ECE 684 Final Project – Sentiment Analysis

Michael Li (zl310), Chengyang Zhou (cz169)

December 9, 2023

Contents

List of Figures

1 Introduction

Sentiment analysis is a popular field in natural language processing that aims to classify the emotional tone of a given text. With the abundance of user-generated content on the internet, sentiment analysis has become increasingly important for businesses, social media platforms, and other organizations to better understand their customers and users. In this write-up, we will explore the application of sentiment analysis to movie reviews using two different machine learning models: Naive Bayes and Long-Short-Term Memory (LSTM). The Naive Bayes model is a generative probabilistic model that estimates the likelihood of each class given the input features, while LSTM is a discriminative model that learns to directly map inputs to outputs [1, 2].

We will use a truncated version of the IMDB dataset [3] that has 50,000 entries in total as well as a synthetic dataset generated using two fine-tuned GPT2 models [4, 5, 6] to train and evaluate these models. Please note that the models used to generate the synthetic dataset are different from the two models that we chose to explore in depth in this project. Through this project, we aim to compare the performance of these two models on the two sentiment analysis datasets and gain insights into their strengths and weaknesses.

2 Methodology and Background Information

2.1 Naive Bayes

Naive Bayes is a probabilistic algorithm based on Bayes Theorem that is commonly used for classification tasks. It has various applications, such as spam filtering and document classification. The algorithm assumes that the features being classified are independent of each other, which is why it is called "naive". Naive Bayes calculates the probability of a data instance belonging to a particular class based on the probabilities of its features and uses these probabilities to make predictions [7]. Naive Bayes is the first model we consider for sentiment classification.

We followed Dr Sharma's usage of scikit-learn's (sklearn) Naive Bayes and CountVectorizer modules, and ntlk's tokenization function [1, 8, 9]. The dataset is read into a dataframe using pandas, tokenised, converted to a matrix, and split into train and test sets using a seven-to-three train-to-test ratio [10]. Model performance is calculated using the Multinomial Naive Bayes' module [1]. The relationship between train-to-test ratio and performance is explored. This model can be considered as the baseline.

2.1.1 CountVectorizer

CountVectorizer can be used to convert non-numerical data (especially text data) into numerical data. One implementation comes from sklearn [1], which transforms text into a vector format based on the frequency of each token in the given text. The CountVectorizer works by creating a vocabulary of unique words from the text and then representing each document or piece of text as a numerical vector, where each element of the vector corresponds to the count of a specific word in the document. This approach helps in capturing the textual information and enables further analysis and modeling tasks. According to [11], CountVectorizer transforms the text into a sparse matrix, where each unique word represents a different column in the matrix. This is similar to the bag-of-words model, which represents lists of tokens as unordered sets of tokens [12]. The CountVectorizer module implemented by sklearn accounts for stop words, the extraction of n-grams, and tokenization patterns [1].

Instead of another mechanism for generating embeddings used with Naive Bayes, such as tf-idf (implemented as TfidfTransformer in sklearn [1]), we felt that it is sufficient to use a bag-of-words model to generate the embeddings of each input. The sentiment of a review is closely related to certain key adjectives and their presence can often indicate whether a review is positive or negative. Therefore, a more complex model for generating embeddings will simply drive up computation costs without leading to significant increases in accuracy.

2.2 LSTM

LSTMs are a type of recurrent neural network (RNN) that can learn long-term dependencies in a sequence of data [2, 13]. LSTMs are constructed using repeating modules with four interacting layers in instance of a module, which allow the network to selectively remember or forget information over time. The four layers are the input gate, the forget gate, the output gate, and the memory cell. The input gate controls the flow of new information into the cell state, the forget gate controls the flow of previous information out of the cell state, and the output gate controls the output of the LSTM module. The memory cell acts as a memory unit that stores information over long periods of time [13]. The goal of a LSTM module is to figure out how much information about past inputs the module needs to memorise [2]. Figure 1 show the connections between successive LSTM modules and the four different gates with their respective activation functions. X_t is the t^{th} encoder input and h_t is the hypothesis. The memory of the $t-1^{\text{th}}$ LSTM module is fed into the top-left input of the t^{th} LSTM module [13].

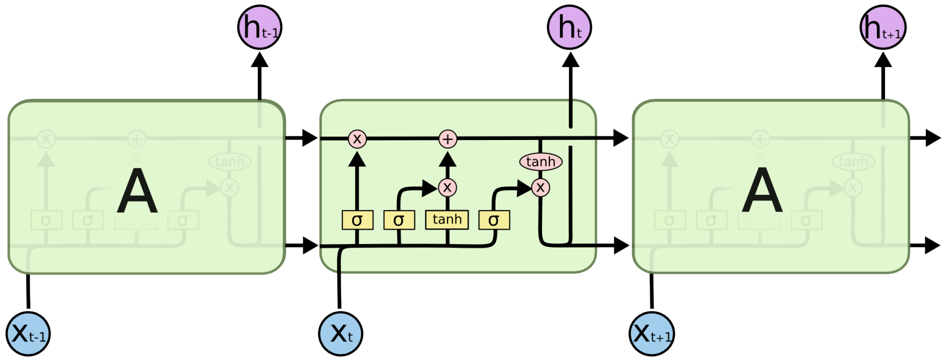


Figure 1: Repeating LSTM modules, sourced from [13].

2.3 Synthetic Data

The synthetic data is produced using two transformers, each of which has been fine-tuned on positive or negative IMDB reviews; the models are downloaded from the Hugging Face platform and the models are interfaced using the Python transformers library [4, 6, 5].
A generative step is as follows, assuming we've read in the original non-synthetic IMDB dataset:

1. Take a random contiguous sequence of tokens with length between 7 and 15 tokens of a random sentence in the original IMDB dataset.
2. Encode the prompt and feed the encodings into the transformer (trained either on negative or positive IMDB data) using the `generate` method with a set of pre-determined hyperparameters to generate a number of distinct outputs in response to the prompt. Hyperparameters for the transformer are tweaked and more detail is given below.
3. Decode the output of the transformer and store the output, i.e. the review, and the sentiment.

Repeat the above steps with different input models to generate more positive and negative reviews. Increasing the number of beams when generating output allows us to generate more than one sequence of outputs (i.e., more than one sentence) from one prompt. Note: positive reviews are only generated using the transformer that's been fine-tuned on positive IMDB reviews; the corresponding concept applies for negative reviews.
We chose to use 1.0 as the temperature. The number of beams, output sequences, and beam groups are the same. According to documentation on Hugging Face concerning transformers [14], "Beam search is a method used that maintains beams (or "multiple hypotheses") at each step, expanding each one and keeping the top-scoring sequences." Using group beam search means that the hypotheses inside of each beam group are independent from other beam groups. The `repetition_penalty` parameter is 20.0 (the higher it is, the less repetition within the same output sequence). The `diversity_penalty` is set as 2.5 (the higher it is, the more diverse the outputs of each beam group) [14]. A consequence of setting number of beams, number of output sequences, and number of beam groups all equal to each other is that one beam group contains one beam in charge of generating one output.

2.4 Overview of experiments

In the first part of the experiments, we present the performance of the Naive Bayes classifier trained on real and synthetic datasets with different train-to-test ratios. In the second part, using the LSTM model, we perform hyperparameter tuning using the non-synthetic IMDB dataset to find a model configuration that maximises validation accuracy on the IMDB dataset. With the most performant set of hyperparameters, the LSTM model is trained and tested using the synthetic dataset.

3 Experimental Results

First, we consider the experimental results using the real-life dataset. Hyperparameter settings for the LSTM model is tuned using the real-life dataset. Then, we train and test the models on the synthetic dataset.

3.1 Results using the Real-life Dataset

3.1.1 Naive Bayes

To test the performance of the Naive Bayes classifier implemented using sklearn's `MultinomialNB` module [1], we compute the training accuracy using different train-test splits. The fraction of total data used as testing data is varied from 0.05 to 0.95 with 0.05 increments. According to Figure 2, when more data is used as testing data, the performance decreases. There is no precipitous decrease in the test accuracy even when the amount of training data becomes much smaller than the amount of test data (note that the y-axis goes from around 0.8 to around 0.9). This is because the Naive Bayes algorithm can perform well with limited training data due to its simplicity and the assumption of independence between features, causing over-fitting to be less likely [15].

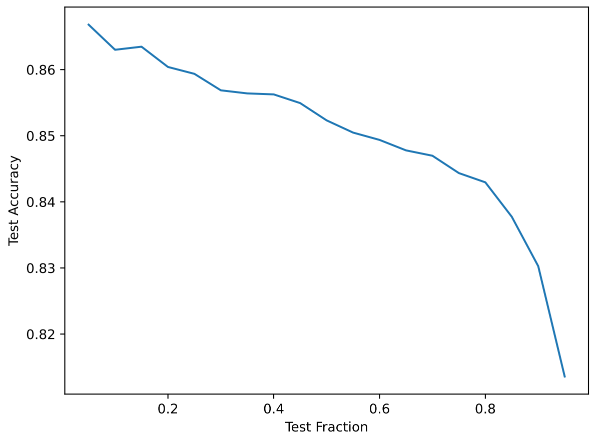


Figure 2: Test accuracy of the Naive Bayes classifier vs the fraction of total data used as testing data.

However, the performance does decrease as expected. With fewer training instances, the estimated probabilities in Naive Bayes can become more variable, leading to less reliable predictions. Additionally, having less training data can result in a limited representation of the true underlying distribution of the features, which may introduce bias and affect the accuracy of the classifier [16]. The Naive Bayes classifier is quite robust even when trained with a very small amount of data, since having 5% of the entire dataset as the training set (around 2,500 (X, y) pairs) is enough to make the classifier perform at around 0.81 accuracy. The accuracy of the Naive Bayes classifier is 0.8569 when 70% of the data is used for training.

3.1.2 LSTM Setup and Data Processing

The raw data read from the csv file is first split into train, test, and validation sets (7 : 2 : 1). Then, the vocabulary dictionary is generated using the train set. Each mini-batch is converted to a list of numerical vectors during training.
The LSTM model is implemented using PyTorch using the `Embedding`, `LSTM`, `Dropout`, and `Linear` modules [17]. The `Embedding` module takes as input a mini-batch time-series vectors, each of which represents one input review. Each numerical input vector is generated using the vocabulary dictionary that maps tokens to integers. For example, if $\vec{v} \in \mathbb{R}^{l \times 1} \forall \{0 < l \leq 256, l \in \mathbb{Z}\}$ represents one review, v_t, the t^{th} element in $\vec{v}$, contains the mapping of the t^{th} token in the vocabulary. The vocabulary is generated using the training data and considers only tokens that appear more than five times; each review is converted to lower case and then tokenised by splitting the input string according to spaces only. Note that ***unknown tokens are discarded***; also note that in this project, the **tokenisation process** for the LSTM is **different** than that for the Naive Bayes classifier. If the vocabulary consists of {`‘word0’: 0, ‘word1’: 1, ‘word2’: 2, ‘word3’: 3`}, then the vector $\vec{v}$ representing the sentence ‘word0 word3 word3 word0 oov1 word1 oov2’ would be $\vec{v} = (0\ 3\ 3\ 0\ 1)$. The trainable `Embedding` module outputs the embeddings of the input that are better suited for the LSTM model.
The `LSTM` module takes a mini-batch of embeddings as input, and the value we’re interested for each input vector in the mini-batch is the last hidden state of the output as a vector. Dropout is applied on the last hidden state vectors. Then, a fully-connected layer down-samples the hidden state vector into a 2×1 vector, which contains the raw logits.

Let us define the default set of hyperparameters, shown below:

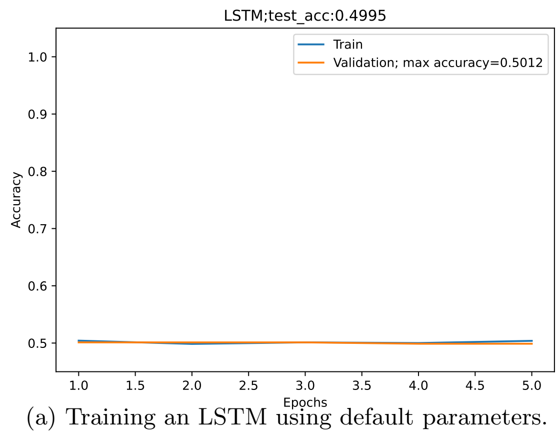
```
class HyperParams:
    def __init__(self):
        self.PAD_INDEX = 0  # used for padding sentences
        self.MAX_LENGTH = 256  # number of tokens of input sentences; longer sentences are truncated,
        # shorter sentences are padded
        self.BATCH_SIZE = 96  # batch size
        self.EMBEDDING_DIM = 1  # dimensions of each embedding vector, which is then fed into the LSTM
        self.HIDDEN_DIM = 100  # dimensions of the hidden state in the LSTM
        self.OUTPUT_DIM = 2  # dimensions of the LSTM predictions (fixed, since there are only two output classes)
        self.N_LAYERS = 1  # number of layers in the LSTM model
        self.DROPOUT_RATE = 0.0  # dropout rate of all layers (except for the output layer) of the LSTM
        self.LR = 0.01  # learning rate
        self.N_EPOCHS = 5  # number of epochs for training
        self.WD = 0  # weight decay
        self.OPTIM = "sgd"  # optimiser
        self.BIDIRECTIONAL = False  # we consider only the uni-directional LSTM in this project
```

Results produced using hyperparameters that deviate from the default set will specify the hyperparameters used. However, since the SGD optimiser with 0.01 as the learning rate does not perform well as shown in later sections, assume that the RMSProp optimiser with 0.001 as the learning rate is used in most discussions below.

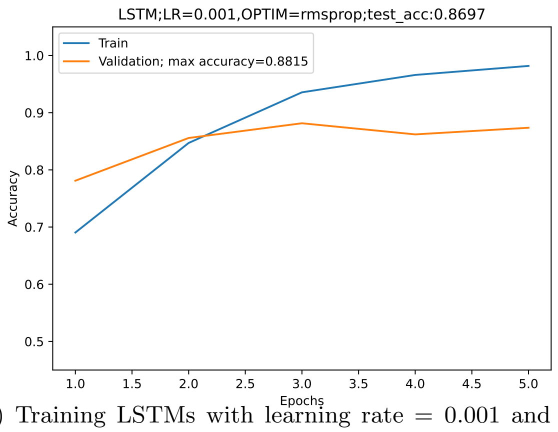
3.1.3 LSTM Hyperparameters Search

We consider the classification results using default hyperparameters as specified on the previous page, shown in Figure 3a. Mainly, we will vary model **depth**, dimensions for the **hidden state**, and dimensions for the **embeddings**. However, it is important to first consider the choice of optimiser.

In general, optimisers such as RMSProp or Adagrad are usually much better choices than the SGD optimiser [17, 18]. Visualisations in [18] (specifically, Images 5-6) show that the model using the SGD optimiser travels at a much slower speed towards minima. In contrast, the Adagrad and RMSProp optimisers are able to lead the model to the local minimum very quickly. As we can see in Figure 3b, the model’s performance using the RMSProp optimiser is much better than the one using the SGD optimiser. Therefore, empirically, we can say that the SGD optimiser does not work for training LSTMs. From this point onwards, we will be using the RMSProp optimiser instead of the SGD optimiser with 0.001 as the learning rate.



(a) Training an LSTM using default parameters.



(b) Training LSTMs with learning rate = 0.001 and the RMSProp optimiser.

Figure 3: Comparison between the performance using the default hyperparameter setting and using an improved set of hyperparameters.

Table 1: Accuracy values during the last epoch of the LSTM model using different **model depth** (RMSProp, $lr = 0.001$).

Number of Layers	1 (default)	2	3	4
Max Validation Accuracy	0.8815	0.8862	0.5005	0.5012
Last-Epoch Test Accuracy	0.8697	0.8659	0.5014	0.5005

Table 1 shows the changes in test/validation accuracy when varying the model depth as represented by the number of layers in the LSTM. We see that the model accuracy becomes random-guessing accuracy when we use more than 2 layers in the model. This is likely because the model is too complex in one respect (number of layers) and not complex enough in other respects such as the dimensions of the hidden state and the embeddings. We also see a slight decrease in accuracy as the model trains for more epochs when there are 1 or 2 layers. This is evinced by the fact that the maximum validation accuracy is higher than the test accuracy after training for five epochs. Since the model is relatively simple with only 1 or 2 layers, this observation could be due to over-fitting after two or three epochs. Table 1 alone does not conclusively indicate that deeper models are worse, however, since deeper models with higher dimensions for hidden state and embeddings may perform perform well.

Table 2: Maximum and last-epoch accuracy values, and number of trainable parameters for the LSTM model with various dimensions for the **hidden states** (RMSProp, $lr = 0.001$).

Dimension	10	20	100 (default)	150	200	260	285	319
Max Validation Accuracy	0.8778	0.8732	0.8815	0.8608	0.8819	0.8726	0.8878	0.5804
Last-Epoch Test Accuracy	0.8734	0.8669	0.8697	0.8579	0.8542	0.8720	0.8781	0.5804
Number of Parameters	57k	59k	98k	149k	220k	331k	386k	468k

Table 2 shows the effects of using different sizes for the hidden state. As the size of the hidden dimension increases, the model performance stays roughly constant; the maximum validation accuracy is generally higher than the last-epoch test accuracy. When the hidden state is of very high dimensionality, the model trains very slowly and reaches a low test accuracy after the fifth epoch. This is because the model is too complex for training to take place in just five epochs. There is no trade-off between model performance and the model size: even when the number of trainable parameters is low (small size for the hidden state), the model is very performant. Therefore, the dimensionality of the hidden state does not necessarily have to be high for the model to perform well.

Table 3: Maximum and last-epoch accuracy values, and number of trainable parameters for the LSTM model with various dimensions for the **embeddings** (RMSProp, $lr = 0.001$).

Dimension	1 (default)	2	8	32	64	128	188	256
Max Validation Accuracy	0.8815	0.8768	0.8801	0.8722	0.8825	0.8833	0.8823	0.8762
Last-Epoch Test Accuracy	0.8697	0.8729	0.8569	0.8610	0.8760	0.8651	0.8742	0.8599
Number of Parameters	98k	155k	498k	1869k	3697k	7354k	10781k	14666k

Table 3 shows that the accuracy does not change much when the dimensions of the embeddings change. This is quite unexpected, since moderately richer embeddings may result in a more discriminating model.

Sweep of the hyperparameter space:
We sampled the number of layers from $(1, 2, 3, 4)$, the dimensions of the hidden state from $(20, 280, 290, 305)$, and the dimensions of the embeddings from $(32, 180, 256)$. Although Table 1 showed that the a large number of layers led to bad models, we though that perhaps the model was going to perform well with a larger number of layers combined with larger embedding dimension and more hidden states. Therefore, we chose to sample all four values for varying the model depth. The dimensions of the hidden state $\in (10, 260, 285)$ seemed to have produced good results in Table 2, so we decided to sample the region around 285 a bit more, hence the hidden state dimensions $\in (20, 280, 290, 305)$. Changing embedding dimension by itself didn't seem to have had a lot of impact according to Table 3, so we sampled a wide range of different embedding dimensions $(32, 180, 256)$. The dropout rate is 0.2 for more complicated models models; most of the models tested in the hyperparameter sweep are "more complicated".
The models are ranked by the "accuracy to number-of-parameters" ratio. Since we discovered that a more complicated model may not bring much improvement to the accuracy, we decided to consider the model size as well as the model accuracy. The goal is to find the hyperparameter settings with the lowest "accuracy to number-of-parameters" ratio.
The **best hyperparameter settings** we found had the following statistics and hyperparameter settings:

```
acc 0.872, num_params 1819690, acc/num_params 4.79203e-07
EMBEDDING_DIM=32, HIDDEN_DIM=20, LR=0.001, OPTIM=rmsprop
```

Therefore, the best hyperparameter settings we found has a small hidden state and moderately-sized embeddings.

3.2 Results using the Synthetic Dataset

The synthetic dataset is much less varied and much more "standard" compared to the IMDB dataset. For example, there are no punctuation or non alpha-numeric characters in the synthetic dataset, whereas those do exist in the IMDB dataset. There are exactly 25k positive and negative entries each in the synthetic dataset. Here is a sample "review,sentiment" pair:

```
"through sheer numbers The story has such great depth in its tone which makes the story so original with other stories that are also very funny full hearted by how much more than
what you can say about this show It s a wonderful movie that is just one tiny little lamp and it was an amazing film a good book a novel I m sure they re both proud of their books
and ill served as a book because of how well written worl for me or ill made possible boonies a comic phoenix I love this series but not only one wojo jon fojo and the whole
world The art blojo is really beautiful and dwarred by many others on my list especially those who have seen the movies before ie some people who like them i am so happy that
they ve got these new shows now im so glad we did i think that there s something out there for everyone here xo s A Little Poppy i will always be ableto see another
favorite of mine if not even though compels were added to make it a long time ago ro ro o",positive
```

3.2.1 Naive Bayes Performance

Since the dataset is much more monotonous, we expected the performance of the Naive Bayes classifier to be much better on the synthetic dataset. The performance of the Naive Bayes classifier is extremely high (Figure 4). The accuracy is consistently very close to 1.00 regardless of the size of the training data. To discover the limit of the Naive Bayes classifier, we trained the model on very few inputs (right-hand plot in Figure 4) and observed that the accuracy is still very high. This means that the features of the two classes in the synthetic dataset are very well-separated.

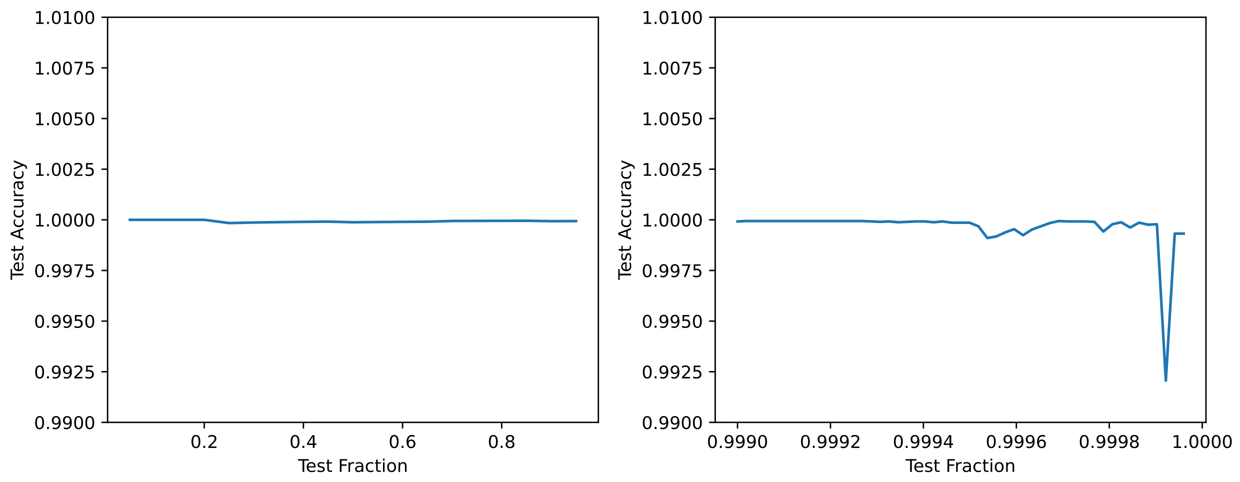
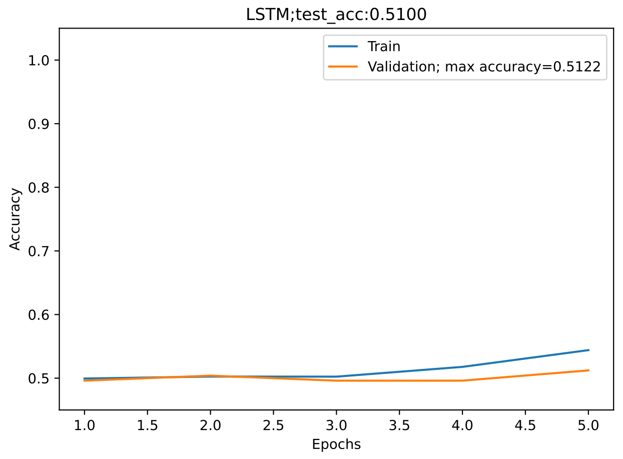
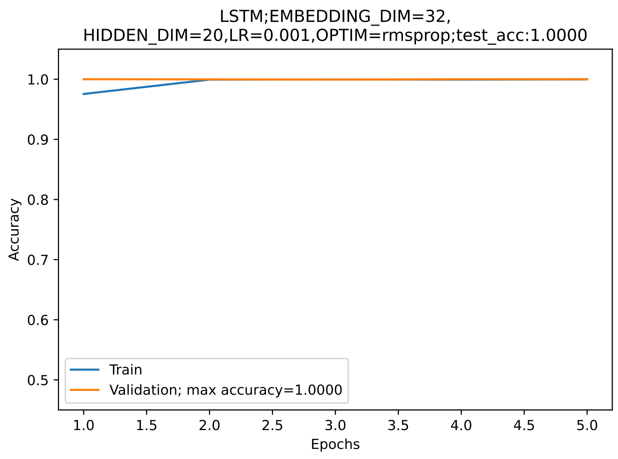


Figure 4: The Naive Bayes classifier trained on the synthetic dataset.

3.2.2 LSTM Performance



(a) LSTM with default hyperparameters



(b) LSTM with best hyperparameters

Figure 5: Accuracy curves of the LSTM trained and tested on synthetic data using default hyperparameters and the “best” hyperparameters according to the parameter sweep

With the best hyperparameters as specified previously, the LSTM comes close to having full accuracy (Figure 5b) and there is only a very small difference between the performance of the Naive Bayes classifier and the LSTM. Note that even with such a simplistic dataset and around 4,000 unique tokens (the IMDB dataset has around 57,000 distinct tokens), the LSTM model with the SGD optimiser and other default parameters is unable to train in five epochs. This demonstrates how unsuitable the SGD optimiser is for training the LSTM in this setting.
Note: we used the same data pre-processing and train-validation-test split strategies for the synthetic data as the real IMDB dataset.

4 Discussion

4.1 Performance using the IMDB dataset

On the real dataset, The Naive Bayes classifier achieved an accuracy of 0.8569, while the LSTM model achieved a higher accuracy of 0.8697 (Figure 3b). However, there are important trade-offs in terms of model size and computational efficiency.
The Naive Bayes classifier assumes independence between features and calculates probabilities based on this

assumption [15]. This makes it computationally lightweight and requires less training data compared to more complex models. As a result, the Naive Bayes classifier can achieve decent accuracy even with limited training data.
On the other hand, the LSTM model, a type of RNN, is geared to capturing long-term dependencies within sequences, which is not necessary for sentiment analysis. Although achieved a higher accuracy of 0.8697 (Figure 3b), there are trade-offs. One significant trade-off is its larger model size compared to the Naive Bayes classifier. LSTMs are much more complex, which results in a larger number of parameters. This larger model size requires more computational resources for training and inference.
Additionally, the LSTM model is computationally more expensive and slower compared to the Naive Bayes classifier. It is simply more time-consuming to communicate between different devices (CPU, GPU, and their respective RAM) and feed in data mini-batch by mini-batch. This can be a limiting factor when dealing with large datasets or real-time applications where efficiency is crucial.
Our computing environment took around one second for training and testing the Naive Bayes classifier on the 112-core Intel(R) Xeon(R) Gold 6348 CPU clocked at a maximum of 2.60GHz (note: only 1 to 2 cores out of the 112 logical cores are used for Naive Bayes and the GPU was not involved). On the other hand, training and testing the LSTM model for five epochs in the same computing environment requires the CPU to load data into the 2-device NVIDIA A100-PCIE-40GB GPU, takes more than 2GB of GPU memory, requires one GPU device to compute at around 35% of its maximum load, and takes more than 2 minutes. Hence, the computing requirements are much larger for the LSTM than the Naive Bayes classifier.

In addition, since we used a more complicated Regex-based tokeniser to pre-process the data for the Naive Bayes classifier, the input data to the Naive Bayes classifier may have been much cleaner compared to the input data to the LSTM model, since the tokenisation process applied in the LSTM training pipeline only involved splitting a review according to white space (non-alphanumerical characters were not removed). Therefore, more thorough string processing may have also contributed to the respectable performance of the Naive Bayes classifier. Hence, there exists a trade-off between model complexity and model performance. This trade-off is negligible, however, since the more complicated model is only more performant by around 1 percentage point.
The process of generating embeddings is also different. For Naive Bayes, we used the bag-of-words model to convert strings into embeddings. With the LSTM, we used a time series vector to represent each review. For sentiment analysis, the temporal relationship between different words in one review does not contribute much to the classification, since many bad reviews contain indicative adjectives. Therefore, the time-series vector used in the LSTM did not give the LSTM an edge over the Naive Bayes classifier.

4.2 Performance using the Synthetic Dataset

Both the Naive Bayes classifier and the LSTM model demonstrated excellent performance, achieving a perfect accuracy of 1.000. This indicates that both models were able to accurately classify the sentiment of movie reviews in the synthetic dataset. However, achieving a perfect accuracy on a synthetic dataset with clear class separation does not imply that both models would perform well on real-world datasets with more complex and nuanced text.
On the synthetic dataset there is no trade-off between accuracy and model size. In fact, the more complicated LSTM using default hyperparameter settings fails to train at all with the very simplistic synthetic dataset (Figure 5a). For the synthetic dataset, the Naive Bayes classifier is the better choice for the reasons concerning the performance-accuracy trade-off listed in section 4.1.

5 Acknowledgements

We express our gratitude for being granted access to the A100 Nvidia GPU associated with Professor Tingjun Chen's lab, as it was crucial for making the machine learning model train within the alotted time window.

References

[1] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[2] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. eprint: `https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[3] Andrew L. Maas et al. "Learning Word Vectors for Sentiment Analysis". In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150. URL: `http://www.aclweb.org/anthology/P11-1015`.

[4] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *CoRR* abs/1910.03771 (2019). arXiv: `1910.03771`. URL: `http://arxiv.org/abs/1910.03771`.

[5] Leandro von Werra. *GPT2-imdb-pos*. May 2021. URL: `https://huggingface.co/lvwerra/gpt2-imdb-pos`.

[6] Manuel Romero. *GPT2-imdb-neg*. May 2021. URL: `https://huggingface.co/mrm8488/gpt2-imdb-neg`.

[7] Daniel Jurafsky and James H. Martin. *Speech and language processing*. 2. ed., [Pearson International Edition]. Prentice Hall series in artificial intelligence. London [u.a.]: Prentice Hall, Pearson Education International, 2009, 1024 S. URL: `http://aleph.bib.uni-mannheim.de/F/?func=find-b&request=285413791&find_code=020&adjacent=N&local_base=MAN01PUBLIC&x=0&y=0`.

[8] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[9] Manish Sharma. *Sentiment Analysis (Movie Review)*. May 2020. URL: `https://github.com/DrManishSharma/NLP/blob/master/SentiAnalysis.ipynb`.

[10] Wes McKinney et al. "Data structures for statistical computing in python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[11] Pratyaksh Jain. *Basics of countvectorizer*. June 2021. URL: `https://towardsdatascience.com/basics-of-countvectorizer-e26677900f9c`.

[12] Vipul Gandhi. *Bag-of-words model for beginners*. Nov. 2019. URL: `https://www.kaggle.com/code/vipulgandhi/bag-of-words-model-for-beginners`.

[13] Christopher Olah. *Understanding LSTM networks*. 2015. URL: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[14] *Utilities for Generation*. URL: `https://huggingface.co/docs/transformers/internal/generation_utils`.

[15] A. Aylin Tokuc. *How to improve naive Bayes classification performance?* May 2023. URL: `https://www.baeldung.com/cs/naive-bayes-classification-performance`.

[16] Jason Brownlee. *Better naive Bayes: 12 tips to get the most from the naive Bayes algorithm*. Aug. 2019. URL: `https://machinelearningmastery.com/better-naive-bayes/`.

[17] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[18] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. May 2020. URL: `https://www.ruder.io/optimizing-gradient-descent/`.