras import models, layers
from tensorflow.keras import optimizers
from keras.layers import Embedding, Flatten, LSTM
```

```
import keras.backend as K
from sklearn.metrics import f1_score as f1

def f1_score(y_true, y_pred):
  tp = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
  posp = K.sum(K.round(K.clip(y_pred, 0, 1)))
  predp = K.sum(K.round(K.clip(y_true, 0, 1)))

  precision = tp / (posp + K.epsilon())
  recall = tp / (predp + K.epsilon())

  f1_score = 2 * (precision * recall) / (precision + recall + K.epsilon())
  return f1_score
```

```
def build_scaled_up():
  model = models.Sequential()

  # Embedding layers
  # network will learn 32-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))
```

```
  # LSTM layer
  model.add(LSTM(32))
  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['acc', f1_score])
  model.build()
  model.summary()
  return model
```

```
def build_GRU():
  model = models.Sequential()

  # Embedding layers
  # network will learn 8-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))

  # LSTM layer
  # includes dropout and reccurrent dropout
  model.add(layers.GRU(32, input_shape=(151848, 500)))

  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['acc', f1_score])
  model.build()
  model.summary()
  return model
```

```
def build_dropout(drop, rec_drop):
  model = models.Sequential()

  # Embedding layers
  # network will learn 8-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))

  # LSTM layer
  # includes dropout and reccurrent dropout
  model.add(LSTM(32,
                 dropout=drop,
                 recurrent_dropout=rec_drop
                 ))

  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['acc', f1_score])
  model.build()
```

```
  model.summary()
  return model
```

```
def build_layers(drop, rec_drop, layer):
  model = models.Sequential()

  # Embedding layers
  # network will learn 8-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))

  # LSTM layer
  # includes dropout and reccurrent dropout
  model.add(LSTM(32,
                 dropout=drop,
                 recurrent_dropout=rec_drop,
                 input_shape=(151848, 500),
                 return_sequences=True))

  if layer==2:
    model.add(LSTM(32,
                   dropout=drop,
                   recurrent_dropout=rec_drop))

  if layer==3:
    model.add(LSTM(32,
                   dropout=drop,
                   recurrent_dropout=rec_drop,
                   return_sequences=True))
    model.add(LSTM(32,
                   dropout=drop,
                   recurrent_dropout=rec_drop))

  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(optimizer='rmsprop',
                loss='binary_crossentropy',
                metrics=['acc', f1_score])
  model.build()
  model.summary()
  return model
```

```
def build_learning_rate(drop, rec_drop, lr):
  model = models.Sequential()

  # Embedding layers
  # network will learn 8-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))

  # LSTM layer
  # includes dropout and reccurrent dropout
  model.add(layers.LSTM(32,
                 dropout=drop,
                 recurrent_dropout=rec_drop,
```

```
                input_shape=(151848, 500))
  )

  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  model.compile(optimizer = optimizers.RMSprop(learning_rate = lr),
                loss='binary_crossentropy',
                metrics=['acc', f1_score])
  model.build()
  model.summary()
  return model
```

```
def build_optimizer(drop, rec_drop, optimizer):
  model = models.Sequential()

  # Embedding layers
  # network will learn 8-dimensional embeddings for each of the 15,000 words
  model.add(Embedding(15000, 8))

  # LSTM layer
  # includes dropout and reccurrent dropout
  model.add(layers.LSTM(32,
                dropout=drop,
                recurrent_dropout=rec_drop,
                input_shape=(151848, 500))
  )

  # output layer
  model.add(layers.Dense(1, activation='sigmoid'))

  if optimizer == 'Adam':
    model.compile(optimizer = optimizers.Adam(),
                  loss='binary_crossentropy',
                  metrics=['acc', f1_score])
  else:
    model.compile(optimizer = optimizers.Nadam(),
                  loss='binary_crossentropy',
                  metrics=['acc', f1_score])
  model.build()
  model.summary()
  return model
```

```
def plot_outputs():

  acc = history.history['acc']
  val_acc = history.history['val_acc']
  loss = history.history['loss']
  val_loss = history.history['val_loss']

  epochs = range(1, len(acc) + 1)

  plt.plot(epochs, acc, 'bo', label='Training acc')
  plt.plot(epochs, val_acc, 'b', label='Validation acc')
```

```
  plt.title('Training and validation accuracy')
  plt.legend()

  plt.figure()

  plt.plot(epochs, loss, 'bo', label='Training loss')
  plt.plot(epochs, val_loss, 'b', label='Validation loss')
  plt.title('Training and validation loss')
  plt.legend()

  plt.show()
```

```
callbacks_list = [keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=5
    )]
```

```
def train(model, epoch):
  return model.fit(x_partial_train, y_partial_train,
                   epochs = epoch,
                   batch_size = 128,
                   callbacks = callbacks_list,
                   validation_data = (x_val, y_val))
```

```
def train_final_model(model, epoch):
  return model.fit(x_train, y_train, epochs = epoch, batch_size = 128)
```

```
def evaluate():
  model.evaluate(x_test, y_test)
```

Model 1: First LSTM Model

```
model = build_scaled_up()
```

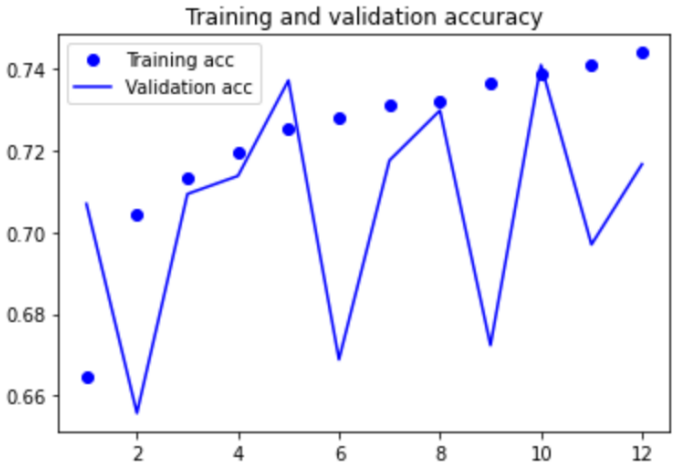
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 8)           120000

 lstm (LSTM)                 (None, 32)                5248

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
```

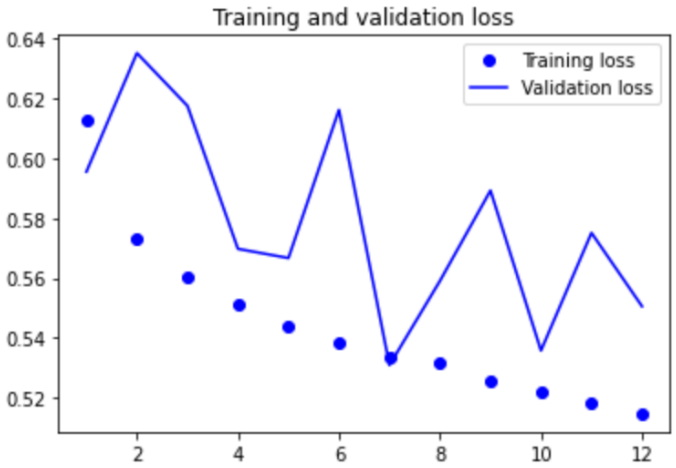
```
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1949/1949 [==============================] - 605s 309ms/step - loss: 0.6128 - acc: 0.6648 - f1_score:
Epoch 2/50
1949/1949 [==============================] - 602s 309ms/step - loss: 0.5730 - acc: 0.7045 - f1_score:
Epoch 3/50
1949/1949 [==============================] - 599s 307ms/step - loss: 0.5603 - acc: 0.7134 - f1_score:
Epoch 4/50
1949/1949 [==============================] - 598s 307ms/step - loss: 0.5512 - acc: 0.7196 - f1_score:
Epoch 5/50
1949/1949 [==============================] - 602s 309ms/step - loss: 0.5440 - acc: 0.7252 - f1_score:
Epoch 6/50
1949/1949 [==============================] - 601s 308ms/step - loss: 0.5383 - acc: 0.7280 - f1_score:
Epoch 7/50
1949/1949 [==============================] - 603s 310ms/step - loss: 0.5336 - acc: 0.7313 - f1_score:
Epoch 8/50
1949/1949 [==============================] - 602s 309ms/step - loss: 0.5316 - acc: 0.7321 - f1_score:
Epoch 9/50
1949/1949 [==============================] - 608s 312ms/step - loss: 0.5255 - acc: 0.7364 - f1_score:
Epoch 10/50
1949/1949 [==============================] - 608s 312ms/step - loss: 0.5218 - acc: 0.7389 - f1_score:
Epoch 11/50
1949/1949 [==============================] - 609s 313ms/step - loss: 0.5184 - acc: 0.7412 - f1_score:
Epoch 12/50
1949/1949 [==============================] - 603s 309ms/step - loss: 0.5148 - acc: 0.7440 - f1_score:
```

```
plot_outputs()
```





Model 2: First GRU Model

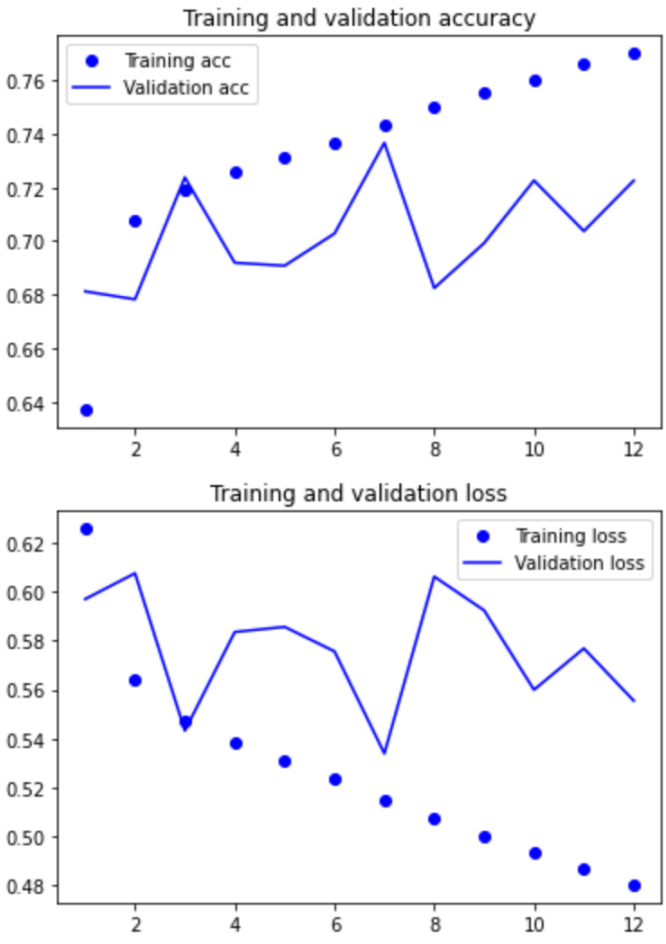
```
model = build_GRU()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 8)           120000

 gru (GRU)                   (None, 32)                4032

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 124,065
Trainable params: 124,065
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 375s 191ms/step - loss: 0.6257 - acc: 0.6372 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 380s 195ms/step - loss: 0.5644 - acc: 0.7074 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 393s 202ms/step - loss: 0.5476 - acc: 0.7190 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 394s 202ms/step - loss: 0.5383 - acc: 0.7258 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 395s 203ms/step - loss: 0.5308 - acc: 0.7311 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 394s 202ms/step - loss: 0.5235 - acc: 0.7369 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 396s 203ms/step - loss: 0.5152 - acc: 0.7433 - f1_score:
Epoch 8/50
1948/1948 [==============================] - 395s 203ms/step - loss: 0.5077 - acc: 0.7498 - f1_score:
Epoch 9/50
1948/1948 [==============================] - 401s 206ms/step - loss: 0.5005 - acc: 0.7551 - f1_score:
Epoch 10/50
1948/1948 [==============================] - 397s 204ms/step - loss: 0.4936 - acc: 0.7602 - f1_score:
Epoch 11/50
1948/1948 [==============================] - 394s 203ms/step - loss: 0.4868 - acc: 0.7659 - f1_score:
Epoch 12/50
1948/1948 [==============================] - 394s 202ms/step - loss: 0.4804 - acc: 0.7700 - f1_score:
```

```
plot_outputs()
```



Tuning: Dropout

Model 3: Rate 20%

```
model = build_dropout(0.2, 0.2)
```

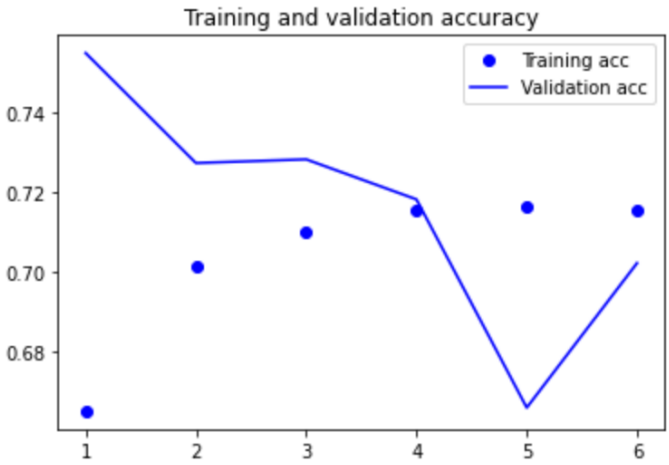
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 8)           120000

 lstm_1 (LSTM)               (None, 32)                5248

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

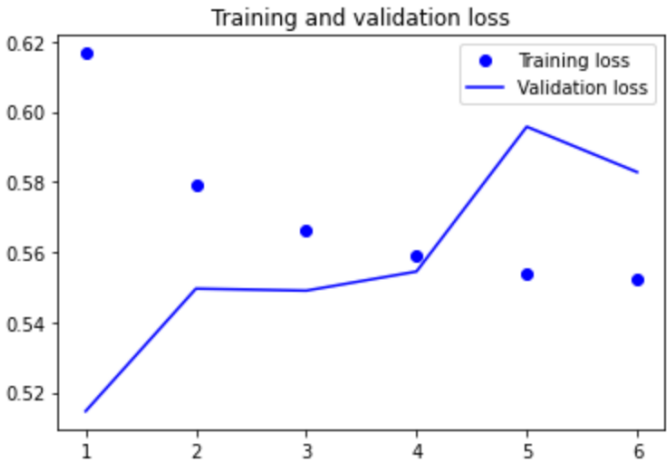
```
history = train(model, 50)
```

```
Epoch 1/50
1949/1949 [==============================] - 1302s 666ms/step - loss: 0.6167 - acc: 0.6651 - f1_score:
Epoch 2/50
1949/1949 [==============================] - 1299s 666ms/step - loss: 0.5791 - acc: 0.7013 - f1_score:
Epoch 3/50
1949/1949 [==============================] - 1298s 666ms/step - loss: 0.5662 - acc: 0.7101 - f1_score:
Epoch 4/50
1949/1949 [==============================] - 1297s 665ms/step - loss: 0.5591 - acc: 0.7155 - f1_score:
Epoch 5/50
1949/1949 [==============================] - 1299s 667ms/step - loss: 0.5539 - acc: 0.7164 - f1_score:
Epoch 6/50
1949/1949 [==============================] - 1300s 667ms/step - loss: 0.5524 - acc: 0.7153 - f1_score:
```

Below I have included graphs showing the training and validation accuracy and loss of this model:

```
plot_outputs()
```





Model 4: Rate 40%

```
model = build_dropout(0.4, 0.4)
```

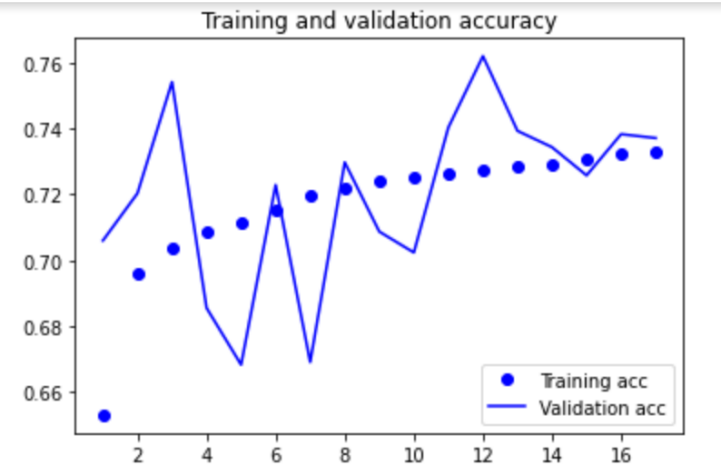
```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, None, 8)           120000

 lstm_4 (LSTM)               (None, 32)                5248

 dense_4 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

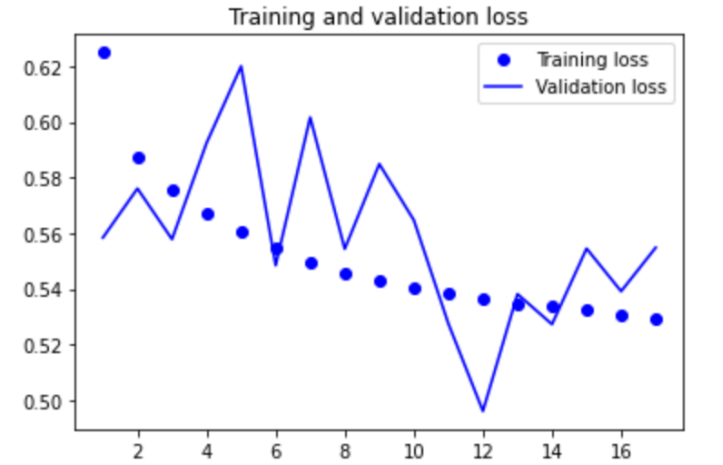
```
history = train(model, 50)
```

```
Epoch 1/50
1949/1949 [==============================] - 1310s 666ms/step - loss: 0.6251 - acc: 0.6530 - f1_score:
Epoch 2/50
1949/1949 [==============================] - 1299s 667ms/step - loss: 0.5874 - acc: 0.6960 - f1_score:
Epoch 3/50
1949/1949 [==============================] - 1298s 666ms/step - loss: 0.5758 - acc: 0.7035 - f1_score:
Epoch 4/50
1949/1949 [==============================] - 1303s 669ms/step - loss: 0.5670 - acc: 0.7088 - f1_score:
Epoch 5/50
1949/1949 [==============================] - 1303s 669ms/step - loss: 0.5609 - acc: 0.7117 - f1_score:
Epoch 6/50
1949/1949 [==============================] - 1298s 666ms/step - loss: 0.5545 - acc: 0.7155 - f1_score:
Epoch 7/50
1949/1949 [==============================] - 1304s 669ms/step - loss: 0.5493 - acc: 0.7197 - f1_score:
Epoch 8/50
1949/1949 [==============================] - 1299s 667ms/step - loss: 0.5457 - acc: 0.7217 - f1_score:
Epoch 9/50
1949/1949 [==============================] - 1297s 666ms/step - loss: 0.5426 - acc: 0.7243 - f1_score:
Epoch 10/50
1949/1949 [==============================] - 1299s 667ms/step - loss: 0.5403 - acc: 0.7253 - f1_score:
Epoch 11/50
1949/1949 [==============================] - 1300s 667ms/step - loss: 0.5385 - acc: 0.7261 - f1_score:
Epoch 12/50
1949/1949 [==============================] - 1301s 668ms/step - loss: 0.5362 - acc: 0.7276 - f1_score:
Epoch 13/50
1949/1949 [==============================] - 1303s 669ms/step - loss: 0.5348 - acc: 0.7288 - f1_score:
Epoch 14/50
1949/1949 [==============================] - 1306s 670ms/step - loss: 0.5336 - acc: 0.7293 - f1_score:
Epoch 15/50
1949/1949 [==============================] - 1320s 677ms/step - loss: 0.5323 - acc: 0.7308 - f1_score:
Epoch 16/50
1949/1949 [==============================] - 1318s 676ms/step - loss: 0.5307 - acc: 0.7326 - f1_score:
Epoch 17/50
1949/1949 [==============================] - 1307s 670ms/step - loss: 0.5292 - acc: 0.7329 - f1_score:
```

Graphs showing the training and validation loss and accuracy can be found below:

```
plot_outputs()
```





Model 5: Rate 30%

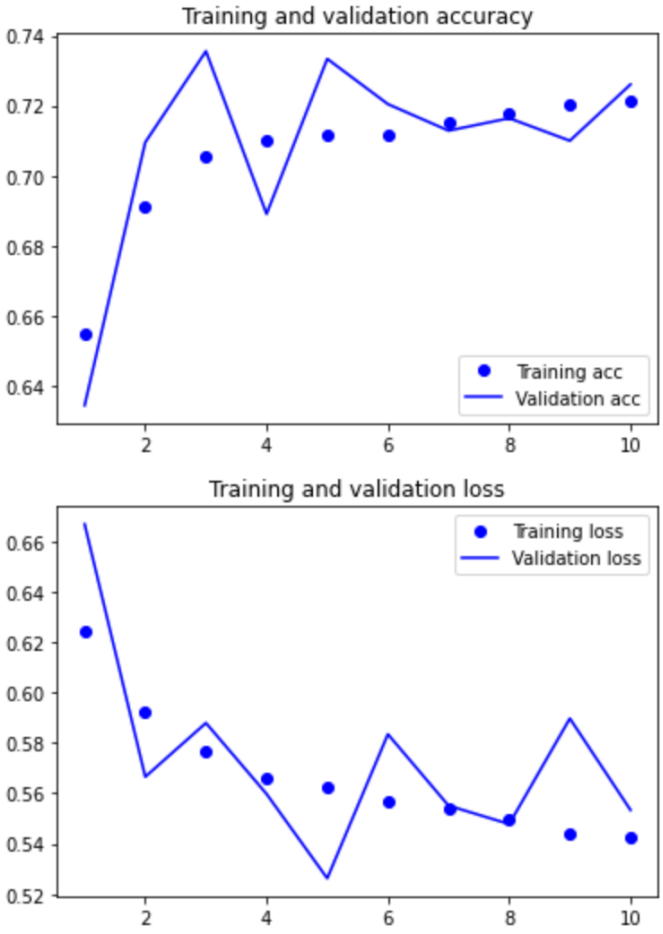
```
model = build_dropout(0.3, 0.3)
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, None, 8)           120000

 lstm_3 (LSTM)               (None, 32)                5248

 dense_3 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1949/1949 [==============================] - 1298s 665ms/step - loss: 0.6244 - acc: 0.6547 - f1_score:
Epoch 2/50
1949/1949 [==============================] - 1296s 665ms/step - loss: 0.5925 - acc: 0.6911 - f1_score:
Epoch 3/50
1949/1949 [==============================] - 1296s 665ms/step - loss: 0.5765 - acc: 0.7054 - f1_score:
Epoch 4/50
1949/1949 [==============================] - 1298s 666ms/step - loss: 0.5657 - acc: 0.7100 - f1_score:
Epoch 5/50
1949/1949 [==============================] - 1295s 664ms/step - loss: 0.5623 - acc: 0.7115 - f1_score:
Epoch 6/50
1949/1949 [==============================] - 1295s 664ms/step - loss: 0.5569 - acc: 0.7119 - f1_score:
Epoch 7/50
1949/1949 [==============================] - 1296s 665ms/step - loss: 0.5538 - acc: 0.7150 - f1_score:
Epoch 8/50
1949/1949 [==============================] - 1331s 683ms/step - loss: 0.5498 - acc: 0.7180 - f1_score:
Epoch 9/50
1949/1949 [==============================] - 1300s 667ms/step - loss: 0.5438 - acc: 0.7203 - f1_score:
Epoch 10/50
1949/1949 [==============================] - 1281s 657ms/step - loss: 0.5424 - acc: 0.7214 - f1_score:
```

```
plot_outputs()
```



Tuning: Number of Layers

Next, I will be changing the architecture by adding layers. Adding layers will increase the capacity of the network, making it more likely to learn important patterns that can allow it to make more accurate predictions.

Failed Model: 2 Layers

For the next model, I will be increasing the number of LSTM layers to 2. The model definition can be seen below.

```
model = build_layers(0.4, 0.4, 2)
```

```
history = train(model, 50)
```

Tuning: Learning Rate

Model 6: 0.003

```
model = build_learning_rate(0.4, 0.4, 0.003)
```

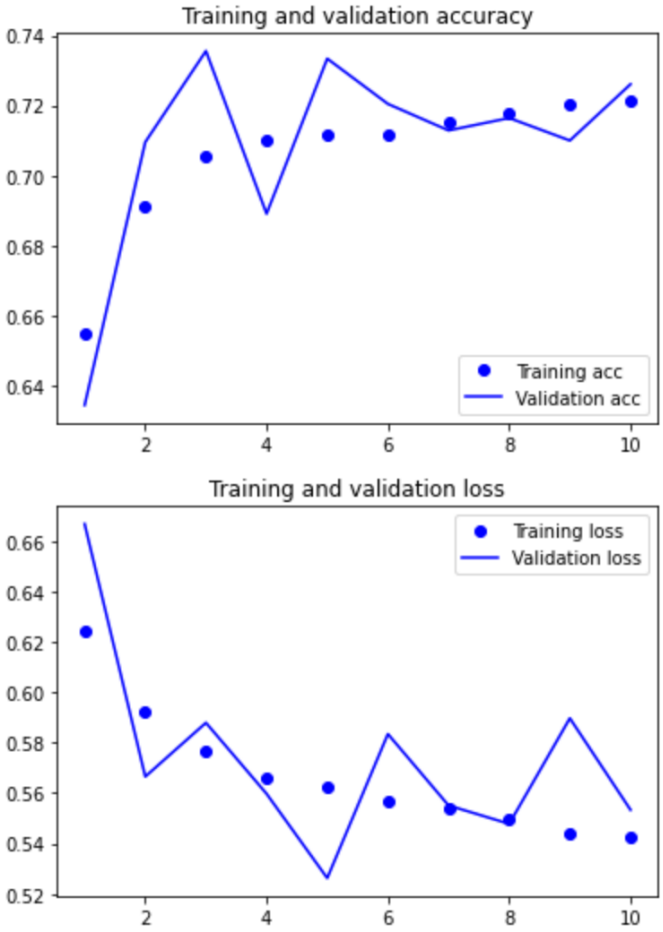
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, None, 8)           120000

 lstm (LSTM)                 (None, 32)                5248

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 961s 492ms/step - loss: 0.6466 - acc: 0.6339 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 961s 493ms/step - loss: 0.6058 - acc: 0.6785 - f1_score:
```

```
Epoch 3/50
1948/1948 [==============================] - 956s 491ms/step - loss: 0.5767 - acc: 0.6950 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 948s 487ms/step - loss: 0.5648 - acc: 0.7047 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 951s 488ms/step - loss: 0.5566 - acc: 0.7116 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 948s 487ms/step - loss: 0.5533 - acc: 0.7128 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 948s 487ms/step - loss: 0.5486 - acc: 0.7174 - f1_score:
Epoch 8/50
1948/1948 [==============================] - 947s 486ms/step - loss: 0.5445 - acc: 0.7200 - f1_score:
Epoch 9/50
1948/1948 [==============================] - 946s 486ms/step - loss: 0.5423 - acc: 0.7221 - f1_score:
Epoch 10/50
1948/1948 [==============================] - 944s 485ms/step - loss: 0.5386 - acc: 0.7253 - f1_score:
Epoch 11/50
1948/1948 [==============================] - 943s 484ms/step - loss: 0.5370 - acc: 0.7262 - f1_score:
```

```
plot_outputs()
```



Model 7: 0.006

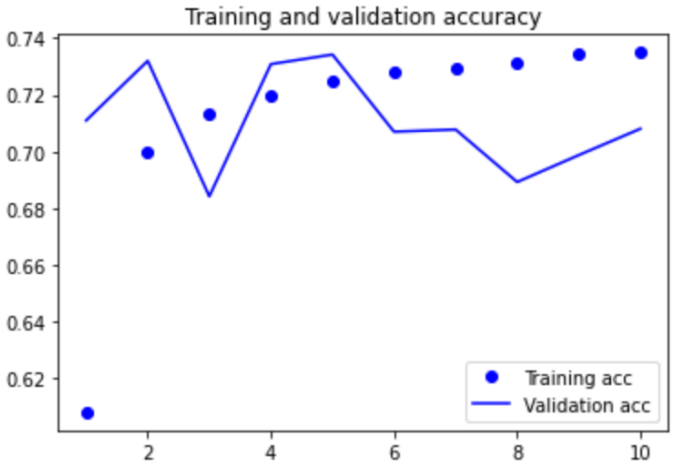
```
model = build_learning_rate(0.4, 0.4, 0.006)
```

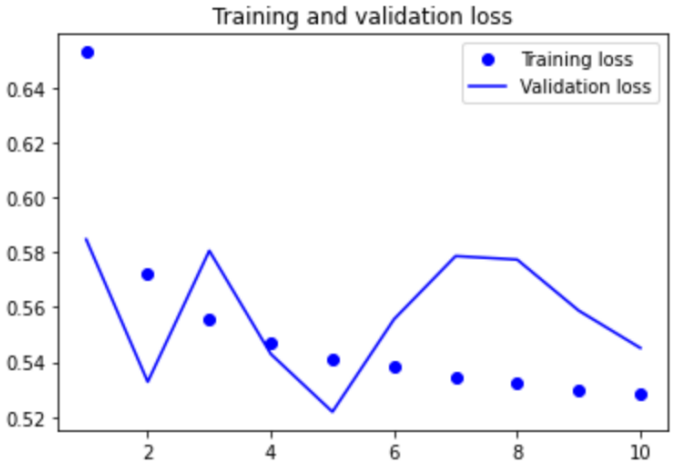
```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, None, 8)           120000

 lstm_1 (LSTM)               (None, 32)                5248

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 949s 486ms/step - loss: 0.6529 - acc: 0.6082 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 948s 487ms/step - loss: 0.5722 - acc: 0.6999 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 942s 483ms/step - loss: 0.5553 - acc: 0.7135 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 947s 486ms/step - loss: 0.5472 - acc: 0.7200 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 946s 486ms/step - loss: 0.5409 - acc: 0.7248 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 946s 486ms/step - loss: 0.5382 - acc: 0.7280 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 945s 485ms/step - loss: 0.5347 - acc: 0.7296 - f1_score:
Epoch 8/50
1948/1948 [==============================] - 942s 484ms/step - loss: 0.5321 - acc: 0.7316 - f1_score:
Epoch 9/50
1948/1948 [==============================] - 943s 484ms/step - loss: 0.5295 - acc: 0.7346 - f1_score:
Epoch 10/50
1948/1948 [==============================] - 946s 486ms/step - loss: 0.5288 - acc: 0.7350 - f1_score:
```

```
plot_outputs()
```





Model 8: 0.004

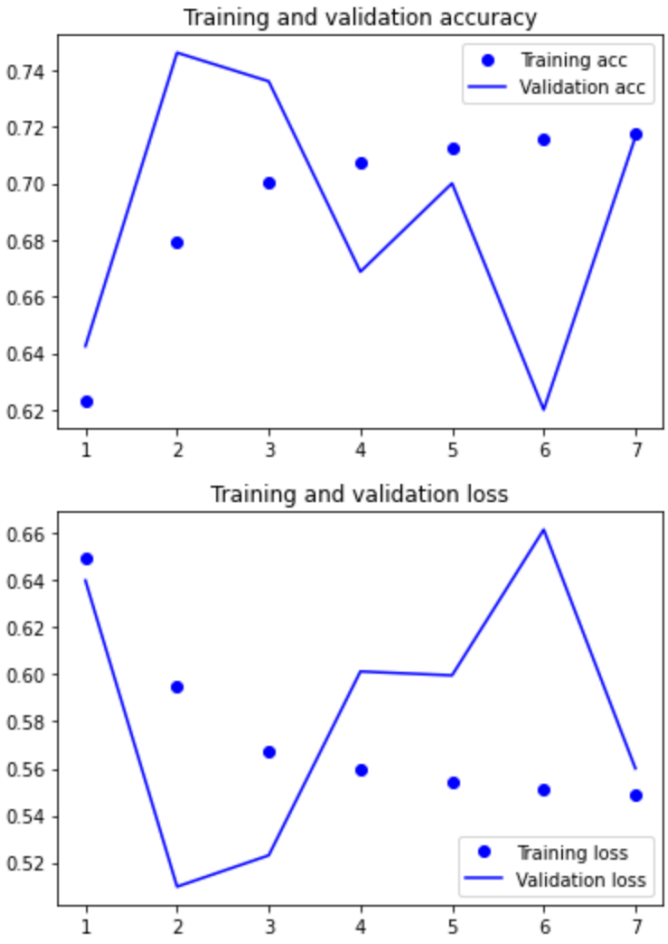
```
model = build_learning_rate(0.4, 0.4, 0.004)
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, None, 8)           120000

 lstm_2 (LSTM)               (None, 32)                5248

 dense_3 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 941s 482ms/step - loss: 0.6496 - acc: 0.6230 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.5950 - acc: 0.6792 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 933s 479ms/step - loss: 0.5675 - acc: 0.7006 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 936s 481ms/step - loss: 0.5600 - acc: 0.7072 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 932s 479ms/step - loss: 0.5539 - acc: 0.7122 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 934s 479ms/step - loss: 0.5508 - acc: 0.7155 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 927s 476ms/step - loss: 0.5487 - acc: 0.7175 - f1_score:
```

```
plot_outputs()
```



Model 9: 0.01

```
model = build_learning_rate(0.4, 0.4, 0.01)
```

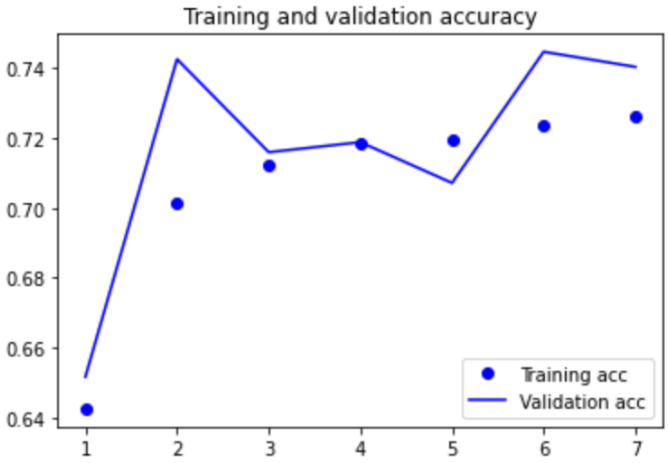
```
Model: "sequential_4"
```

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 embedding_4 (Embedding)     (None, None, 8)           120000

 lstm_3 (LSTM)               (None, 32)                5248

 dense_4 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 924s 473ms/step - loss: 0.6234 - acc: 0.6426 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 922s 473ms/step - loss: 0.5695 - acc: 0.7014 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 923s 474ms/step - loss: 0.5560 - acc: 0.7121 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 928s 476ms/step - loss: 0.5481 - acc: 0.7188 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 931s 478ms/step - loss: 0.5484 - acc: 0.7195 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 921s 473ms/step - loss: 0.5425 - acc: 0.7239 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 921s 473ms/step - loss: 0.5398 - acc: 0.7261 - f1_score:
```

```
plot_outputs()
```



Model 10: Adam Optimizer

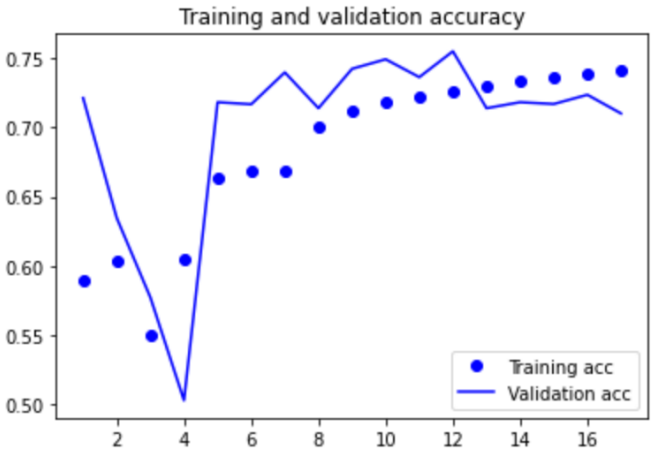
```
model = build_optimizer(0.4, 0.4, 'Adam')
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_5 (Embedding)     (None, None, 8)           120000

 lstm_4 (LSTM)               (None, 32)                5248

 dense_5 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train(model, 50)
```

```
Epoch 1/50
1948/1948 [==============================] - 939s 479ms/step - loss: 0.6668 - acc: 0.5899 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 938s 482ms/step - loss: 0.6543 - acc: 0.6032 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 937s 481ms/step - loss: 0.6840 - acc: 0.5500 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 943s 484ms/step - loss: 0.6586 - acc: 0.6054 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 948s 487ms/step - loss: 0.6140 - acc: 0.6635 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 950s 488ms/step - loss: 0.6087 - acc: 0.6686 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.6111 - acc: 0.6680 - f1_score:
Epoch 8/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.5759 - acc: 0.7004 - f1_score:
Epoch 9/50
1948/1948 [==============================] - 938s 481ms/step - loss: 0.5647 - acc: 0.7114 - f1_score:
Epoch 10/50
1948/1948 [==============================] - 934s 479ms/step - loss: 0.5553 - acc: 0.7178 - f1_score:
Epoch 11/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.5502 - acc: 0.7216 - f1_score:
Epoch 12/50
1948/1948 [==============================] - 938s 481ms/step - loss: 0.5437 - acc: 0.7264 - f1_score:
Epoch 13/50
1948/1948 [==============================] - 933s 479ms/step - loss: 0.5369 - acc: 0.7298 - f1_score:
Epoch 14/50
1948/1948 [==============================] - 937s 481ms/step - loss: 0.5317 - acc: 0.7338 - f1_score:
Epoch 15/50
1948/1948 [==============================] - 936s 480ms/step - loss: 0.5281 - acc: 0.7366 - f1_score:
Epoch 16/50
1948/1948 [==============================] - 937s 481ms/step - loss: 0.5238 - acc: 0.7391 - f1_score:
Epoch 17/50
1948/1948 [==============================] - 944s 484ms/step - loss: 0.5208 - acc: 0.7411 - f1_score:
```

```
plot_outputs()
```



Model 11: Nadam Optimizer

```
model = build_optimizer(0.4, 0.4, 'Nadam')
```

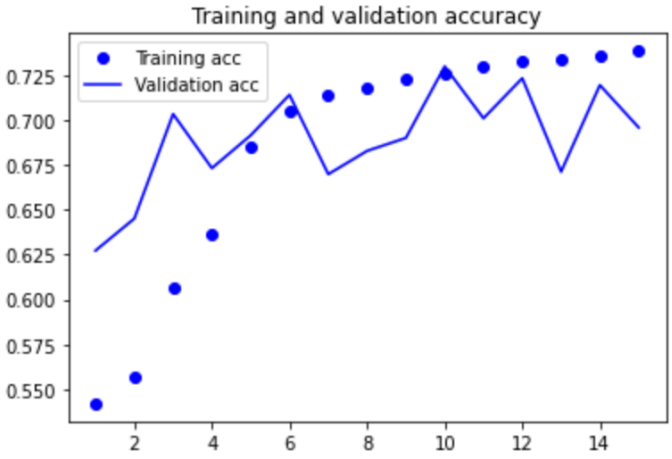
```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_6 (Embedding)     (None, None, 8)           120000

 lstm_5 (LSTM)               (None, 32)                5248

 dense_6 (Dense)             (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

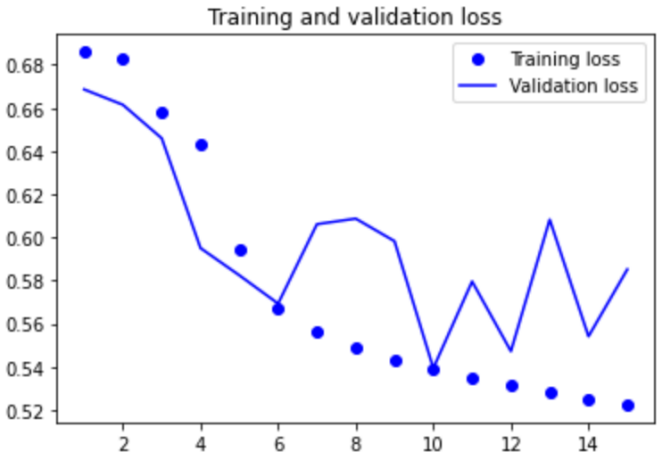
```
history = train(model, 50)
```

```
Epoch 1/50
```

```
1948/1948 [==============================] - 936s 479ms/step - loss: 0.6860 - acc: 0.5421 - f1_score:
Epoch 2/50
1948/1948 [==============================] - 938s 482ms/step - loss: 0.6830 - acc: 0.5571 - f1_score:
Epoch 3/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.6582 - acc: 0.6065 - f1_score:
Epoch 4/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.6432 - acc: 0.6360 - f1_score:
Epoch 5/50
1948/1948 [==============================] - 935s 480ms/step - loss: 0.5948 - acc: 0.6846 - f1_score:
Epoch 6/50
1948/1948 [==============================] - 937s 481ms/step - loss: 0.5667 - acc: 0.7046 - f1_score:
Epoch 7/50
1948/1948 [==============================] - 938s 482ms/step - loss: 0.5564 - acc: 0.7136 - f1_score:
Epoch 8/50
1948/1948 [==============================] - 934s 479ms/step - loss: 0.5488 - acc: 0.7178 - f1_score:
Epoch 9/50
1948/1948 [==============================] - 933s 479ms/step - loss: 0.5429 - acc: 0.7225 - f1_score:
Epoch 10/50
1948/1948 [==============================] - 931s 478ms/step - loss: 0.5386 - acc: 0.7261 - f1_score:
Epoch 11/50
1948/1948 [==============================] - 939s 482ms/step - loss: 0.5346 - acc: 0.7296 - f1_score:
Epoch 12/50
1948/1948 [==============================] - 935s 480ms/step - loss: 0.5312 - acc: 0.7322 - f1_score:
Epoch 13/50
1948/1948 [==============================] - 934s 480ms/step - loss: 0.5283 - acc: 0.7339 - f1_score:
Epoch 14/50
1948/1948 [==============================] - 935s 480ms/step - loss: 0.5253 - acc: 0.7354 - f1_score:
Epoch 15/50
1948/1948 [==============================] - 933s 479ms/step - loss: 0.5226 - acc: 0.7384 - f1_score:
```

```
plot_outputs()
```





Training Final Model

```
final_model = build_dropout(0.4, 0.4)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, None, 8)           120000

 lstm (LSTM)                 (None, 32)                5248

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 125,281
Trainable params: 125,281
Non-trainable params: 0
_________________________________________________________________
```

```
history = train_final_model(final_model, 12)
```

```
Epoch 1/12
4094/4094 [==============================] - 2717s 663ms/step - loss: 0.5313 - acc: 0.7484 - f1_score:
Epoch 2/12
4094/4094 [==============================] - 2670s 652ms/step - loss: 0.5086 - acc: 0.7551 - f1_score:
Epoch 3/12
4094/4094 [==============================] - 2645s 646ms/step - loss: 0.4906 - acc: 0.7692 - f1_score:
Epoch 4/12
4094/4094 [==============================] - 2790s 681ms/step - loss: 0.4811 - acc: 0.7772 - f1_score:
Epoch 5/12
4094/4094 [==============================] - 2809s 686ms/step - loss: 0.4769 - acc: 0.7800 - f1_score:
Epoch 6/12
4094/4094 [==============================] - 2703s 660ms/step - loss: 0.4732 - acc: 0.7827 - f1_score:
Epoch 7/12
4094/4094 [==============================] - 2753s 672ms/step - loss: 0.4704 - acc: 0.7841 - f1_score:
Epoch 8/12
4094/4094 [==============================] - 2595s 634ms/step - loss: 0.4685 - acc: 0.7856 - f1_score:
Epoch 9/12
4094/4094 [==============================] - 2596s 634ms/step - loss: 0.4677 - acc: 0.7858 - f1_score:
Epoch 10/12
4094/4094 [==============================] - 2593s 633ms/step - loss: 0.4673 - acc: 0.7860 - f1_score:
Epoch 11/12
4094/4094 [==============================] - 2592s 633ms/step - loss: 0.4666 - acc: 0.7868 - f1_score:
Epoch 12/12
4094/4094 [==============================] - 2605s 636ms/step - loss: 0.4660 - acc: 0.7870 - f1_score:
```

```
y_pred = final_model.predict(x_test)
```

```
y_pred_labels = np.where(y_pred > 0.5, 1, 0)
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, make_scorer
print('Accuracy: %.2f' % (accuracy_score(y_test, y_pred_labels)))
print('Precision score: %.2f' % (precision_score(y_test, y_pred_labels)))
print('Recall score: %.2f' % (recall_score(y_test, y_pred_labels)))
print('F1 score: %.2f' % (f1_score(y_test, y_pred_labels)))
```

```
Accuracy: 0.78
Precision score: 0.74
Recall score: 0.27
F1 score: 0.40
```

Transformer Models

Loading the Data

```
! pip install datasets
! pip install transformers
```

```
from google.colab import files

files.upload()
```

Browse... No files selected.

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

```
{'kaggle.json': b'{"username":"yasminpaksoy","key":"9092d77ded0787db0dc92dec0c6c058c"}'}
```

```
# making kaggle directory
! mkdir ~/.kaggle

# copying api login info into directory
! cp kaggle.json ~/.kaggle/

# allocating required permissions
! chmod 600 ~/.kaggle/kaggle.json
```

```
! kaggle datasets download rmisra/imdb-spoiler-dataset

# unzipping dataset
! unzip imdb-spoiler-dataset.zip
```

```
Downloading imdb-spoiler-dataset.zip to /content
 99% 329M/331M [00:02<00:00, 146MB/s]
100% 331M/331M [00:02<00:00, 165MB/s]
Archive:  imdb-spoiler-dataset.zip
  inflating: IMDB_movie_details.json
  inflating: IMDB_reviews.json
```

Preparing the Data

```
from datasets import load_dataset
dataset = load_dataset('json', data_files='IMDB_reviews.json')
```

```
Using custom data configuration default-26625881577fe713


Downloading and preparing dataset json/default to /root/.cache/huggingface/datasets/json/default-26625


Downloading data files:   0%|          | 0/1 [00:00<?, ?it/s]


Extracting data files:   0%|          | 0/1 [00:00<?, ?it/s]


Dataset json downloaded and prepared to /root/.cache/huggingface/datasets/json/default-26625881577fe71


  0%|          | 0/1 [00:00<?, ?it/s]
```

```
dataset = dataset['train']
```

```
from datasets import Dataset, DatasetDict
train_testvalid = dataset.train_test_split(test_size=500) # splitting into train and test
train_discard = train_testvalid['train'].train_test_split(train_size=1000) # further splitting train i
test_valid = train_testvalid["test"].train_test_split(test_size=250) # further splitting test into val
datasets = DatasetDict({"train": train_discard["train"], "test": test_valid["train"], "valid": test_va
```

```
datasets # checking output looks correct
```

```
DatasetDict({
    train: Dataset({
        features: ['review_date', 'movie_id', 'user_id', 'is_spoiler', 'review_text', 'rating', 'revie
        num_rows: 1000
    })
    test: Dataset({
        features: ['review_date', 'movie_id', 'user_id', 'is_spoiler', 'review_text', 'rating', 'revie
        num_rows: 250
    })
    valid: Dataset({
        features: ['review_date', 'movie_id', 'user_id', 'is_spoiler', 'review_text', 'rating', 'revie
        num_rows: 250
    })
```

```
})
```

```
datasets = datasets.remove_columns(column_names = ['review_date', 'movie_id', 'user_id', 'rating', 're
```

```
datasets # checking dataset again
```

```
DatasetDict({
    train: Dataset({
        features: ['is_spoiler', 'review_text'],
        num_rows: 1000
    })
    test: Dataset({
        features: ['is_spoiler', 'review_text'],
        num_rows: 250
    })
    valid: Dataset({
        features: ['is_spoiler', 'review_text'],
        num_rows: 250
    })
})
```

```
datasets["train"].features # checking types of features
```

```
{'is_spoiler': Value(dtype='bool', id=None),
 'review_text': Value(dtype='string', id=None)}
```

```
# casting boolean into ints for training later on
from datasets import Value
datasets["train"] = datasets["train"].cast_column('is_spoiler', Value('int64'))
datasets["valid"] = datasets["valid"].cast_column('is_spoiler', Value('int64'))
datasets["test"] = datasets["test"].cast_column('is_spoiler', Value('int64'))
```

```
Casting the dataset:   0%|          | 0/1 [00:00<?, ?ba/s]


Casting the dataset:   0%|          | 0/1 [00:00<?, ?ba/s]


Casting the dataset:   0%|          | 0/1 [00:00<?, ?ba/s]
```

```
datasets["train"].features
```

```
{'is_spoiler': Value(dtype='int64', id=None),
 'review_text': Value(dtype='string', id=None)}
```

```
datasets["valid"].features
```

```
{'is_spoiler': Value(dtype='int64', id=None),
 'review_text': Value(dtype='string', id=None)}
```

```
datasets["test"].features
```

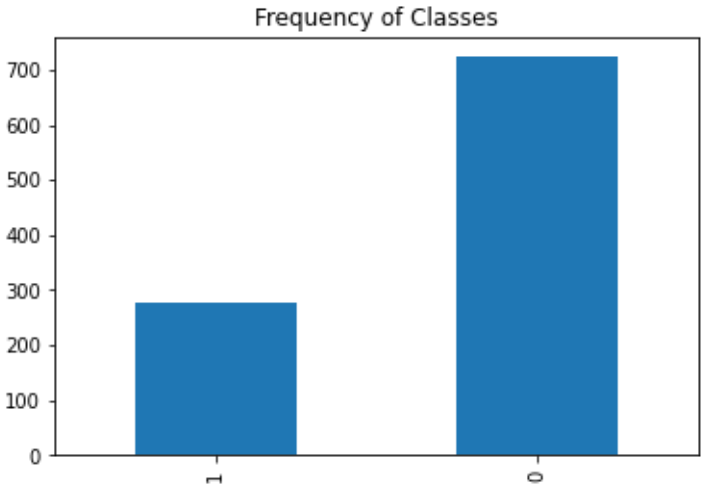
```
{'is_spoiler': Value(dtype='int64', id=None),
 'review_text': Value(dtype='string', id=None)}
```

Checking Train for Imbalance

```
# casting to pandas DataFrame
import pandas as pd
datasets.set_format(type="pandas")
trainDF = datasets["train"][:]
trainDF.head()
```

	is_spoiler	review_text
0	1	First I would like to say that I felt the "Cha...
1	0	It was disgustingly tempting to give this movi...
2	0	This movie has a great production, filming, be...
3	0	Yes, it sounds like a weird combination. A sub...
4	0	This is not a bad film, just not a great film....

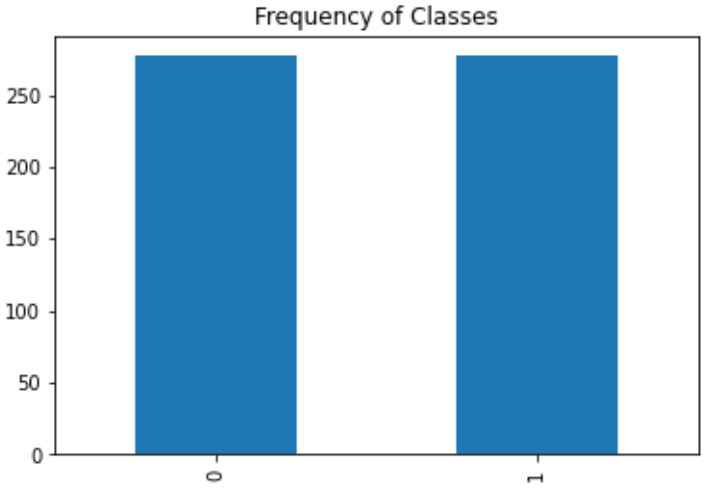
```
# checking imbalance via graph
import matplotlib.pyplot as plt
trainDF["is_spoiler"].value_counts(ascending=True).plot.bar()
plt.title("Frequency of Classes")
plt.show()
```




```
# balancing via undersampling
import numpy as np
from imblearn.under_sampling import RandomUnderSampler

x = trainDF["review_text"].to_numpy()
x = x.reshape(-1, 1)
y = trainDF["is_spoiler"].to_numpy()
y = y.reshape(-1, 1)
rus = RandomUnderSampler()
x, y = rus.fit_resample(x, y)
```

```
trainDF = pd.DataFrame({'is_spoiler': y, 'review_text': x.flatten()}, columns=['is_spoiler', 'review_t
```

```
# checking imbalance
trainDF["is_spoiler"].value_counts(ascending=True).plot.bar()
plt.title("Frequency of Classes")
plt.show()
```




```
len(trainDF) # length of new train dataset
```

```
554
```

```
datasets["train"] = Dataset.from_pandas(trainDF) # converting dataframe back to dataset
```

Hugging Face Format

```
datasets.reset_format() # reset from pandas
```

```
datasets = datasets.rename_column("is_spoiler", "labels") # rename label column
```

```
datasets["train"]
```

```
Dataset({
    features: ['labels', 'review_text'],
    num_rows: 554
})
```

```
datasets["valid"]
```

```
Dataset({
    features: ['labels', 'review_text'],
    num_rows: 250
})
```

```
datasets["test"]
```

```
Dataset({
    features: ['labels', 'review_text'],
    num_rows: 250
})
```

Tokenizer

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
```

```
Downloading:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/483 [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/226k [00:00<?, ?B/s]
```

```
Downloading:   0%|          | 0.00/455k [00:00<?, ?B/s]
```

```
def preprocess_function(examples):
    return tokenizer(examples["review_text"], padding=True, truncation=True)
```

```
encoded_datasets = datasets.map(preprocess_function, batched=True, batch_size=None)
```

```
  0%|          | 0/1 [00:00<?, ?ba/s]
```

```
  0%|          | 0/1 [00:00<?, ?ba/s]
```

```
  0%|          | 0/1 [00:00<?, ?ba/s]
```

```
# checking new columns
print(encoded_datasets["train"].column_names)
```

```
['labels', 'review_text', 'input_ids', 'attention_mask']
```

Models

```
from transformers import AutoModelForSequenceClassification, Trainer, TrainingArguments
num_labels = 2
batch_size = 64
model_name = "name"
logging_steps = len(encoded_datasets["train"]) // batch_size
```

```
# getting metrics ready
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
def metrics(pred):
  zero_division = 0
  logits, labels = pred
  preds = np.argmax(logits, axis=-1)
```

```
  accuracy = accuracy_score(labels, preds)
  recall = recall_score(labels, preds)
  precision = precision_score(labels, preds)
  f1 = f1_score(labels, preds)
  return {"accuracy":accuracy, "recall":recall, "precision":precision, "f1":f1}
```

```
def train_model(model):
  trainer = Trainer(model=model,
                    args=train_args,
                    compute_metrics=metrics,
                    train_dataset=encoded_datasets["train"],
                    eval_dataset=encoded_datasets["valid"],
                    tokenizer=tokenizer
                    )
  return trainer.train()
```

Model 1: Initial Model

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels = num
```

```
Downloading:   0%|          | 0.00/256M [00:00<?, ?B/s]


Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing Distil
- This IS expected if you are initializing DistilBertForSequenceClassification from the checkpoint of
- This IS NOT expected if you are initializing DistilBertForSequenceClassification from the checkpoint
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and in
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-5,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.01,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 21:12, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.690300	0.685966	0.556000	0.825397	0.342105	0.483721
2	0.674100	0.674988	0.584000	0.793651	0.354610	0.490196
3	0.656200	0.581263	0.760000	0.507937	0.524590	0.516129
4	0.614900	0.655422	0.640000	0.793651	0.393701	0.526316
5	0.585800	0.564331	0.724000	0.666667	0.466667	0.549020
6	0.554000	0.551607	0.724000	0.603175	0.463415	0.524138
7	0.505000	0.651649	0.636000	0.777778	0.388889	0.518519
8	0.462600	0.545436	0.724000	0.555556	0.460526	0.503597
9	0.429900	0.596640	0.700000	0.682540	0.438776	0.534161
10	0.404000	0.598151	0.700000	0.682540	0.438776	0.534161

Tuning Learning Rate

Model 2: Learning Rate 2e-6

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels = num
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing Distil
- This IS expected if you are initializing DistilBertForSequenceClassification from the checkpoint of
- This IS NOT expected if you are initializing DistilBertForSequenceClassification from the checkpoint
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and in
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-6,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.01,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 22:12, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.693600	0.698382	0.392000	0.913043	0.301435	0.453237
2	0.692900	0.698061	0.400000	0.927536	0.306220	0.460432
3	0.693100	0.695239	0.452000	0.884058	0.321053	0.471042
4	0.685500	0.695390	0.448000	0.898551	0.321244	0.473282
5	0.687200	0.692984	0.476000	0.869565	0.329670	0.478088
6	0.685000	0.692614	0.488000	0.869565	0.335196	0.483871
7	0.680500	0.691867	0.496000	0.869565	0.338983	0.487805
8	0.685900	0.691740	0.492000	0.869565	0.337079	0.485830
9	0.683500	0.691384	0.492000	0.869565	0.337079	0.485830
10	0.683500	0.691237	0.496000	0.869565	0.338983	0.487805

Model 3: Learning Rate 2e-4

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels = num
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-4,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.01,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 21:20, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.707900	0.672339	0.740000	0.072464	0.833333	0.133333
2	0.689300	0.683495	0.476000	0.927536	0.336842	0.494208
3	0.631700	0.646755	0.572000	0.884058	0.381250	0.532751

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
4	0.539200	0.536840	0.644000	0.782609	0.421875	0.548223
5	0.556700	0.831954	0.600000	0.811594	0.391608	0.528302
6	0.427000	1.110767	0.572000	0.797101	0.371622	0.506912
7	0.252400	0.926237	0.692000	0.463768	0.444444	0.453901
8	0.101300	0.986084	0.692000	0.463768	0.444444	0.453901
9	0.044300	1.250778	0.652000	0.536232	0.402174	0.459627
10	0.021900	1.251190	0.684000	0.420290	0.426471	0.423358

Tuning Weight Decay

Model 4: Weight Decay 0.04

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels = num
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-4,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.04,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 25:42, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.708500	0.681746	0.556000	0.760563	0.364865	0.493151
2	0.660300	0.642788	0.580000	0.788732	0.383562	0.516129
3	0.606500	0.535894	0.756000	0.436620	0.596154	0.504065
4	0.572600	0.635883	0.640000	0.690141	0.418803	0.521277
5	0.500300	1.342482	0.452000	0.873239	0.326316	0.475096

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
6	0.508700	0.602628	0.720000	0.605634	0.505882	0.551282
7	0.473800	0.846239	0.720000	0.619718	0.505747	0.556962
8	0.182500	1.124780	0.620000	0.774648	0.410448	0.536585
9	0.146100	0.740015	0.756000	0.535211	0.575758	0.554745
10	0.194500	0.838702	0.736000	0.605634	0.530864	0.565789

Model 5: Weight Decay 0.08

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", num_labels = num
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-4,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.02,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[80/80 23:01, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.714900	0.649887	0.748000	0.000000	0.000000	0.000000
2	0.678300	0.663792	0.716000	0.158730	0.357143	0.219780
3	0.682700	0.639513	0.644000	0.682540	0.383929	0.491429
4	0.670900	0.565342	0.752000	0.428571	0.509434	0.465517
5	0.581000	0.690596	0.632000	0.650794	0.369369	0.471264
6	0.461400	1.093322	0.560000	0.841270	0.346405	0.490741
7	0.249100	0.745153	0.684000	0.492063	0.397436	0.439716
8	0.164600	0.963204	0.628000	0.698413	0.372881	0.486188

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
9	0.085400	1.109534	0.632000	0.650794	0.369369	0.471264
10	0.044100	1.319980	0.608000	0.698413	0.357724	0.473118

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarning
  _warn_prf(average, modifier, msg_start, len(result))
```

Dropout

```
from transformers import DistilBertConfig
```

```
train_args = TrainingArguments(num_train_epochs=10,
                               output_dir=model_name,
                               learning_rate=2e-4,
                               per_device_train_batch_size=batch_size,
                               per_device_eval_batch_size=batch_size,
                               weight_decay=0.02,
                               evaluation_strategy="epoch",
                               disable_tqdm=False,
                               logging_steps=logging_steps,
                               log_level="error")
```

Model 6: 20%

```
config = DistilBertConfig(dropout=0.2,
                          attention_dropout=0.2)
```

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", config=config)
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[80/80 23:12, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.725800	0.660403	0.748000	0.000000	0.000000	0.000000
2	0.682300	0.677757	0.660000	0.825397	0.412698	0.550265
3	0.659700	0.532959	0.728000	0.523810	0.464789	0.492537

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
4	0.619800	0.566074	0.716000	0.507937	0.444444	0.474074
5	0.443300	0.629145	0.700000	0.571429	0.428571	0.489796
6	0.430100	0.604703	0.736000	0.412698	0.472727	0.440678
7	0.318300	1.292567	0.548000	0.793651	0.333333	0.469484
8	0.192300	0.855883	0.720000	0.539683	0.453333	0.492754
9	0.101700	1.223539	0.628000	0.682540	0.370690	0.480447
10	0.093800	1.184582	0.652000	0.650794	0.386792	0.485207

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1318: UndefinedMetricWarning
  _warn_prf(average, modifier, msg_start, len(result))
```

Model 7: 30%

```
config = DistilBertConfig(dropout=0.3,
                          attention_dropout=0.3)
```

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", config=config)
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 1:28:53, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.706100	0.742532	0.284000	1.000000	0.284000	0.442368
2	0.696100	0.674490	0.720000	0.028169	0.666667	0.054054
3	0.692900	0.691121	0.540000	0.774648	0.357143	0.488889
4	0.680200	0.729027	0.604000	0.704225	0.390625	0.502513
5	0.653600	0.856912	0.504000	0.774648	0.337423	0.470085
6	0.584700	0.754535	0.528000	0.704225	0.340136	0.458716
7	0.521600	0.825264	0.620000	0.647887	0.396552	0.491979
8	0.313400	1.379924	0.528000	0.774648	0.350318	0.482456

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
9	0.299800	1.217356	0.560000	0.704225	0.359712	0.476190
10	0.206000	1.296479	0.548000	0.718310	0.354167	0.474419

Number of Layers

Default: 6

Model 8: 10 Layers

```
config = DistilBertConfig(dropout=0.2,
                          attention_dropout=0.2,
                          n_layers = 10)
```

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", config=config)
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 2:36:11, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.831400	0.762255	0.284000	1.000000	0.284000	0.442368
2	0.701400	0.843346	0.464000	0.859155	0.329730	0.476563
3	0.688600	0.689032	0.592000	0.704225	0.381679	0.495050
4	0.673200	0.579294	0.676000	0.507042	0.439024	0.470588
5	0.642600	0.633940	0.632000	0.647887	0.407080	0.500000
6	0.576000	0.735334	0.660000	0.591549	0.428571	0.497041
7	0.458600	0.851364	0.580000	0.492958	0.336538	0.400000
8	0.294900	0.988496	0.560000	0.661972	0.353383	0.460784
9	0.227400	1.108227	0.540000	0.661972	0.340580	0.449761
10	0.167800	1.076251	0.588000	0.563380	0.357143	0.437158

Model 9: 8 Layers

```
config = DistilBertConfig(dropout=0.2,
```

```
                           attention_dropout=0.2,
                           n_layers = 8)
```

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", config=config)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing Distil
- This IS expected if you are initializing DistilBertForSequenceClassification from the checkpoint of
- This IS NOT expected if you are initializing DistilBertForSequenceClassification from the checkpoint
Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and in
```

```
trainer = train_model(model)
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[90/90 2:01:49, Epoch 10/10]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.763300	0.692859	0.560000	0.876923	0.358491	0.508929
2	0.702000	0.559336	0.744000	0.015385	1.000000	0.030303
3	0.670400	0.536364	0.696000	0.507692	0.428571	0.464789
4	0.559900	0.680211	0.744000	0.030769	0.666667	0.058824
5	0.795200	0.560393	0.732000	0.323077	0.477273	0.385321
6	0.472800	0.759089	0.724000	0.369231	0.461538	0.410256
7	0.420600	0.913932	0.584000	0.630769	0.338843	0.440860
8	0.196400	1.287967	0.520000	0.723077	0.315436	0.439252
9	0.164000	1.176983	0.616000	0.523077	0.343434	0.414634
10	0.087400	1.329993	0.596000	0.584615	0.339286	0.429379

Training Final Model

```
config = DistilBertConfig(dropout=0.2,
                          attention_dropout=0.2,
                          n_layers = 6)
```

```
train_args = TrainingArguments(num_train_epochs=3,
                               output_dir=model_name,
```

```
                                  learning_rate=2e-4,
                                  per_device_train_batch_size=batch_size,
                                  per_device_eval_batch_size=batch_size,
                                  weight_decay=0.02,
                                  evaluation_strategy="epoch",
                                  disable_tqdm=False,
                                  logging_steps=logging_steps,
                                  log_level="error")
```

```
model = AutoModelForSequenceClassification.from_pretrained("distilbert-base-uncased", config = config)
```

```
trainer = Trainer(model=model,
                  args=train_args,
                  compute_metrics=metrics,
                  train_dataset=encoded_datasets["train"],
                  eval_dataset=encoded_datasets["valid"],
                  tokenizer=tokenizer)
trainer.train()
```

```
/usr/local/lib/python3.7/dist-packages/transformers/optimization.py:309: FutureWarning: This implement
  FutureWarning,
```

[27/27 06:50, Epoch 3/3]

Epoch	Training Loss	Validation Loss	Accuracy	Recall	Precision	F1
1	0.704400	0.743731	0.256000	1.000000	0.256000	0.407643
2	0.696500	0.664154	0.740000	0.078125	0.454545	0.133333
3	0.687200	0.713671	0.556000	0.765625	0.337931	0.468900

```
TrainOutput(global_step=27, training_loss=0.6930700408087836, metrics={'train_runtime': 425.3645, 'tra
```

```
preds = trainer.predict(encoded_datasets["test"])
```

[4/4 00:45]

```
y_pred = np.argmax(preds.predictions, axis=-1)
```

```
print('Accuracy: %.2f' % (accuracy_score(preds.label_ids, y_pred)))
print('Precision score: %.2f' % (precision_score(preds.label_ids, y_pred)))
print('Recall score: %.2f' % (recall_score(preds.label_ids, y_pred)))
print('F1 score: %.2f' % (f1_score(preds.label_ids, y_pred)))
```

```
Accuracy: 0.56
Precision score: 0.34
```

```
Recall score: 0.74
F1 score: 0.47
```

Bibliography

[1] "Spoiler definition & meaning," 2022. [Online]. Available: https://www.merriam-webster.com/dictionary/spoiler>

[2] J. E. Rosenbaum and B. K. Johnson, "Who's afraid of spoilers? need for cognition, need for affect, and narrative selection and enjoyment." *Psychology of Popular Media Culture*, vol. 5, no. 3, p. 273, 2016.

[3] W. H. Levine, M. Betzner, and K. S. Autry, "The effect of spoilers on the enjoyment of short stories," *Discourse processes*, vol. 53, no. 7, pp. 513–531, 2016.

[4] A. Abad-Santos and J. Zarracina, "Poll: Americans only hate spoilers if they're not the ones doing the spoiling," 2016. [Online]. Available: https://www.vox.com/2016/7/6/12098828/spoiler-etiquette

[5] M. Meimaridis and T. Oliveira, "The pleasure of spoiling: The spectrum of toxicity behind spoilers in brazil," *Paticipations Journal of Audience and Reception*, vol. 15, no. 1, pp. 272–290, 2018.

[6] S. Kemp, "Digital in the united kingdom: All the statistics you need in 2021," 2021. [Online]. Available: https://datareportal.com/reports/digital-2021-united-kingdom

[7] M. Zoller Seitz, "The results of vulture's spoiler poll are in, and secrecy advocates won't be happy," 2016. [Online]. Available: https://www.vulture.com/2016/05/presenting-the-results-of-vultures-spoiler-poll.html

[8] "Definition of live-tweeting," N/A. [Online]. Available: https://idioms.thefreedictionary.com/live+tweeting

[9] J. Stoll, "Netflix households in the uk 2014-2021," 2022. [Online]. Available: https://www.statista.com/statistics/529734/netflix-households-in-the-uk/

[10] F. Chollet, *Deep learning with Python*. Simon and Schuster, 2018.

[11] I. C. Education, "What is natural language processing?" 2020. [Online]. Available: https://www.ibm.com/cloud/learn/natural-language-processing

[12] B. Profitt, "Spoiler slayer," 2022. [Online]. Available: https://chrome.google.com/webstore/detail/spoiler-slayer/mploapfinhlhbgddjadjnhgiockogjlc

[13] "Spoiler shield," 2022. [Online]. Available: https://chrome.google.com/webstore/detail/spoiler-shield/geedjmnplkdpfigfefbnegoppimkfekn

[14] A. Gaudion, "How to avoid spoilers for your most anticipated movies and tv shows," 2021. [Online]. Available: https://metro.co.uk/2021/12/15/spider-man-no-way-home-how-to-avoid-spoilers-15774891/

[15] A. McArthur, "How to avoid movie spoilers: Tips for movie fans and critics," 2017. [Online]. Available: https://heartsofkyber.wordpress.com/2018/04/27/how-to-avoid-movie-spoilers-tips-for-movie-fans-and-critics/

[16] M. Wan, R. Misra, N. Nakashole, and J. McAuley, "Fine-grained spoiler detection from large-scale review corpora," *arXiv preprint arXiv:1905.13416*, 2019.

[17] B. Chang, I. Lee, H. Kim, and J. Kang, ""killing me" is not a spoiler: Spoiler detection model using graph neural networks with dependency relation-aware attention mechanism," *arXiv preprint arXiv:2101.05972*, 2021.

[18] E. Shleyner, "The ideal social media post length: A guide for every platform," 2018. [Online]. Available: https://blog.hootsuite.com/ideal-social-media-post-length/

[19] S. Jeon, S. Kim, and H. Yu, "Spoiler detection in tv program tweets," *Information Sciences*, vol. 329, pp. 220–235, 2016.

[20] S. Haykin, *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.

[21] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2019.

[22] M. Minsky and S. Papert, *Perceptrons*. MIT press, 1969.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[24] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[25] I. Education, "What are recurrent neural networks?" 2022. [Online]. Available: https://www.ibm.com/cloud/learn/recurrent-neural-networks

[26] V. Lendave, "Lstm vs gru in recurrent neural network: A comparative study," 2021. [Online]. Available: https://analyticsindiamag.com/lstm-vs-gru-in-recurrent-neural-network-a-comparative-study/

[27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[28] F. Chollet, *Deep Learning With Python, Second Edition*, 2nd ed. Manning Publications, 2022.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[30] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[31] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[32] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[33] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[34] "Hugging face – the ai community building the future." [Online]. Available: https://huggingface.co/

[35] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[36] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[37] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[38] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[39] R. Misra, "Imdb spoiler dataset," 05 2019. [Online]. Available: https://www.kaggle.com/rmisra/imdb-spoiler-dataset

[40] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Pearson Education India, 2016.

[41] K. Team, "Keras documentation: Rmsprop." [Online]. Available: https://keras.io/api/optimizers/rmsprop/

[42] ——, "Keras documentation: Adam." [Online]. Available: https://keras.io/api/optimizers/adam/

[43] T. Dozat, "Incorporating nesterov momentum into adam," *N/A*, 2016.

[44] K. Team, "Keras documentation: Nadam." [Online]. Available: https://keras.io/api/optimizers/Nadam/

[45] ——, "Keras documentation: Lstm layer." [Online]. Available: https://keras.io/api/layers/recurrent_layers/lstm/

[46] D. Vasani, "This thing called weight decay," 2019. [Online]. Available: https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab

[47] X. Liang, "What is xlnet and why it outperforms bert," 2019. [Online]. Available: https://towardsdatascience.com/what-is-xlnet-and-why-it-outperforms-bert-8d8fce710335

[48] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "Xlnet: Generalized autoregressive pretraining for language understanding," *Advances in neural information processing systems*, vol. 32, 2019.