Sentiment Analysis with IMDb Dataset

Le Duc Anh Tuan
Hanoi University of Science and Technology
tuan.lda204929@sis.hust.edu.vn

Nguyen Van Thanh Tung
Hanoi University of Science and Technology
tung.nvt190090@sis.hust.edu.vn

Hoang Gia Nguyen
Hanoi University of Science and Technology
nguyen.hg204889@sis.hust.edu.vn

Nguyen Huu Tuan Duy
Hanoi University of Science and Technology
duy.nht204907@sis.hust.edu.vn

Hoang Long Vu
Hanoi University of Science and Technology
vu.hl204897@sis.hust.edu.vn

Abstract

Sentiment analysis is a powerful method to obtain helpful information about a review and classify it as positive or negative sentiment. In this Sentiment Analysis with IMDb Movie Reviews Project, we wish to apply Machine Learning and Deep Learning approaches to measure the accuracy of the model and identify the best algorithm for sentiment analysis.

1 Introduction

IMDb Reviews is a large dataset for binary sentiment classification, provided by Stanford AI Lab. The dataset consists of 50,000 highly polar reviews (in English) with an even number of examples for training and testing purposes. The dataset also contains additional unlabelled data.

The dataset is imported directly from the `tensorflow_datasets`, with the default split of 50/50 for training and test dataset, along with $50,000$ additional unlabelled examples for Unsupervised learning. The average text length for the comments in the dataset is of 231.3 words. In the entire collection, no more than 30 reviews are allowed for any given movie because reviews for the same movie tend to have correlated ratings. Further, the train and test sets contain a disjoint set of movies, so no significant performance is obtained by memorizing movie-unique terms and their associated with observed labels.

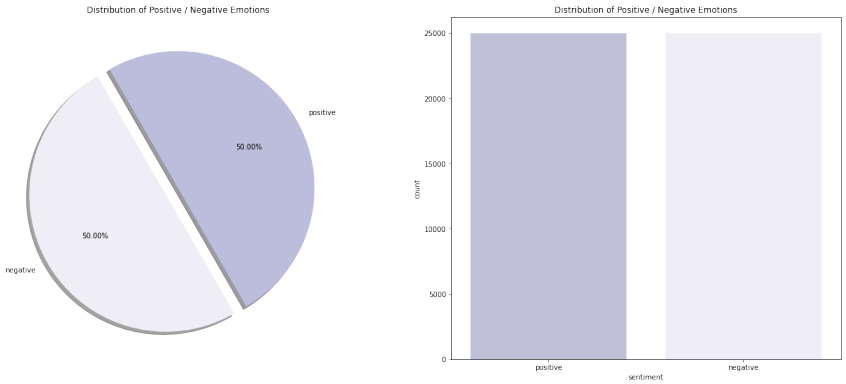


Figure 1: Partition of labels in the dataset (*Source: kaggle.com)*

After text processing phase with Word Vectorisation, we will implement multiple algorithms to evaluate the accuracy. ML algorithms are traditional algorithms including Supervised Learning (Decision Tree, Random Forest, SVM, Logistic Regression, XGBoost) and Unsupervised Learning (K-Means Clustering), whereas DL algorithms work with multilayers artificial neural network and are expected to give better output. The output of the model is to classify a movie review as Positive (1) or Negative (0).

An example a review classified as `Positive` (1):

"One of the other reviewers has mentioned that after watching just 1 Oz episode you'll be hooked. They are right, as this is exactly what happened with me.

The first thing that struck me about Oz was its brutality and unflinching scenes of violence, which set in right from the word GO. Trust me, this is not a show for the faint hearted or timid. This show pulls no punches with regards to drugs, sex or violence. Its is hardcore, in the classic use of the word.

It is called OZ as that is the nickname given to the Oswald Maximum Security State Penitentary. It focuses mainly on Emerald City, an experimental section of the prison where all the cells have glass fronts and face inwards, so privacy is not high on the agenda. Em City is home to many..Aryans, Muslims, gangstas, Latinos, Christians, Italians, Irish and more....so scuffles, death stares, dodgy dealings and shady agreements are never far away.

I would say the main appeal of the show is due to the fact that it goes where other shows wouldn't dare. Forget pretty pictures painted for mainstream audiences, forget charm, forget romance...OZ doesn't mess around. The first episode I ever saw struck me as so nasty it was surreal, I couldn't say I was ready for it, but as I watched more, I developed a taste for Oz, and got accustomed to the high levels of graphic violence. Not just violence, but injustice (crooked guards who'll be sold out for a nickel, inmates who'll kill on order and get away with it, well mannered, middle class inmates being turned into prison bitches due to their lack of street skills or prison experience) Watching Oz, you may become comfortable with what is uncomfortable viewing....thats if you can get in touch with your darker side."

An example of `Negative` review:

"Once again Mr. Costner has dragged out a movie for far longer than necessary. Aside from the terrific sea rescue sequences, of which there are very few I just did not care about any of the characters. Most of us have ghosts in the closet, and Costner's character are realized early on, and then forgotten until much later, by which time I did not care. The character we should really care about is a very cocky, overconfident Ashton Kutcher. The problem is he comes off as kid who thinks he's better than anyone else around him and shows no signs of a cluttered closet. His only obstacle appears to be winning over Costner. Finally when we are well past the half way point of this stinker, Costner tells us all about Kutcher's ghosts. We are told why Kutcher is driven to be the best with no prior inkling or foreshadowing. No magic here, it was all I could do to keep from turning it off an hour in."

2 Data Preprocessing

2.1 Clean and tokenize

We do the basic learning techniques to clean the data:

- remove "(<.*?>)" markup
- remove punctuation marks
- remove white space
- remove all strings that contain a non-letter
- convert to lower
- tokenization: the process of replacing sensitive data with unique identification symbols that retain all the essential information about the data without compromising its security.
- remove stop words

The stopwords vocabulary and word tokenizer is imported from the `nltk` library. Here is a brief glance at some reviews before and after applying the clean and tokenizing process.

	text
0	b"This was an absolutely terrible movie. Don't...
1	b'I have been known to fall asleep during film...
2	b'Mann photographs the Alberta Rocky Mountains...
3	b'This is the kind of film for a snowy Sunday ...
4	b'As others have mentioned, all the women that...
...	...
24995	b'I have a severe problem with this show, seve...
24996	b'The year is 1964. Ernesto "Che" Guevara, hav...
24997	b'Okay. So I just got back. Before I start my ...
24998	b'When I saw this trailer on TV I was surprise...
24999	b'First of all, Riget is wonderful. Good comed...

Figure 2: Some reviews (before cleaning)

	text
0	[this, absolutely, terrible, movie, dont, lur...
1	[i, known, fall, asleep, films, usually, due,...
2	[mann, photographs, alberta, rocky, mountains...
3	[this, kind, film, snowy, sunday, afternoon, ...
4	[as, others, mentioned, women, go, nude, film...
...	...
24995	[i, severe, problem, show, several, actually,...
24996	[the, year, ernesto, che, guevara, cuban, cit...
24997	[okay, got, back, start, review, let, tell, o...
24998	[when, saw, trailer, tv, surprised, may, six,...
24999	[first, riget, wonderful, good, comedy, myste...

Figure 3: Some reviews (after cleaning and tokenizing)

2.2 Machine Learning approach: Tf-idf

TF-IDF (term frequency-inverse document frequency) is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. This is done by multiplying two metrics: how many times a word appears in a document, and the inverse document frequency of the word across a set of documents.

$$\text{idf}(t, D) = \log \frac{|D|}{|\{d \in D; t \in d\}|+1} = \log \frac{|D|}{\text{df}(d,t)+1}$$
$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

Where:

- $|D|$ is the number of documents in the collection.
- $\text{df}(d, t) = |\{d \in D; t \in d\}|$ is the frequency of the document $d \in D$ that the word *t* appears.
- $tf(t, d)$ is the frequency of the word *t* in the document *d*.

The vocabulary of TF-IDF Vectorizer for out dataset would be roughly as below:

```
tfidf.vocabulary_
```

```
{'bthis': 595693,
 'absolutely': 19413,
 'terrible': 4217087,
 'movie': 2786750,
 'dont': 1179277,
 'lured': 2551897,
 'christopher': 770395,
 'walken': 4570086,
 'michael': 2705582,
 'ironside': 2199716,
 'great': 1869786,
```

Figure 4: A glance at TF-IDF vocabulary

The second thing we can do to further improve our model is to provide it with more context. Considering every word independently can lead to some errors. For instance, if the word *"good"* occurs in a text, we will naturally say that this text is positive, even if the actual expression that occurs is actually *"not good"*. These mistakes can be avoided with the concept of N-grams.

An N-gram is a set of N successive words (e.g., very good [2-gram, or *bigram*] and not good at all [4-gram]). Using N-grams, we produce richer word sequences.

The `TfidfVectorizer` implementation provided by the `sklearn` library allows users to integrate N-gram under the parameter named `ngram_range`. This parameter is tuned on various combination of ranges from 1 to 3 (`ngram_range=(1, 1); (1, 2); (2, 2); (1, 3); (2, 3); (3, 3)`), using Grid Search with 10-fold cross validation schema on three Machine Learning algorithms (further details are discussed in the next sections).

2.3 Deep Learning approach: Word Embedding (word2vec)

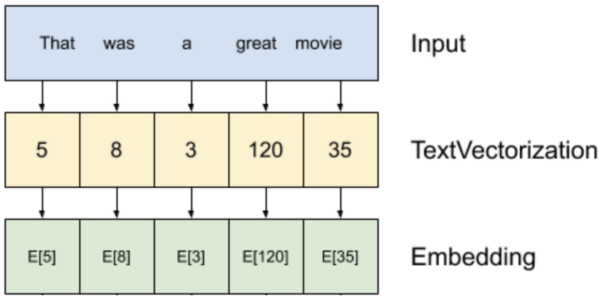


Figure 5: An example of Word Embedding

Natural language is a complex system used to express meanings. In this system, words are the basic unit of the meaning. As the name implies, word vectors are vectors used to represent words, and can also be considered as feature vectors or representations of words. The technique of mapping words to real vectors is called word embedding. In recent years, word embedding has gradually become the basic knowledge of natural language processing.

2.3.1 One-Hot Vectors

We used one-hot vectors to represent words (characters are words) in Section 9.5. Suppose that the number of different words in the dictionary (the dictionary size) is N, and each word corresponds to a different integer (index) from 0 to $N-1$. To obtain the one-hot vector resentation for any word with index i, we create a length- N vector with all 0 s and set the element at position i to 1 . In this way, each word is represted as a vector of length N, and it can be used directly by neural networks. Although one-hot word vectors are easy to construct, they are usually not a good choice. A main reason is that one-hot word vectors cannot accurately express the similarity between different words,

such as the cosine similarity that we often use. For vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their cosine similarity is the cosine of the angle between them:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} \in [-1, 1]$$

Since the cosine similarity between one-hot vectors of any two different words is 0 , one-hot vectors cannot encode similarities among words.

Although one-hot word vectors are easy to construct, they are usually not a good choice. A main reason is that one-hot word vectors cannot accurately express the similarity between different words, such as the cosine similarity that we often use. For vectors $x, y \in \mathrm{R}^d$, their cosine similarity is the cosine of the angle between them. Since the cosine similarity between one-hot vectors of any two different words is 0, one-hot vectors cannot encode similarities among words.

2.3.2 The Skip-Gram Model

The skip-gram model assumes that a word can be used to generate its surrounding words in a text sequence. Take the text sequence "the", "man", "loves", "his", "son" as an example. Let's choose "loves" as the center word and set the context window size to 2. As shown in 6 given the center word "loves", the skip-gram model considers the conditional probability for generating the context words. "the", "man", "his", and "son", which are no more than 2 words away from the center word:

P("the", "man", "his", "son" | "loves").

Assume that the context words are independently generated given the center word (i.e., conditional independence). In this case, the above conditional probability can be rewritten as

$$P(\text{" the" |"loves" }) \cdot P(\text{" man" | "loves" }) \cdot P(\text{ his" | "loves" }) \cdot P(\text{ son" | "loves" }).$$

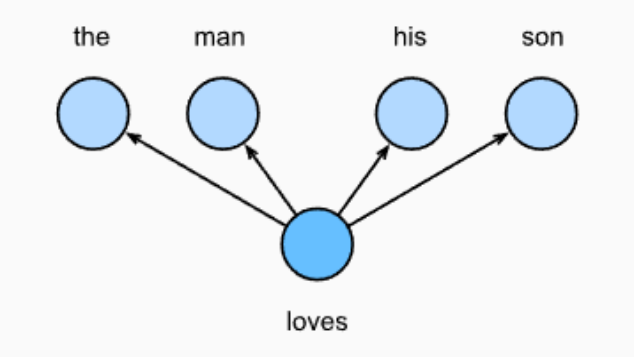


Figure 6: The skip-gram model considers the conditional probability of generating the surrounding context words given a center word.

word with index i in the dictionary, denote by $\mathbf{v}_i \in \mathbb{R}^d$ and $\mathbf{u}_i \in \mathbb{R}^d$ its two vectors when used as a center word and a context word, respectively. The conditional probability of generating any context word w_o (with index o in the dictionary) given the center word w_c (with index c in the dictionary) can be modeled by a softmax operation on vector dot products:

$$P\left(w_o \mid w_c\right) = \frac{\exp\left(\mathbf{u}_o^\top \mathbf{v}_c\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \mathbf{v}_c\right)},$$

where the vocabulary index set $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$. Given a text sequence of length T, where the word at time step t is denoted as $w^{(t)}$. Assume that context words are independently generated given any center word. For context window size m, the likelihood function of the skipgram model is the probability of generating all context words given any center word:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, j \neq 0} P\left(w^{(t+j)} \mid w^{(t)}\right)$$

where any time step that is less than 1 or greater than T can be omitted.

2.3.3 The Continuous Bag of Words (CBOW) Model

The continuous bag of words (CBOW) model is similar to the skip-gram model. The major difference from the skip-gram model is that the example, in the same text sequence "the", "man", "loves", "his", and "son", with "loves" as the center word and the context window size being 2, the continuous bag of words model considers the conditional probability of generating the center word "loves" based on the context word "man", "his" and "son" (as shown in Fig. 7), which is

$$P \textbf{ ("loves" | "the", "man", "his", "son").}$$

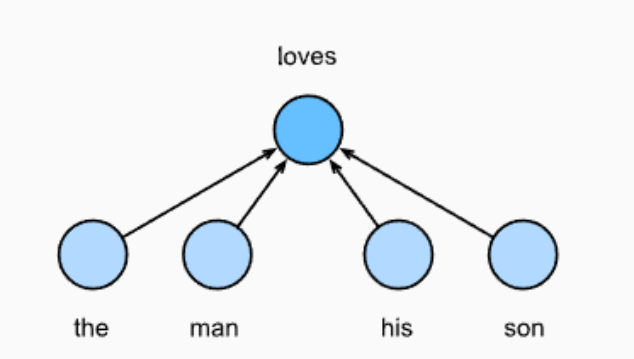


Figure 7: The continuous bag of words model considers the conditional probability of generating the center word given its surrounding context words.¶

Since there are multiple context words in the continuous bag of words model, these context word vectors are averaged in the calculat of the conditional probability. Specifically, for any word with index i in the dictionary, denote by $\mathbf{v}_i \in \mathbb{R}^d$ and $\mathbf{u}_i \in \mathbb{R}^d$ its two vectors when used as a context word and a center word (meanings are switched in the skip-gram model), respectively. The conditional probability of generating center word w_c (with index c in the dictionary) given its surrounding context words $w_{o_1}, \ldots, w_{o_{2m}}$ (with index $o_1, \ldots, o_{2m}$ in the dictionary) can be modeled by

$$P\left(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}\right) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top \left(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}}\right)\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top \left(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}}\right)\right)}.$$

For brevity, let $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$ and $\overline{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots + \mathbf{v}_{o_{2m}}\right)/(2m)$. Then (15.1.10). can be simplified as

$$P\left(w_c \mid \mathcal{W}_o\right) = \frac{\exp\left(\mathbf{u}_c^\top \overline{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \overline{\mathbf{v}}_o\right)}.$$

Given a text sequence of length T, where the word at time step t is denoted as $w^{(t)}$. For context window size m, the likelihood function of the continuous bag of words model is the probability of generating all center words given their context words:

$$\prod_{t=1}^{T} P\left(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}\right).$$

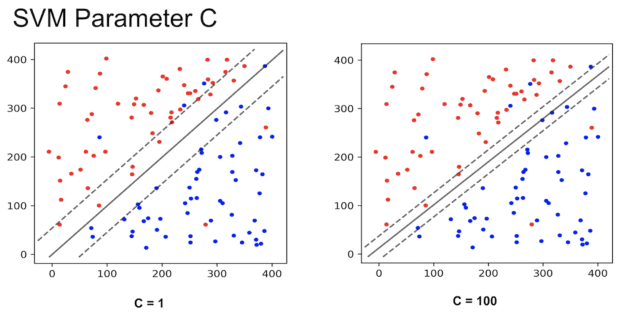
3 Machine Learning

3.1 SVM

SVM is a linear classification method, whose main purpose is to define a separator in the search space that can separate different classes. This separator is commonly referred to as a hyperplane. One of the advantages of this SVM method is that it is quite good at classifying high-dimensional data because the method tries to determine the optimal direction of discrimination in the feature space by examining the right feature combination.

Since our problem is linear (just positive and negative) here, we will go for "linear SVM" (1). The `ngram_range` parameter is tuned to yield the combination of unigram, bi-gram and tri-gram (`ngram_range=(1, 3)`)as the best model for SVM, which resulted in an increase of around 1.5% accuracy on the test set. The `LinearSVC` model provided by `sklearn` supports the tuning of the hyper-paremeter *C*. This parameter tells the SVM optimisation how much we want to avoid misclassifying each training example.

For large values of *C*, the optimisation will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of *C* will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane misclassifies more points.



Example on the effect of C

The default model uses $C = 1.0$ yields the accuracy on the training and test set as 99.996% and 88.98% respectively.

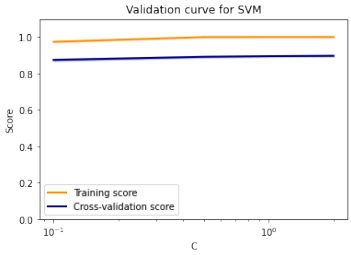


Figure 8: *C* from 0.4 to 0.8

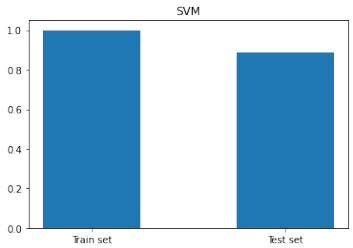


Figure 9: SVM model with $C = 0.5$

We used the Grid Search with 10-fold cross validation schema to find the best value of *C*. The model seems to start overfit since $C = 0.8$, and a value of 0.5 should suffice to balance the running time, and slightly reduce overfitting.

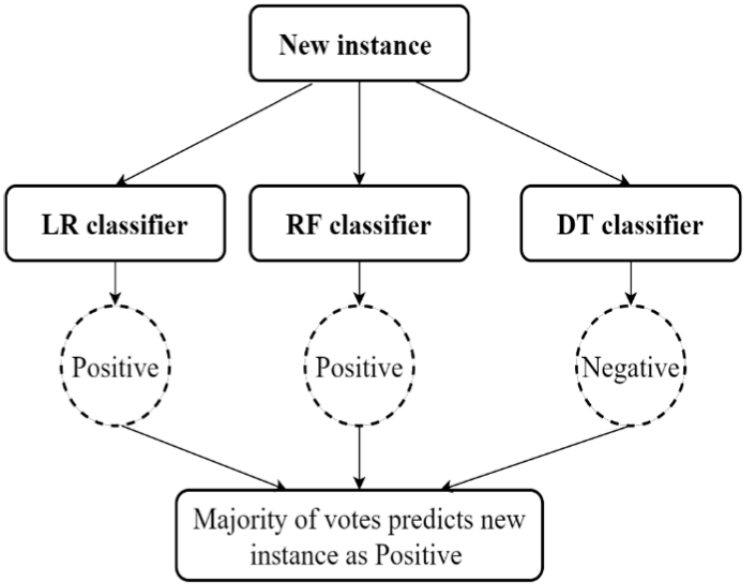
The linear SVM model with $C = 0.5$ achieves 99.64% accuracy on the train set, and 88.78% on the test set.

3.2 Voting

The intuition behind Voting Classifier is based on the Law of Large number. Suppose we toss a fair coin 4 times, then it is impossible to be sure that we will receive exactly 2 heads and 2 tails, but maybe 1 heads and 3 tails i.e. 25% of chance. By the Law of Large number, as we keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (which is 50%). Voting Classifier follows the same fashion, where we can achieve a higher accuracy than the best classifier in the ensemble provided there are a sufficient number of weak learners and they are sufficiently diverse. One way to improve the diversity is by combining multiple base classifiers in the Voting Classifier.

The maximum voting technique is quite famous in decision making that is implemented to get best voted predicted class by all the algorithms. For this technique, all of the results generated by above three algorithms is gathered and then take mod of the predicted class of each document.

Hard voting obtains an overall class prediction based on majority of predictions of ML algorithms in the ensemble.



In this project, we propose three base classifiers: Logistic Regression (LR), Decision Trees (DT) and Random Forest (RF). (2).

Logistic Regression (LR) Logistic regression is a process of modeling the probability of a discrete outcome given an input variable. The most common logistic regression models a binary outcome; something that can take two values such as true/false, yes/no, and so on. Multinomial logistic regression can model scenarios where there are more than two possible discrete outcomes. Logistic regression is a useful analysis method for classification problems, where you are trying to determine if a new sample fits best into a category (4).

LR uses a Logistic function in estimation of positive or negative class labels denoted by y for features of data w: $p(y = \pm 1|x, w) = \frac{1}{1+e^{-w^T h(x)}}$. After tuning using Grid Search with 10-fold cross validation, we yield that C makes the most significant impact on the accuracy, with the optimal value of 2. The accuracy of the LR model (with $C = 2$) evaluated on the train and test set are 95.74% and 88.32% respectively.

Decision Trees (DT) Decision Trees are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. Previous works have demonstrated that decision trees (DT) are very effective for ensemble modeling (Bauer & Kohavi, 1999; Breiman, 1996; Drucker & Cortes, 1995; Freund & Schapire, 1996; Kim, 2018; Kim & Upneja, 2014; Margineantu & Dietterich, 1997; Quinlan, 1996; Tamon & Xiang, 2000; Zhang, Burer, & Street, 2006) (3). The Decision Tree is also tuned to yield *entropy* for the parameter `criterion` for the best accuracy, which is 100% and 85.2% for the train and test set.

Random Forest (RF) RF grows multiple classifier trees with bootstrap sampling with replacement to train each tree with a different part of the dataset. We tried tuning `n_estimators`, `max_depth` to increase the accuracy of the model. However, the tuning does not seem to help much, thus we kept default values for these. The `criterion` hyper-parameter is tuned to yield *entropy* as the best. The accuracy on the train and test set of RF is 100% and 85.12% respectively.

The accuracy for the Voting Classifier with three base classifiers: LR, DT and RF on the test set is 86.932%. The model unfortunately did not outperform Logistic Regression classifier. The reason may lies in the fact that Logistic Regression itself is a quite good classifier, with a significantly higher accuracy compared to Random Forest and Decision Tree. The aggregation of these 3 models did not take full advantage of the Logistic Regression classifier, thus weakens the overall accuracy of the model in the Voting classifier.

Algorithms	Test set accuracy(%)
Logistic Regression	88.32
Random Forest	85.12
Decision Tree	85.20
Hard Voting Classifier	86.93

Table 1: Algorithms in Hard Voting Ensemble Summary

3.3 XGBoost

Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. Concretely, this method tries to fit the new predictor to the residual errors made by the previous predictor. In other words, the model constructs an algorithm to tackle the following optimization problem:

$min_{c_n=1:N, w_n=1:N} L(y, \sum_{n=1}^{N} c_n w_n))$

In which:

- L: loss function
- y: label
- c_n: weight of the *n*-th weak leaner
- w_n: *n*-th weak learner

The Boosting algorithm will try to find the local optimum after each predictor is added to the ensemble, thus gradually converging to the global optimal solution.

Let us consider the formula for Gradient Descent: $\theta_n = \theta_{n-1} - \eta \frac{\partial}{\partial w} L(W_{n-1})$

If we assume a sequence of model boosting as a function W, then each learner can be considered a parameter w. We can minimize the loss function $L(y, W)$ by Gradient Descent as follows:

$W_n = W_{n-1} - \eta \frac{\partial}{\partial w} L(W_{n-1})$

Therefore, it is evident that $c_n w_n \approx -\eta \frac{\partial}{\partial w} L(W_{n-1})$ (where w_n is the next model to be added in the sequence.

As a result, we can see that the new model will learn to fit the term $-\eta \frac{\partial}{\partial w} L(W_{n-1})$, which is also known as the *pseudo-residuals*.

In summary, the procedure of a Gradient Boosting Algorithm is as follows:

1. Initialize the pseudo-residuals, which is the same for every data point.
2. For each iteration i:
    - Train the new model to fit the pseudo-residuals value
    - Find the weight c_i of the new model
    - Update the main model: $W = W + c_i * w_i$
    - Calculate the new pseudo-residuals value for the next model ($-\eta \frac{\partial}{\partial w} L(W_{n_1})$)

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. XGBoost stands for eXtreme Gradient Boosting.

The initial accuracy on test set of XGBoost (with default value for parameters) is 80.676%. After trying tuning `learning_rate`, `max_depth`, `n_estimator`, `gamma`, and `subsample`, we found that

`learning_rate` and `max_depth` have the most significant impacts on the accuracy of the XGBoost classifier. After narrowing down the scope and range of parameters, we use GridSearch with 10-fold CV to find the best combination.

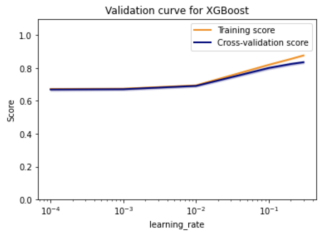


Figure 10: `learning_rate` from 0 to 0.1

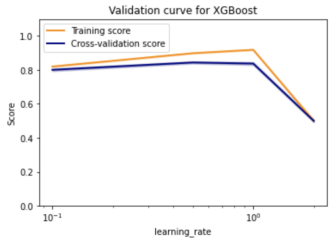


Figure 11: `learning_rate from 0 to 2`

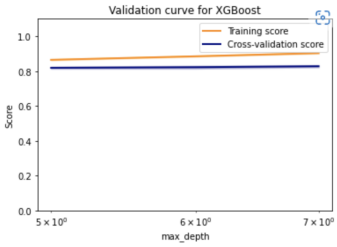


Figure 12: `max_depth` from 5 to 7

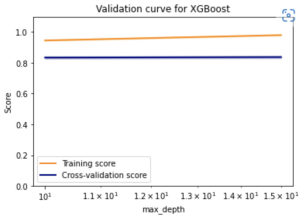


Figure 13: `max_depth` from 10 to 15

The best combination (`learning_rate` $= 0.5$ and `max_depth` $= 7$) yields the accuracy of 97.91% for the train set and 84.628% for the test set.

4 Deep Learning

4.1 Pretrained Word2Vec x Mini-batch KMeans

Cleaning and tokenizing data: this usually involves lowercasing text, removing non-alphanumeric characters, or stemming words.

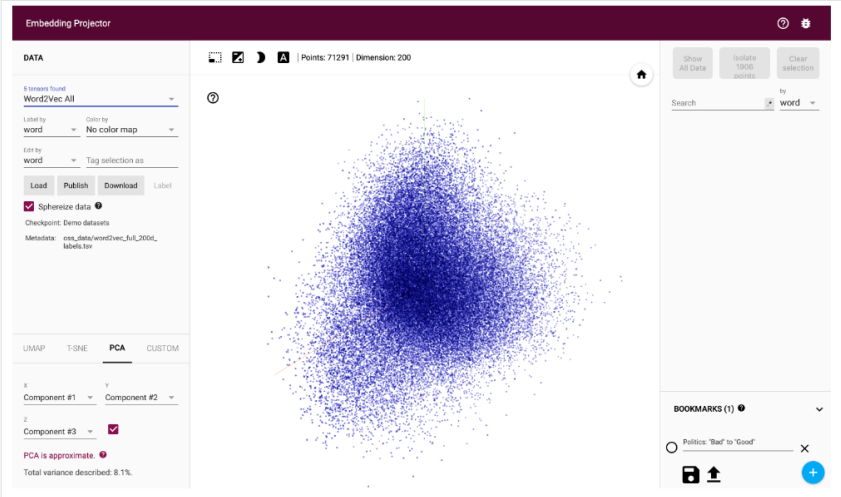
Generating vector representations of the documents: this concerns the mapping of documents from words into numerical vectors—some common ways of doing this include using bag-of-words models or word embeddings. In this repository we used pre-trained fasttext-wiki-news-subwords-300 mix IMDb data to generate word embeddings.

Applying a clustering algorithm on the document vectors: this requires selecting and applying a clustering algorithm to find the best possible groups using the document vectors. Some frequently used algorithms include K-means, DBSCAN, or Hierarchical Clustering.

We use the MiniBatchKMeans is a variant of the KMeans algorithm which uses mini-batches to reduce the computation time, while still attempting to optimise the same objective function. Mini-batches are subsets of the input data, randomly sampled in each training iteration. These mini-batches drastically reduce the amount of computation required to converge to a local solution. In contrast to other algorithms that reduce the convergence time of k-means, mini-batch k-means produces results that are generally only slightly worse than the standard algorithm.

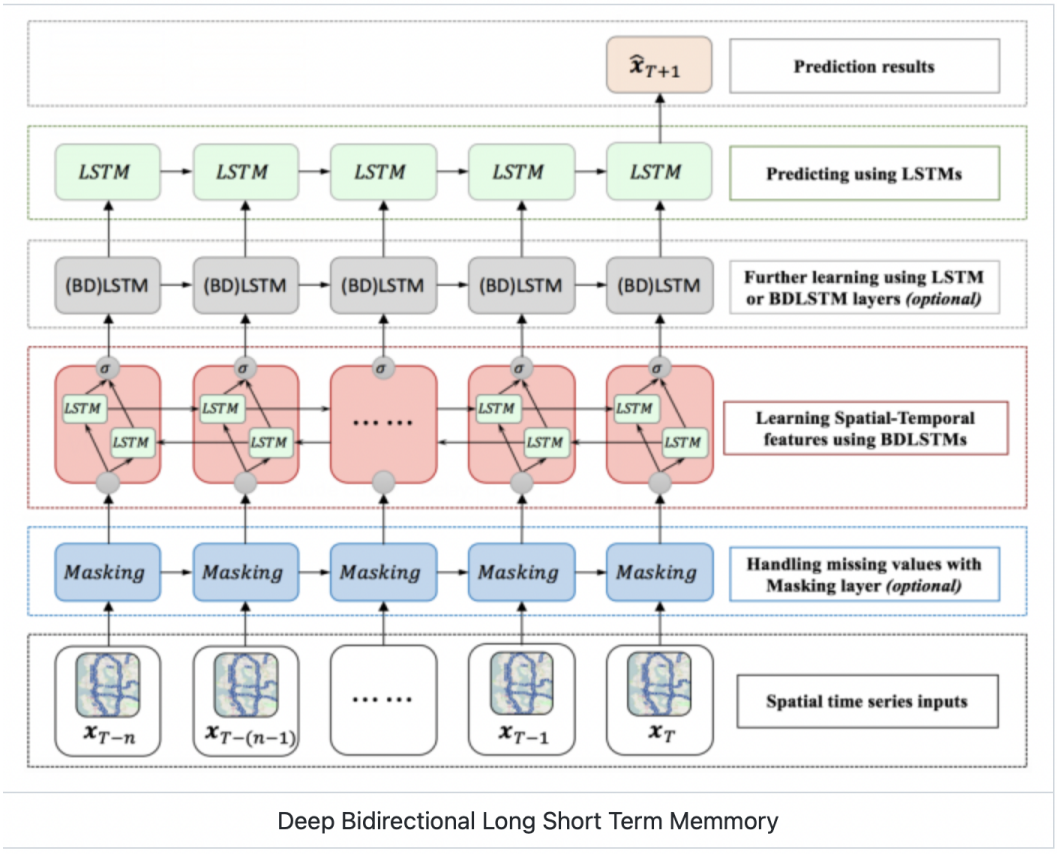
The algorithm iterates between two major steps, similar to vanilla k-means. In the first step, samples are drawn randomly from the dataset, to form a mini-batch. These are then assigned to the nearest centroid. In the second step, the centroids are updated. In contrast to k-means, this is done on a per-sample basis. For each sample in the mini-batch, the assigned centroid is updated by taking the streaming average of the sample and all previous samples assigned to that centroid. This has the effect of decreasing the rate of change for a centroid over time. These steps are performed until convergence or a predetermined number of iterations is reached.

MiniBatchKMeans converges faster than KMeans, but the quality of the results is reduced. In practice this difference in quality can be quite small



Word2Vec

4.2 Deep Bi-directional Network



Deep Bidirectional Long Short Term Memmory

4.2.1 Long Short-Term Memory (LSTM)

The challenge to address long-term information preservation and short-term input skipping in latent variable models has existed for a long time. One of the earliest approaches to address this was the long short-term memory (LSTM) [Hochreiter Schmidhuber, 1997]. It shares many of the properties of the GRU. Interestingly, LSTMs have a slightly more complex design than GRUs but predates GRUs by almost two decades.

4.2.1.1 Gated Memory Cell

Arguably LSTM's design is inspired by logic gates of a computer. LSTM introduces a memory cell (or cell for short) that has the same shape as the hidden state (some literatures consider the memory cell as a special type of the hidden state), engineered to record additional information. To control the memory cell we need a number of gates. One gate is needed to read out the entries from the cell. We will refer to this as the output gate. A second gate is needed to decide when to read data into the cell. We refer to this as the input gate. Last, we need a mechanism to reset the content of the cell, governed by a forget gate. The motivation for such a design is the same as that of GRUs, namely to be able to decide when to remember and when to ignore inputs in the hidden state via a dedicated mechanism. Let's see how this works in practice.

4.2.1.1.1 Input Gate, Forget Gate, and Output Gate

Just like in GRUs, the data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Fig. 12. They are processed by three fully connected layers with a sigmoid activation function to compute the values of the input, forget, and output gates. As a result, values of the three gates are in the range of **(0, 1)**.

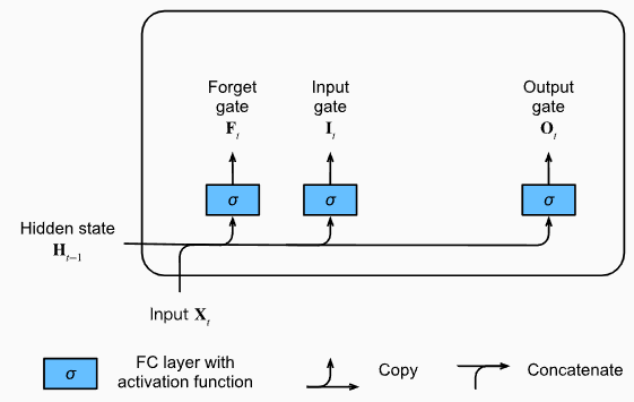


Figure 14: Computing the input gate, the forget gate, and the output gate in an LSTM model.

Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n \times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n \times h}$. They are calculated as follows:

$$\begin{aligned}\mathbf{I}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i\right)\\ \mathbf{F}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f\right)\\ \mathbf{O}_t &= \sigma\left(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o\right)\end{aligned}$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ are bias parameters.

4.2.1.1.2 Candidate Memory Cell

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the candidate memory cell $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$. Its computation is similar to that of the three gates described above, but using a tanh function with a value range for $(-1, 1)$ as the activation

function. This leads to the following equation at time step t :

$$\tilde{\mathbf{C}}_t = \tanh\left(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c\right),$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter. A quick illustration of the candidate memory cell is shown in Fig. 13.

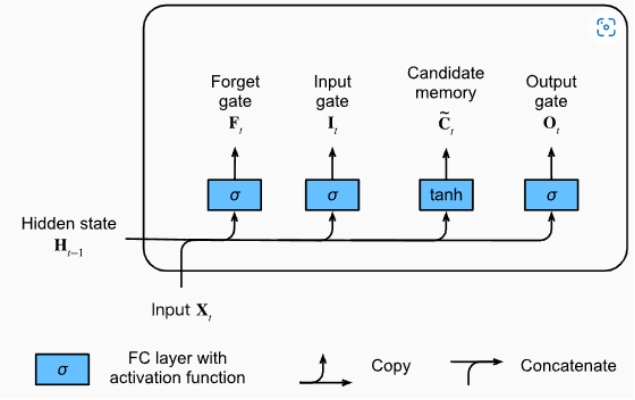


Figure 15: Computing the candidate memory cell in an LSTM model.¶

4.2.1.1.3 Memory Cell

In GRUs, we have a mechanism to govern input and forgetting (or skipping). Similarly, in LSTMs we have two dedicated gates for such purposes: the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget gate $\mathbf{F}_t$ addresses how much of the old memory cell content $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain. Using the same pointwise multiplication trick as before, we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If the forget gate is always approximately 1 and the input gate is always approximately 0 , the past memory cells $\mathbf{C}_{t-1}$ will be saved over time and passed to the current time step. This design is introduced to alleviate the vanishing gradient problem and to better capture long range dependencies within sequences. We thus arrive at the flow diagram in Fig. 14.

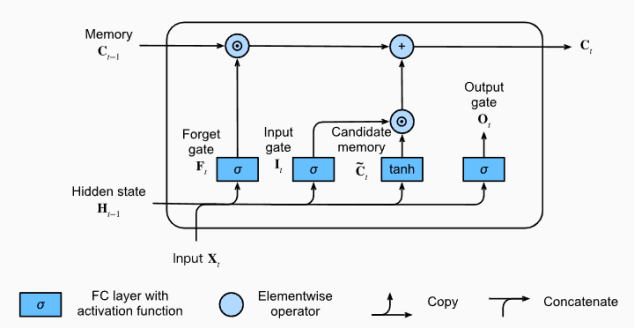


Figure 16: Computing the memory cell in an LSTM model.

4.2.1.1.4 Hidden State

Last, we need to define how to compute the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. This is where the output gate comes into play. In LSTM it is simply a gated version of the tanh of the memory cell. This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh\left(\mathbf{C}_t\right).$$

Whenever the output gate approximates 1 we effectively pass all memory information through to the predictor, whereas for the output gate close to 0 we retain all the information only within the memory cell and perform no further processing. Fig. 15 has a graphical illustration of the data flow.

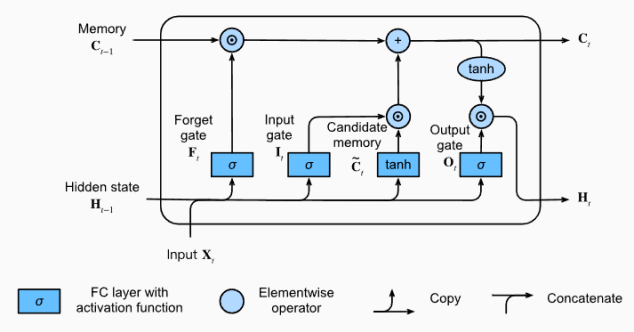


Figure 17: Computing the hidden state in an LSTM model.

4.2.2 Gated Recurrent Units (GRU)

Bidirectional RNNs were introduced by [Schuster & Paliwal,1997]. For a detailed discussion of the various architectures see also the paper [Graves & Schmidhuber, 2005]. Let's look at the specifics of such a network.

For any time step t, given a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d) and let the hidden layer activation function be ϕ. In the bidirectional architecture, we assume that the forward and backward hidden states for this time step are $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ and $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$, respectively, where h is the number of hidden units. The forward and backward hidden state updates are as follows:

$$\overrightarrow{\mathbf{H}}_t = \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}\right)$$
$$\overleftarrow{\mathbf{H}}_t = \phi\left(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}\right)$$

where the weights $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$, $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, and biases $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all the model parameters. Next, we concatenate the forward and backward hidden states $\overrightarrow{\mathbf{H}}_t$ and $\overleftarrow{\mathbf{H}}_t$ to obtain the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ to be fed into the output layer. In deep bidirectional RNNs with multiple hidden layers, such information is passed on as input to the next bidirectional layer. Last, the output layer computes the output $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q):

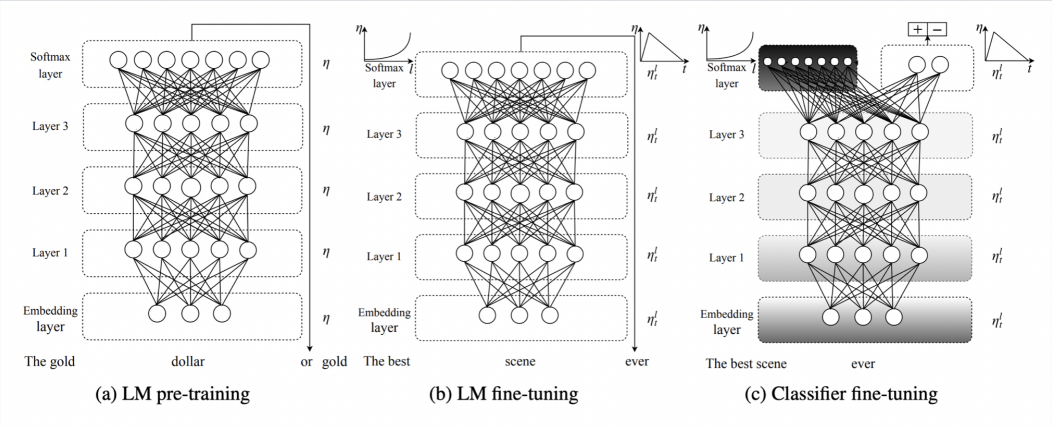
$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q.$$

Here, the weight matrix $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. In fact, the two directions can have different numbers of hidden units.

4.2.3 Bidirectional

Researchers have developed more sophisticated types of RNN to deal with the shortcomings of the standard RNN model: Bidirectional RNN, Deep Bidirectional RNN and Long Short Term Memory network. Bidirectional RNN is based on the idea that the output at each time may not only depend on the previous elements in the sequence, but also depend on the next elements in the sequence. For instance, to predict a missing word in a sequence, we may need to look at both the left and the right context. A bidirectional RNN24 consists of two RNNs, which are stacked on the top of each other. The one that processes the input in its original order and the one that processes the reversed input sequence. The output is then computed based on the hidden state of both RNNs. Deep bidirectional RNN is similar to bidirectional RNN. The only difference is that it has multiple layers per time step, which provides higher learning capacity but needs a lot of training data

4.3 Universal Language Model Fine-tuning



Howard et al. in Universal Language Model Fine-tuning for Text Classification

We are interested in the most general inductive transfer learning setting for NLP: Given a static source task TS and any target task T_T with $T_S \neq T_T$, we would like to improve performance on T_T . Language modeling can be seen as the ideal source task and a counter-part of ImageNet for NLP: It captures many facets of language relevant for downstream tasks, such as long-term dependencies, hierarchical relations, and sentiment . In contrast to tasks like MT nd entailment, it provides data in near-unlimited quantities for most domains and languages. Additionally, a pretraid LM can be easily adapted to the idiosyncrasies of a target task, which we show significantly improves performance . Moreover, language modeling already is a key component of existing tasks such as MT and dialogue modeling. Formally, language modeling induces a hypothesis space H that should be useful for many other NLP tasks.

We propose Universal Language Model Fine- tuning (ULMFiT), which pretrains a language model (LM) on a large general-domain corpus and fine-tunes it on the target task using novel tech- niques. The method is universal in the sense that it meets these practical criteria: 1) It works across tasks varying in document size, number, and label type; 2) it uses a single architecture and training process; 3) it requires no custom feature engineering or preprocessing; and 4) it does not require additional indomain documents or labels.

In our experiments, we use the state-of-the-art language model AWD-LSTM, a regular LSTM (with no attention, short-cut connections, or other sophisticated additions) with various tuned dropout hyperparameters. Analogous to CV, we expect that downstream performance can be improved by using higher- performance language models in the future.

4.3.1 General-domain LM pretraining

An ImageNet-like corpus for language should be large and capture general properties of language. We pretrain the language model on Wikitext-103 consisting of 28,595 preprocessed Wikipedia articles and 103 million words. Pretraining is most beneficial for tasks with small datasets and enables generalization even with 100 labeled examples. We leave the exploration of more diverse pretraining corpora to future work, but expect that they would boost performance. While this stage is the most expensive, it only needs to be performed once and improves performance and convergence of downstream models.

4.3.2 Target task LM fine-tuning

No matter how diverse the general-domain data used for pretraining is, the data of the target task will likely come from a different distribution. We thus fine-tune the LM on data of the target task. Given a pretrained general-domain LM, this stage converges faster as it only needs to adapt to the

idiosyncrasies of the target data, and it allows us to train a robust LM even for small datasets. We propose discriminative fine-tuning and slanted triangular learning rates for fine-tuning the LM, which we introduce in the following.

Discriminative fine-tuning As different layers capture different types of information, they should be fine-tuned to different extents. To this end, we propose a novel fine-tuning method, discriminative fine-tuning. Instead of using the same learning rate for all layers of the model, discriminative fine-tuning allows us to tune each layer with different learning rates. For context, the regular stochastic gradient descent (SGD) update of a model's parameters θ at time step t looks like the following:

$$\theta_n = \theta_{n-1} - \eta \nabla J(\theta)$$

where η is the learning rate and $\nabla J(\theta)$ is the gradient with regard to the model's objective function. For discriminative fine-tuning, we split the parameters θ into θ^1, . . . , θ^l where l contains the parameters of the model at the l-th layer and L is the number of layers of the model. Similarly, we obtain η^1,...,η^l where η^l is the learning rate of the l-th layer. The SGD update with discriminative fine- tuning is then the following:

We empirically found it to work well to first choose the learning rate L of the last layer by fine-tuning only the last layer and using θ^l -1 = θ^l /2.6 as the learning rate for lower layers.

Slanted triangular learning rates For adapting its parameters to task-specific features, we would like the model to quickly converge to a suitable region of the parameter space in the beginning of training and then refine its parameters. Using the same learning rate (LR) or an annealed learning rate throughout training is not the best way to achieve this behaviour. Instead, we propose slanted triangular learning rates (STLR), which first linearly increases the learning rate and then linearly decays it according to the following up- date schedule, which can be seen in Figure below:

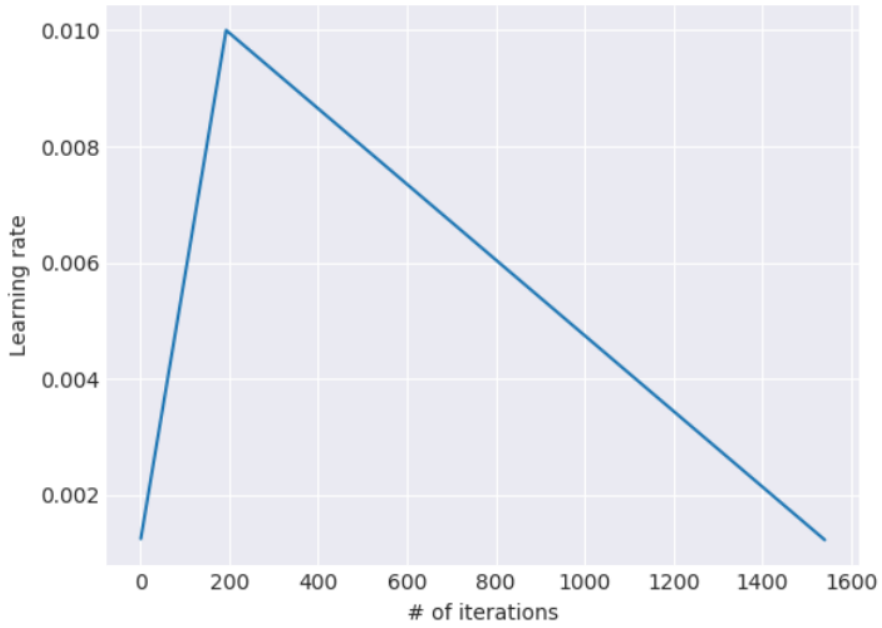
$$cut = \left[T cut\_frac \right]$$

$$p = \begin{cases} t/cut & t < cut \\ 1 - \frac{t - cut}{cut(1/cut\_frac - 1)} & otherwise \end{cases}$$

$$\eta_t = \eta_{max} . \frac{1 + p(ratio - 1)}{ratio}$$

where T is the number of training iterations, cut_frac is the fraction of iterations we increase the LR, cut is the iteration when we switch from increasing to decreasing the LR, p is the fraction of the number of iterations we have increased or will decrease the LR respectively, ratio specifies how much smaller the lowest LR is from the maximum LR max, and t is the learning rate at iteration t. We generally use cut_frac = 0.1, $ratio$ = 32 and η_{max} = 0.01.

STLR modifies triangular learning rates with a short increase and a long decay period, which we found key for good performance



The slanted triangular learning rate schedule used for ULMFiT as a function of the number of training iterations

4.3.3 Target task classifier fine-tuning

Finally, for fine-tuning the classifier, we augment the pretrained language model with two additional linear blocks. Following standard practice for CV classifiers, each block uses batch normalization and dropout, with ReLU activations for the intermediate layer and a softmax activation that outputs a probability distribution over target classes at the last layer. Note that the parameters in these task-specific classifier layers are the only ones that are learned from scratch. The first linear layer takes as the input the pooled last hidden layer states.

Concat pooling The signal in text classification tasks is often contained in a few words, which may occur anywhere in the document. As input docu- ments can consist of hundreds of words, information may get lost if we only consider the last hid- den state of the model. For this reason, we concatenate the hidden state at the last time step hT of the document with both the max-pooled and the mean-pooled representation of the hidden states over as many time steps as fit in GPU memory H = h_1,...,h_T:

$$hc = [hT, maxpool(H), meanpool(H)]$$

where [] is concatenation.
Fine-tuning the target classifier is the most crit- ical part of the transfer learning method. Overly aggressive fine-tuning will cause catastrophic for- getting, eliminating the benefit of the information captured through language modeling; too cautious fine-tuning will lead to slow convergence (and resultant overfitting). Besides discriminative fine- tuning and triangular learning rates, we propose gradual unfreezing for fine-tuning the classifier.

Gradual unfreezing Rather than fine-tuning all layers at once, which risks catastrophic forgetting, we propose to gradually unfreeze the model starting from the last layer as this contains the least general knowledge: We first unfreeze the last layer and fine-tune all un- frozen layers for one epoch. We then unfreeze the next lower frozen layer and repeat, until we fine- tune all layers until convergence at the last itera- tion. This is similar to 'chain-thaw', except that we add a layer at a time to the set of 'thawed' layers, rather than only training a single layer at a time.

While discriminative fine-tuning, slanted trian- gular learning rates, and gradual unfreezing all are beneficial on their own, we show in Section 5 that they complement each other and enable our method to perform well across diverse datasets.

BPTT for Text Classification (BPT3C) Language models are trained with backpropagation through time (BPTT) to enable gradient propagation for large input sequences. In order to make fine-tuning a classifier for large documents feasible, we propose BPTT for Text Classification (BPT3C): We divide the document into fixed- length batches of size b. At the beginning of each batch, the model is initialized with the final state of the previous batch; we keep track of the hid- den states for mean and max-pooling; gradients are back-propagated to the batches whose hidden states contributed to the final prediction. In practice, we use variable length backpropagation sequences. Bidirectional

Dataset	Type	# classes	# examples
TREC-6	Question	6	5.5k
IMDb	Sentiment	2	25k
Yelp-bi	Sentiment	2	560k
Yelp-full	Sentiment	5	650k
AG	Topic	4	120k
DBpedia	Topic	14	560k

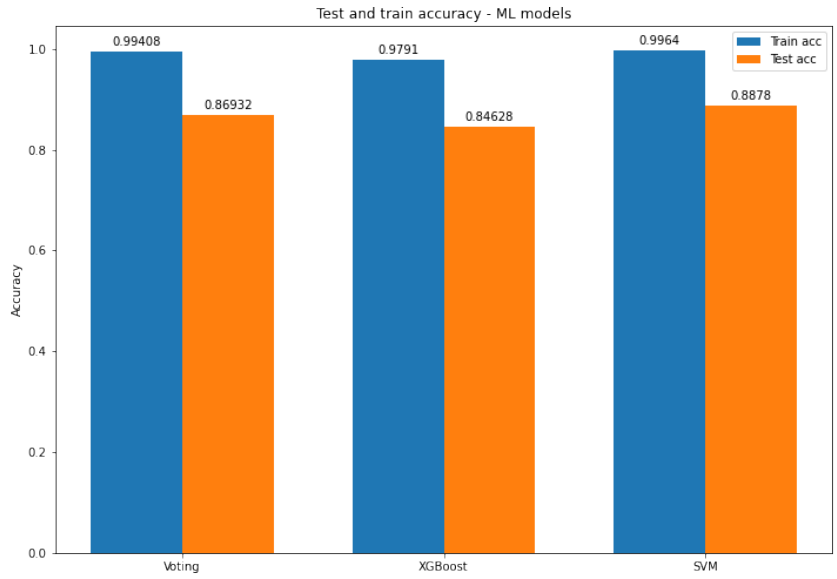
Text classification datasets and tasks with number of classes and training examples.

language model Similar to existing work, we are not limited to fine-tuning a unidirectional language model. For all our experiments, we pretrain both a forward and a backward LM. We fine-tune a classifier for each LM independently using BPT3C and average the classifier predictions.

5 Evaluation

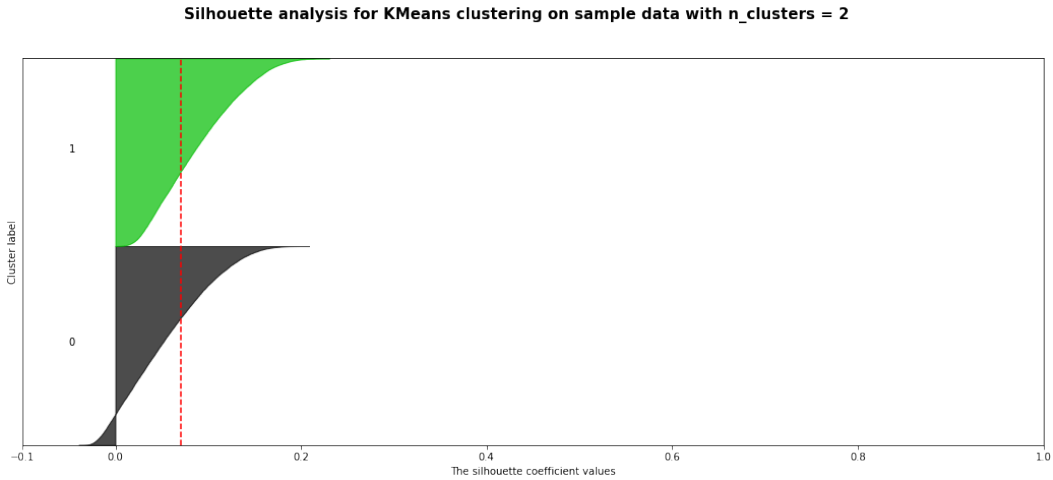
5.1 Machine Learning

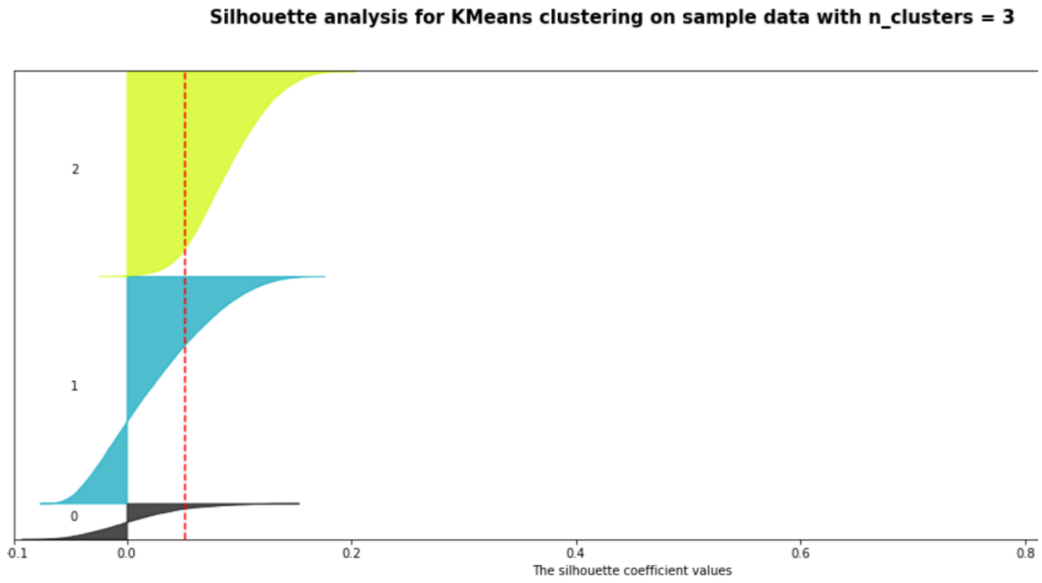
We chose **accuracy** as the metric to measure the performance of our Supervised Machine Learning algorithms. Below is the plot of training and test accuracy of Machine Learning models. It can be seen that all three models achieved quite good accuracy of about 88 percent in average.



Silhouette Score is a metric to evaluate the performance of clustering algorithm. It uses compactness of individual clusters(intra cluster distance) and separation amongst clusters (inter cluster distance) to measure an overall representative score of how well our clustering algorithm has performed.

Silhouette coefficients near +1 indicate that the sample is far away from the neighboring clusters. A value of 0 indicates that the sample is on or very close to the decision boundary between two neighboring clusters and negative values indicate that those samples might have been assigned to the wrong cluster.



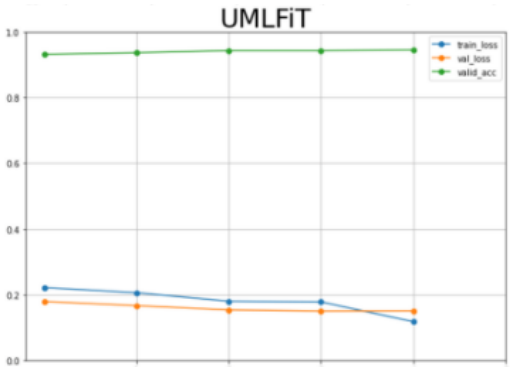


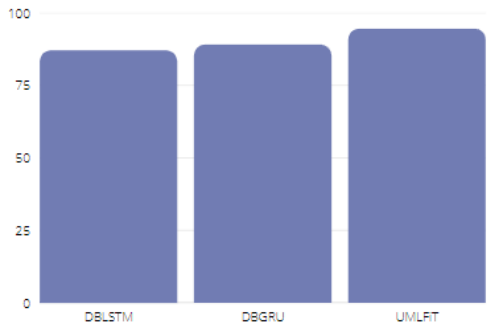
5.2 Deep Learning

5.2.1 DBNet

Accuracy(%)	DB-LSTM (4e)	DB-GRU (2e)
Train	0.9790	0.9724
Test	0.8772	**0.8893**

5.2.2 UMLFiT





6 Conclusion

Humans are naturally dependent of the opinions and experience of others, thus have a high tendency to seek the reviews of products before trying out themselves. However, it would be painful to scour the correct information from a multitude of reviews present on the internet. Based on the data from IMDb dataset, we proposed many different algorithms to find the best sentiment classifier for our

problem. In terms of Machine Learning models, Linear Support Vector Machine has outperformed other algorithms with an accuracy of 88.78%. For Deep Learning models, UMLFitT achieved the highest accuracy of 94.5%.

References

[1] YONATHA WIJAYA, Komang Dhiyo; KARYAWATI, Anak Agung Istri Ngurah Eka. The Effects of Different Kernels in SVM Sentiment Analysis on Mass Social Distancing. JELIKU (Jurnal Elektronik Ilmu Komputer Udayana), [S.l.], v. 9, n. 2, p. 161-168, nov. 2020. ISSN 2654-5101

[2] Akın Özçift. Medical sentiment analysis based on soft voting ensemble algorithm.

[3] Soo YoungKima, ArunUpnejab. Majority voting ensemble with a decision trees for business failure prediction during economic downturns.

[4] Thomas W. Edgar, David O. Manz, in Research Methods for Cyber Security, 2017

Appendix

DBLSTM - Deep Bi-directional LSTM

Lei Zhang, Shuai Wang, Bing Liu. Deep Learning for Sentiment Analysis: A Survey

Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, Jianfeng Gao. Deep Learning Based Text Classification: A Comprehensive Review

Mike Schuster, Kuldip K. Bidirectional Recurrent Neural Networks

Tomas Mikolov, Hya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality

Sepp Hochreiter, Jürgen Schmidhuber. LONG SHORT-TERM MEMORY

Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space

UMLFiT - Universal Language Model Fine-tuning

Stephen Merity, Nitish Shirish Keskar, Richard Socher. Regularizing and Optimizing LSTM Language Models

Jeremy Howard, Sebastian Ruder. Universal Language Model Fine-tuning for Text Classification

Leslie N. Smith. A DISCIPLINED APPROACH TO NEURAL NETWORK HYPER-PARAMETERS: PART 1 – LEARNING RATE, BATCH SIZE, MOMENTUM, AND WEIGHT DECAY