FACULTY OF SCIENCE, ENGINEERING AND ENVIRONMENT

ASSESSMENT REPORT (NATURAL LANGUAGE PROCESSING)

Text Classification of Text generated by Large Language Models using Embedding Space

SUBMITTED BY

JOSHUA EMMANUEL EVUETAPHA (@00790872)

DATE

23-04-2024

INTRODUCTION

Text Classification is a popular Natural Language Processing (NLP) technique that automatically categorises text into predefined labels (Ali, 2022). It can be applied to various tasks, from email spam detection to language detection to sentiment analysis. Text classification uses supervised machine learning to classify text. It involves passing the text data and a predefined label into a machine learning algorithm to learn the patterns between the text and the label.

Large language models have been part of our daily lives since they became mainstream with Chatgpt. Many open-source models have been developed, and the number is growing. These models produce large volumes of textual data daily and have been integrated into many applications, producing more AI-generated content, from blog posts to posts on social media. This experiment aims to identify the AI model that generates text using Machine learning and Text classification. Four open-source LLMs were prompted to write an article on one thousand topics each. The following libraries will be used for this task: Tensorflow, an open-source machine learning library, and the GPT-4all Python SDK, which helps us create chat sessions with these LLMs in a Python environment.

The Large Language Models used in this experiment are;

- **Meta-LLama-3b-8b-Instruct** an open-source model with 8 billion parameters trained by Meta.
- **Nous-Hermes-2-Mistral-7b-DPO** is an open-source model with 7 billion parameters trained by Mistral & Nous Research.
- **Phi-3-mini-4k-instruct** is an open-source model with 3.8 billion parameters trained by Microsoft.
- **Orca-mini-3b** is an open-source model with 3 billion parameters, which Microsoft also trained.

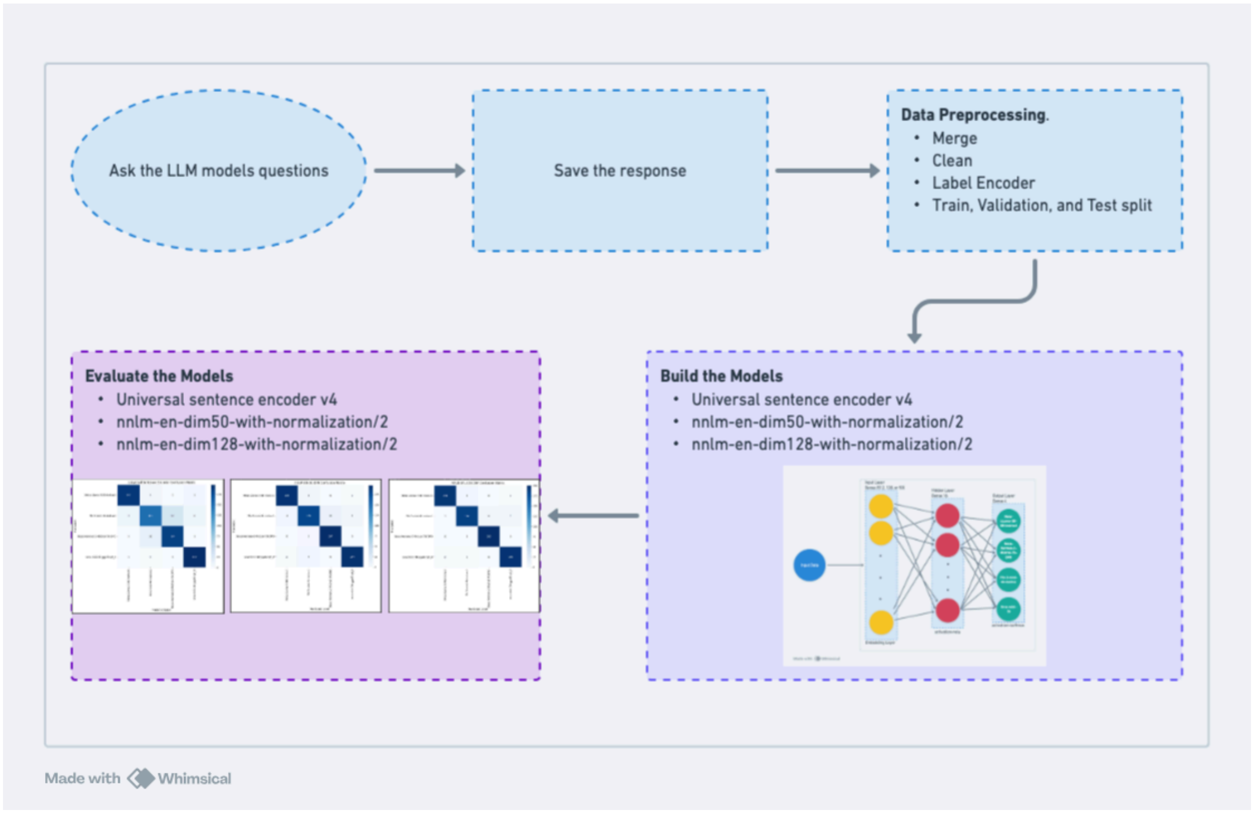

Ask the LLM models questions
Save the response
Data Preprocessing.
Merge
Clean
Label Encoder
Train, Validation, and Test split
Build the Models
Universal sentence encoder v4
nnlm-en-dim50-with-normalization/2
nnlm-en-dim128-with-normalization/2
Evaluate the Models
Universal sentence encoder v4
nnlm-en-dim50-with-normalization/2
nnlm-en-dim128-with-normalization/2
Made with Whimsical

Figure 1

METHODOLOGIES AND LIBRARIES

Libraries Used

The libraries for the tasks are:

- **Gpt4all:** Gpt4all Python SDK allows us to create chat sessions with multiple LLM models using Python.
- **Google.Colab**: This library allows us to access and store files on Google Drive from a Colab environment.
- **Pandas:** Pandas is a fast, powerful, flexible, and easy-to-use open-source data analysis and manipulation tool built with Python.
- **TensorFlow:** TensorFlow is a library for Machine Learning and Artificial Intelligence. It is free and open-source, popular for deep learning, and was developed by the Google Brain Team.
- **Tensorflow Hub:** Tensorflow Hub is a repository for pre-trained Machine Learning Models that can be fine-tuned and deployed. The Embeddings for this task are from Tensorflow Hub.

- **Scikit Learn is** a free, open-source machine learning and data modelling library for Python. It implements many machine learning algorithms for classification and regression tasks.
- **Numpy:** Numpy is a free, open-source mathematical and scientific computational library for Python. Numpy makes it easy to work with multi-dimensional arrays and matrices.
- **Matplotlib:** Matplotlib is a data visualisation library that helps to plot graphs in Python.
- **Seaborn:** Seaborn is a Data visualisation library built on top of Matplotlib. Seaborn makes creating clear and expressive plots easy and useful when working with pandas DataFrames.

DATA COLLECTION AND PROCESSING

The data were collected using the Gpt4all Python SDK to prompt each model with one thousand predefined article topics. Each model was sent the same prompt to write an article on the same topic, and the responses from each model were saved to a CSV file for further processing. The data collection process spanned days because of the large amount of GPU required to run a Large Language Model (LLM), which resulted in four CSV files, each containing responses and prompts from a particular Model.

Joining the Datasets

The data are stored in four different CSV files, each containing responses and prompts for a particular model. We need to put all the models data in a single Dataframe, and we will join them together using pandas concatenation to create a single Dataframe.

Data Exploration

The data set is balanced, with each Model having exactly 1000 prompts and responses.

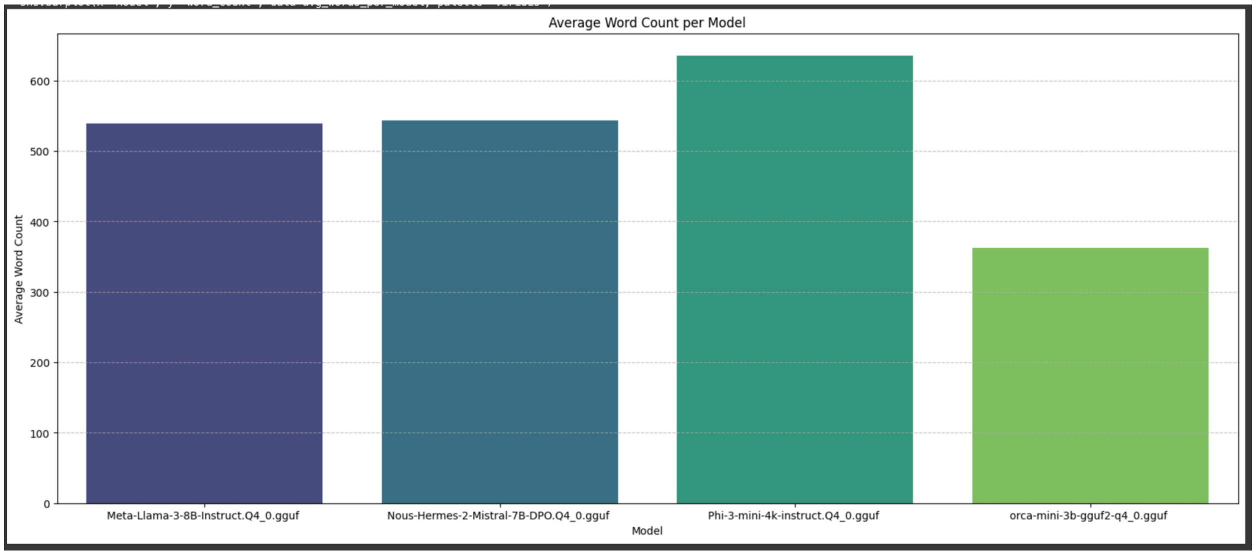


Figure 2

The average word per Model shows that the Orca-mini model gave the shortest article responses with an average of 368 words per response. In comparison, Phi-3-mini-4k-instruct gave the longest article responses with an average of about 668 words per response. Nous-Hermes-2-Mistral-7b-DPO came second with an average of 548 words per response, coming in a close third is Phi-3-mini-4k-instruct with an average of 519 words per response.

Preprocessing

The data consists of 4000 rows; 1000 rows from each model. Because the data is balanced, much preprocessing is not needed. However, responses from Meta-LLama-3b-8b-Instruct tend to include this character "**"; the model uses those characters to indicate headings and subheadings. While these characters would help classify responses from Meta-LLama-3b-8b-Instruct, they are not words, and because of that, they will be removed.

Encoding

Computers do not understand text, so we need to convert our labels (the LLM model) into numerical forms. We have four categories, so we will label them from 0 to 3, using the ***LabelEncoder*** from scikit learn. In our case, zero represents Meta-LLama-3b-8b-Instruct, one represents Nous-Hermes-2-Mistral-7B, two represents Phi-3-mini-4k-instruct, and three represents Orca-mini-3b.

Splitting

In order to train, validate and test the models, the dataset is split into training, validation, and test sets. The training set is used to train the model, while the validation set is used to validate the model during training. The test dataset is used to test or evaluate the model's performance. The datasets are split in the following ratio: 60% for training, 20% for validation, and 20% for testing. They are also split evenly across models, so each model has equal representation in the training, validation, and testing sets. The splitting process is as follows: First, the dataset is divided into Training and Temporary sets, 60:40. The training dataset is used for training the model, while the Temporary dataset is split into validation and testing sets in the following ratio, 50:50.

MODEL BUILDING AND EMBEDDING

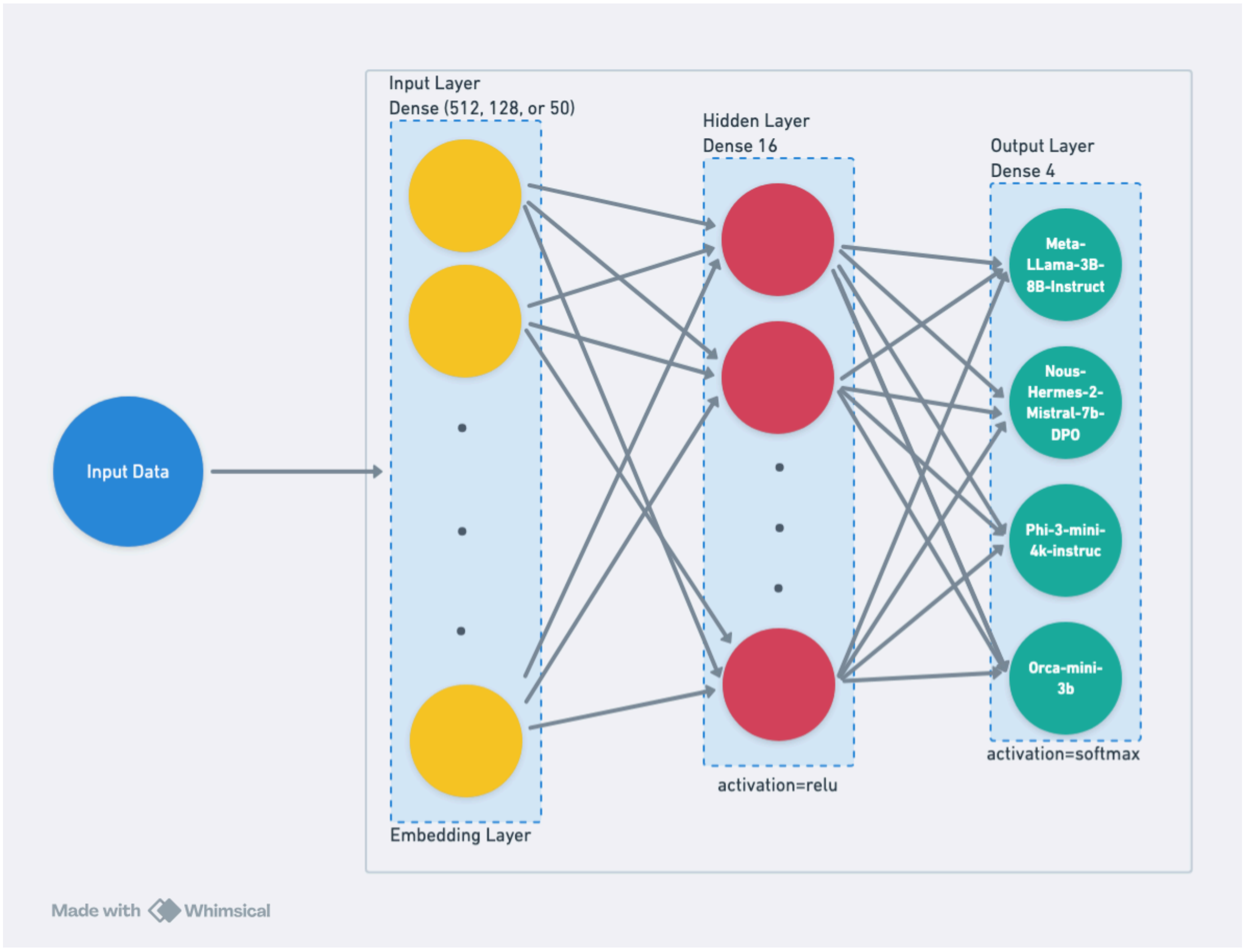


Figure 3

Model Architecture

We will build a Sequential Model, which allows us to stack layers one after another linearly. Sequential Models are one of the easiest types of models to build. The model has 3 layers, one input layer (embedding layer), one hidden layer and one output layer..

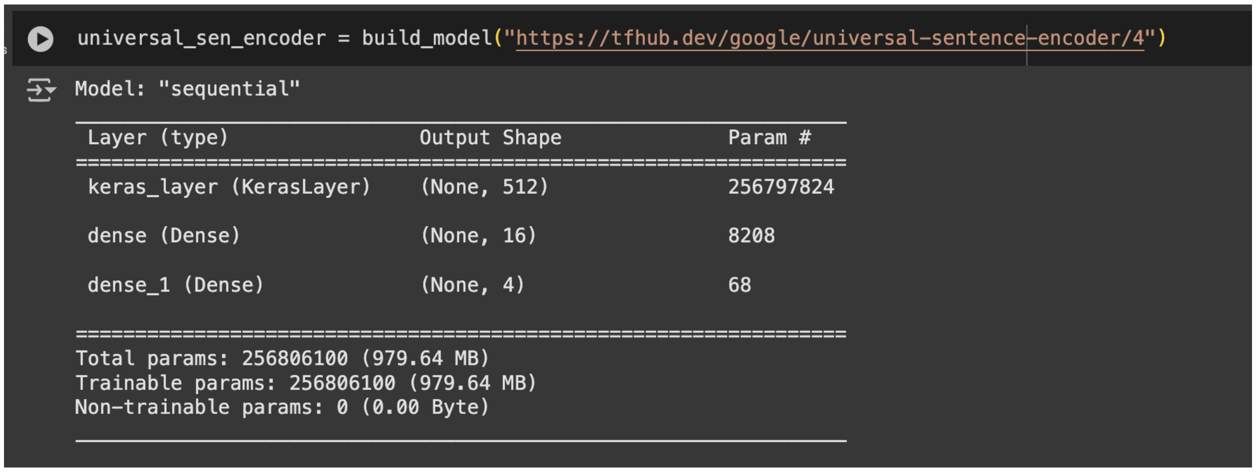
1. The first layer is the pre-trained embedding layer from TensorFlow Hub. This layer is trainable and will be fine-tuned on the datasets. Three different text embeddings will be used for this experiment.
2. The second layer is the hidden layer, a dense layer with 16 neurons and the Relu activation function. The second layer learns complex (non-linear) relationships in the training dataset.
3. The third layer is the output layer, a dense layer with four outputs indicating the number of classes in the dataset. It uses the Softmax activation to transform the vector of real numbers into a probability distribution. Each output value lies between 0 and 1, and the sum of all outputs equals 1, making it suitable for multi-class classification tasks.

Embeddings

This experiment involves three English embeddings from the Tensorflow Hub.

1. Universal Sentence Encoder v4
2. nnlm-en-dim50-with-normalization/2
3. nnlm-en-dim128-with-normalization/2

Universal Sentence Encoder V4



```
universal_sen_encoder = build_model("https://tfhub.dev/google/universal-sentence-encoder/4")
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 512)               256797824

 dense (Dense)               (None, 16)                8208

 dense_1 (Dense)             (None, 4)                 68

=================================================================
Total params: 256806100 (979.64 MB)
Trainable params: 256806100 (979.64 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 4

Layers:

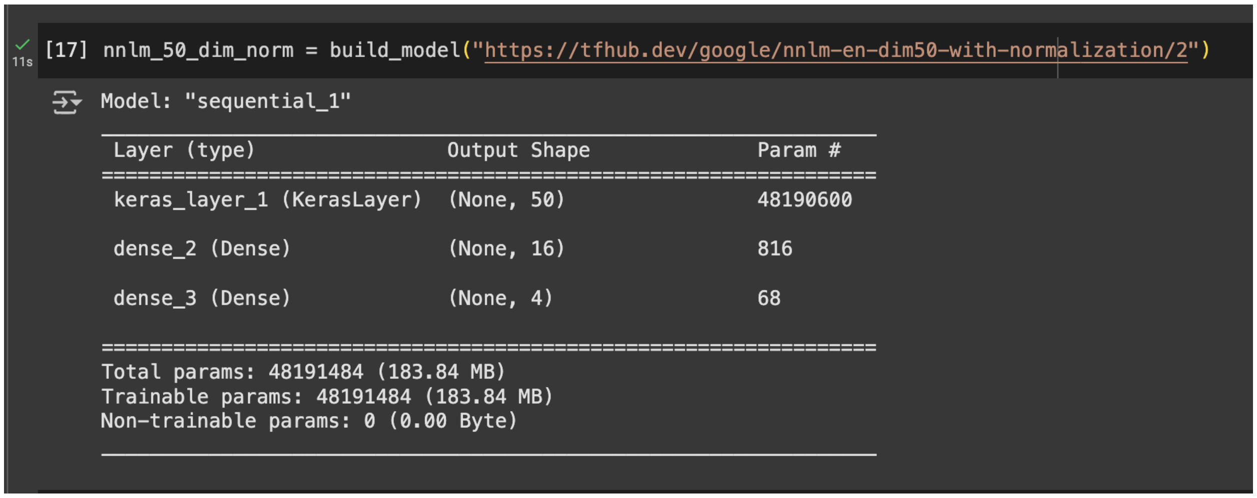
Input Layer (KerasLayer): The first layer is the Input layer, a pre-trained layer (Universal Sentence Encoder v4) with 256797824 parameters, and a vector of 512 dimensions as output.

Hidden Layer (Dense): Next is the Hidden layer, a Dense layer that transforms the 128-dimensional vector from the Input layer into a 16-dimensional output vector with 8208 parameters.

Output Layer (Dense): The final layer is the Output layer, which transforms the 16-dimensional vector from the hidden layer into a 4-dimensional output vector representing each class. It has 68 parameters.

This model has a total of 256797824 trainable parameters, and occupies 979.64 MB in memory.

NNLM-EN-DIM50-WITH-NORMALIZATION/2



```
[17] nnlm_50_dim_norm = build_model("https://tfhub.dev/google/nnlm-en-dim50-with-normalization/2")

Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer_1 (KerasLayer)  (None, 50)                48190600

 dense_2 (Dense)             (None, 16)                816

 dense_3 (Dense)             (None, 4)                 68

=================================================================
Total params: 48191484 (183.84 MB)
Trainable params: 48191484 (183.84 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 5

Layers:

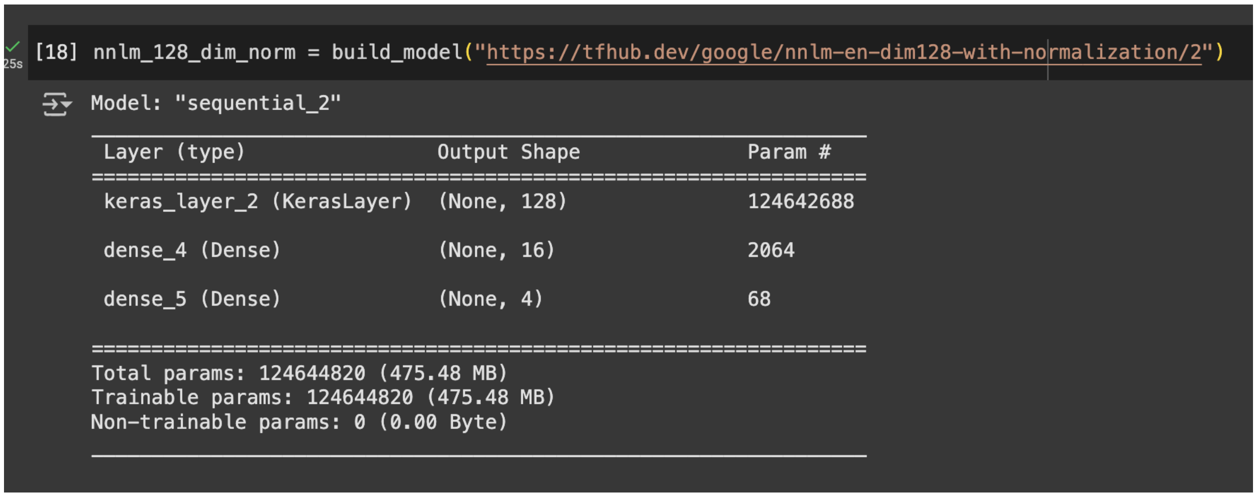
Input Layer (KerasLayer): The first layer is the Input layer, a pre-trained layer (nnlm-en-dim50-with-normalization/2), with 48190600 parameters, and a vector of 50 dimensions as output.

Hidden Layer (Dense): Next is the Hidden layer, a Dense layer that transforms the 50-dimensional vector from the Input layer into a 16-dimensional output vector with 816 parameters.

Output Layer (Dense): The final layer is the Output layer, which transforms the 16-dimensional vector from the hidden layer into a 4-dimensional output vector representing each class. It has 68 parameters.

This model has a total of 48190600 trainable parameters, and occupies 183.84 MB in memory.

NNLM-EN-DIM128-WITH-NORMALIZATION/2



```
[18] nnlm_128_dim_norm = build_model("https://tfhub.dev/google/nnlm-en-dim128-with-normalization/2")

Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 keras_layer_2 (KerasLayer)  (None, 128)               124642688 
                                                                 
 dense_4 (Dense)             (None, 16)                2064      
                                                                 
 dense_5 (Dense)             (None, 4)                 68        
                                                                 
=================================================================
Total params: 124644820 (475.48 MB)
Trainable params: 124644820 (475.48 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 6

Layers:

Input Layer (KerasLayer): The first layer is the Input layer, which is a pre-trained layer (nnlm-en-dim128-with-normalization/2) with 124642688 parameters, and a vector of 128 dimensions as output.

Hidden Layer (Dense): Next is the Hidden layer, a Dense layer that transforms the 128-dimensional vector from the input layer into a 16-dimensional output vector with 2064 parameters.

Output Layer (Dense): The final layer is the Output layer, which transforms the 16-dimensional vector from the hidden layer into a 4-dimensional output vector representing each class. It has 68 parameters.

This model has a total of 124644786 trainable parameters, and occupies 475.48 MB in memory.

Compilation Parameters

- **Optimiser**: Adam is a powerful optimiser that combines the strengths of two other optimisers, Momentum and RMSprop, in adjusting the learning rates during training (Geeksforgeeks, 2025).
- **Loss Function**: Categorical Cross-Entropy is a popular loss function ideal for multi-class classification problems. It computes the difference between the predicted and actual values to find the optimal solution for the model by adjusting its weight (Geeksforgeeks, 2024).
- **Metric**: Accuracy, which measures how accurately the model classifies the text. It is a fraction of the number of times the model makes the correct prediction.

RESULTS AND VISUALISATION

The result shows the performance of three text embeddings from Tensorflow Hub in classifying text generated by Large Language Models. The Models were trained for 25 epochs each, and with each epoch, the accuracy and loss for training and validation were reported.

Universal Sentence Encoder v4

Surprisingly, this model had the lowest accuracy of 92.5% and a validation loss of 0.2471, despite having the highest dimension out of the three models. It showed consistent improvements in accuracy and decreasing loss, but it started showing signs of overfitting after the sixth epoch, with little improvement on both training and validation accuracy. The low performance of this model can be attributed to overfitting, as the model has too many parameters but not enough data to learn from effectively. The performance of this model can be improved by getting more data from the LLM.

NNLM English 50-dim embeddings

This model had a final validation accuracy of 95.6% and a validation loss of 0.1764. It was almost at par with the performance of the NNLM-128-dim and beat the Universal Sentence Encoder Model despite having fewer dimensions. It showed consistent improvements in accuracy and decreased loss as the epoch increased, which suggests that the Model had a good learning progression over time.

NNLM English 128-dim embeddings

This model had a final validation accuracy of 96.25% and a validation loss of 0.1404. It is also the best-performing Model. It showed consistent improvements in accuracy and decreased losses as the epoch increased, which suggests that the Model had a good learning progression over time.

Conclusion

The NNLM models performed well, with validation accuracies exceeding 95%. This reflects their capability to handle the classification task. However, the Universal Sentence Encoder performed poorly in this task, suggesting overfitting because it has too many parameters and fewer data points to learn.

Training and Validation accuracy

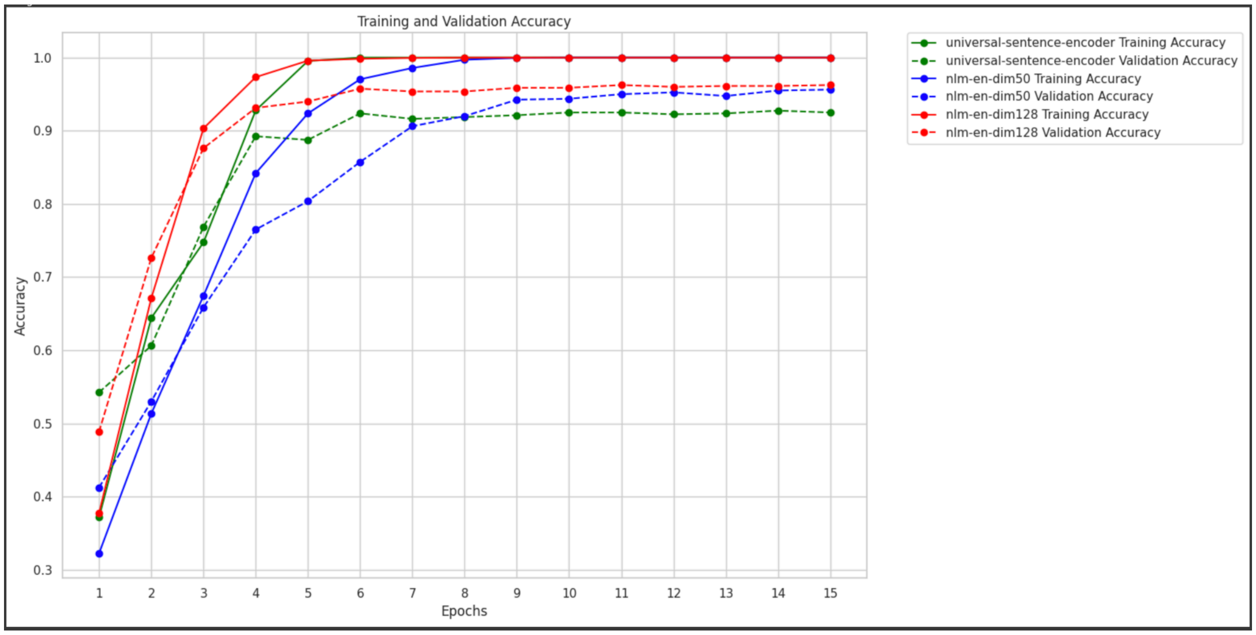


Figure 7

Universal sentence encoder v-4

The solid green line shows the training accuracy, and it started at a low accuracy of 38% but quickly converged at 100% accuracy by the sixth epoch. The dotted green line indicates the validation accuracy, and it tries to mirror the solid green line, but it was not consistent. This shows that the model has poor generalisation and ended with a final validation accuracy of 92%. This model had the lowest accuracy when compared to the rest.

NNLM English 50-dim embeddings

The solid blue line shows the training accuracy, starting at a low accuracy of 33% and reaching 100% by the 9th epoch. It was the last model to reach its peak training accuracy. The dotted blue line indicates validation accuracy. It mirrors the solid blue line. This shows the model has good generalisation and ended with a peak validation accuracy of 95.6%. This model had the second-highest accuracy, close to the 128-dimensional model.

NNLM English 128-dim embeddings

The solid red line shows the training accuracy, starting at a low accuracy of 35%. It converged at 100% accuracy by the sixth epoch, becoming the second model to reach its peak training accuracy. The dotted red line indicates validation accuracy. It mirrors the solid red line. This shows the model's good generalisation and ended with a peak validation accuracy of 96.25%. This model generally had the highest accuracy, making it the best model for our dataset.

Training and Validation loss

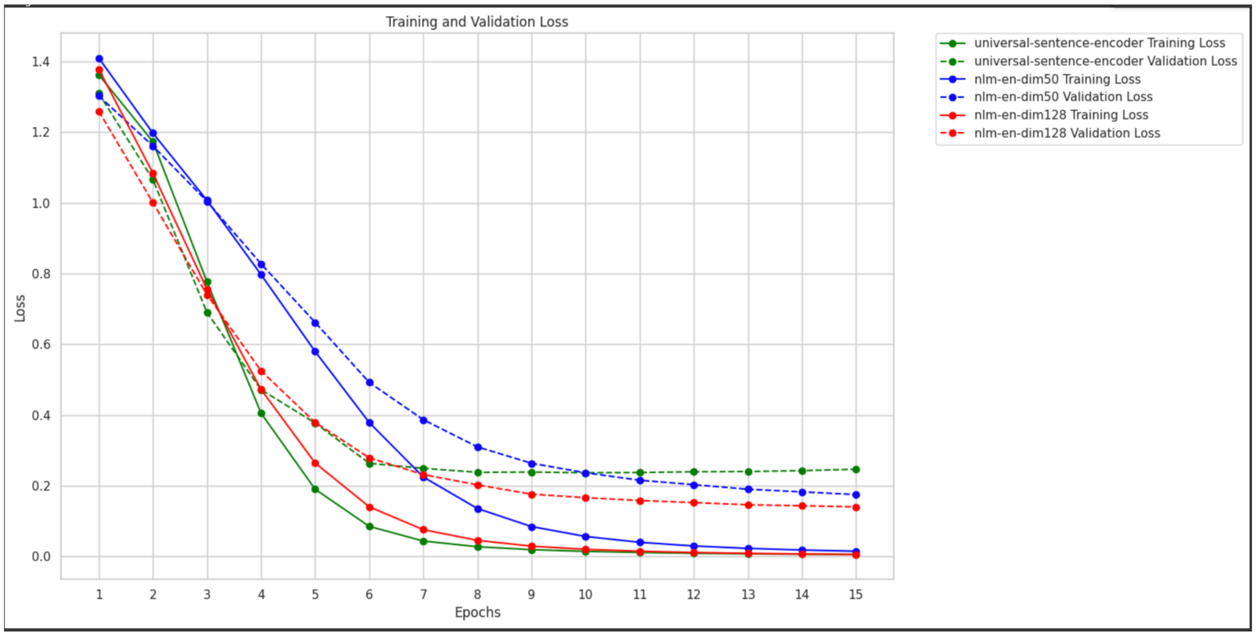


Figure 8

Universal sentence encoder v-4

The solid green line shows the training loss, and it decreases sharply from epoch 1 to 7 and then decreases gradually from epoch 7 to 15. The dotted green line indicates the validation loss and tries to mirror the solid green line. However, after the sixth epoch, the loss starts to increase, indicating overfitting, as the validation loss worsens with more epochs, while the training loss decreases (improves). Early stopping would help this model to prevent overfitting.

NNLM English 50-dim embeddings

The solid blue line shows the training loss, which decreases sharply and consistently from epoch 1 to 12 and then gradually from epoch 12 to 15. The dotted blue line indicates the validation loss, which mirrors the solid blue line to the end. This model shows continuous improvement even to the 15th epoch, suggesting that more runs will still improve it.

NNLM English 128-dim embeddings

The solid red line shows the training loss, which decreases sharply from epoch 1 to 8 and gradually from epoch 8 to 15. The dotted red line indicates the validation loss and mirrors the solid red line. However, after the 12th epoch, the loss does not decrease much, which suggests that more epochs might improve the model, but it will require fewer epochs than the 50-dimensional model.

Confusion Matrix

A confusion matrix, usually represented as a table, is a good tool for evaluating a model's performance. It provides better insights into the model's performance and makes it easy to compute other metrics, including precision, recall, and F-1 score.

The NNLM Models were outstanding in classifying text generated by Nous-Hermes-2-Mistral-7B, with NNLM-128-dim having 100% accuracy and NNLM-50-dim making just three errors. The Universal Sentence Encoder best classified text generated by orca-mini-3b with just three errors, while NNLM-50-dim and NNLM-128-dim made 6 and 4 errors, respectively. All three models were good at classifying text from Meta-LLama-3b-8b-Instruct. However, they all had the least success when trying to classify text generated by Phi-3-mini-4k-instruct, with Universal Sentence Encoder v4, NNLM-50-dim and NNLM-128-dim models making 50, 21 and 14 errors, respectively.

The highest false positives were Phi-3-mini-4k-instruct responses classified as Nous-Hermes-2-Mistral-7B responses, with Universal Sentence Encoder making 33 of such errors, NNLM-50-dim making 12 of such errors, and NNLM-128-dim making 6 of such errors. This shows that NNLM-128-dim is a better classifier.

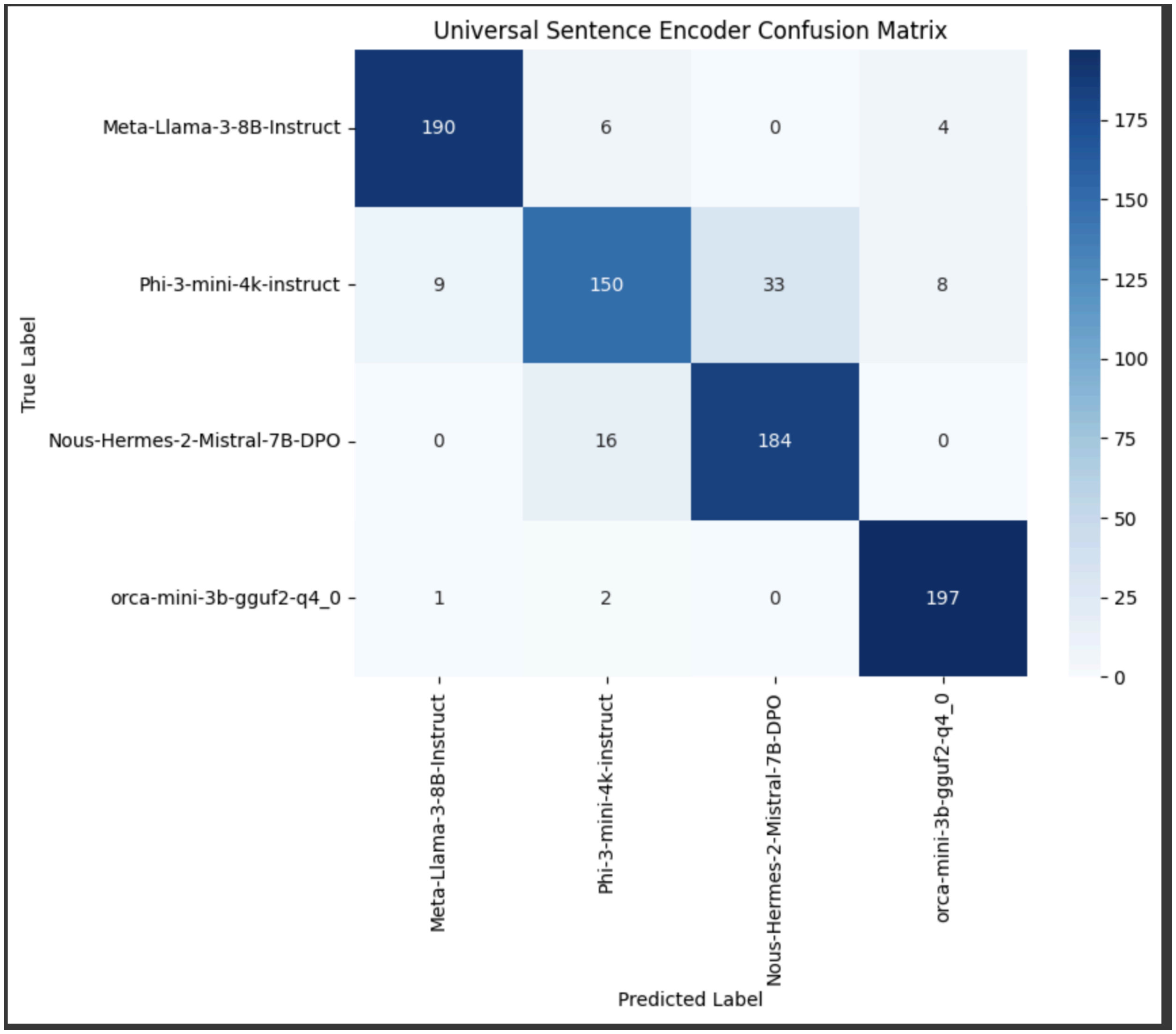

Universal Sentence Encoder Confusion Matrix
True Label
Predicted Label
Meta-Llama-3-8B-Instruct
Phi-3-mini-4k-instruct
Nous-Hermes-2-Mistral-7B-DPO
orca-mini-3b-gguf2-q4_0
190
6
0
4
9
150
33
8
0
16
184
0
1
2
0
197
175
150
125
100
75
50
25
0

Figure 9

‘

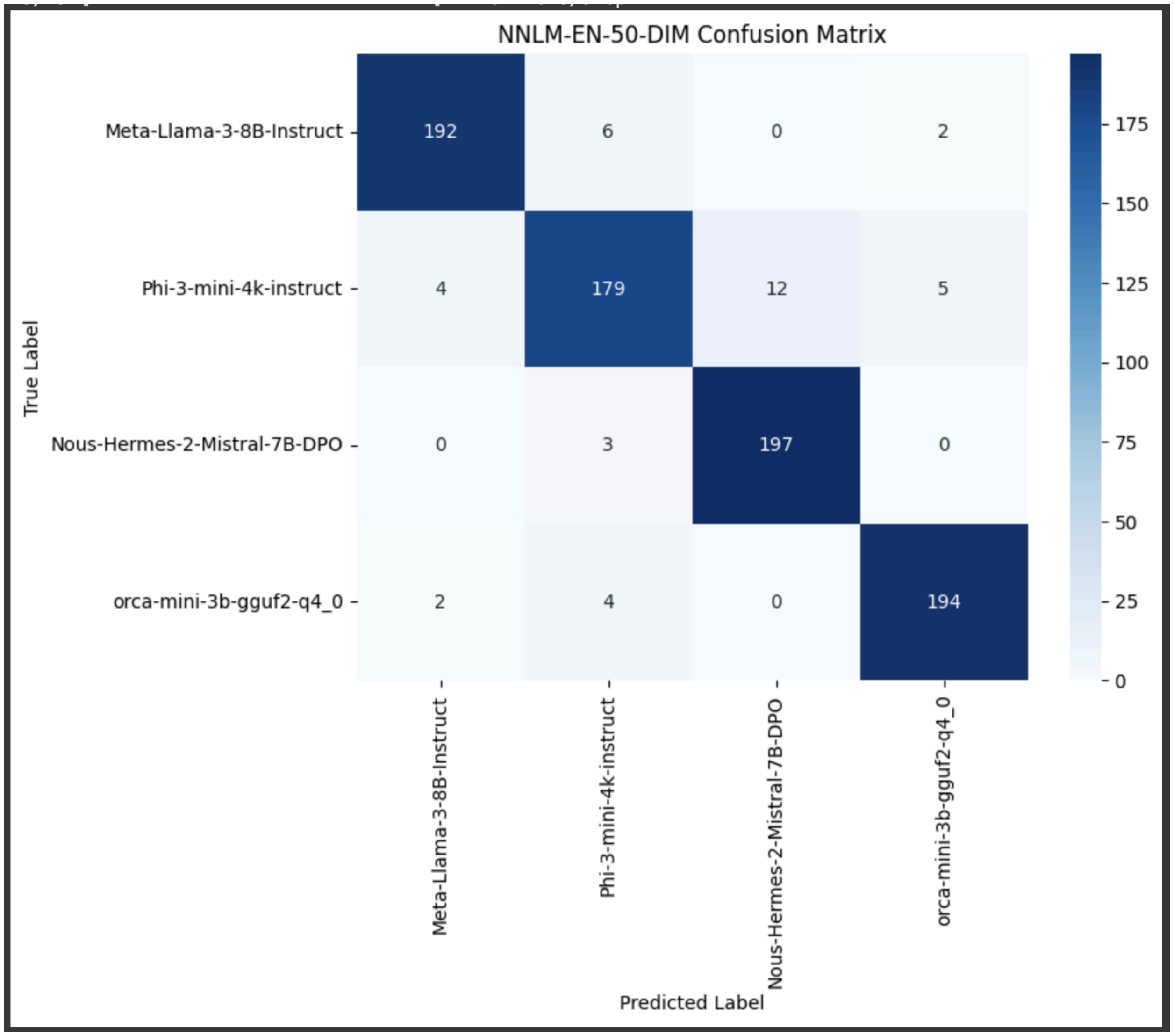

NNLM-EN-50-DIM Confusion Matrix
True Label
Meta-Llama-3-8B-Instruct
Phi-3-mini-4k-instruct
Nous-Hermes-2-Mistral-7B-DPO
orca-mini-3b-gguf2-q4_0
192
6
0
2
4
179
12
5
0
3
197
0
2
4
0
194
Meta-Llama-3-8B-Instruct
Phi-3-mini-4k-instruct
Nous-Hermes-2-Mistral-7B-DPO
orca-mini-3b-gguf2-q4_0
Predicted Label
175
150
125
100
75
50
25
0

Figure 10

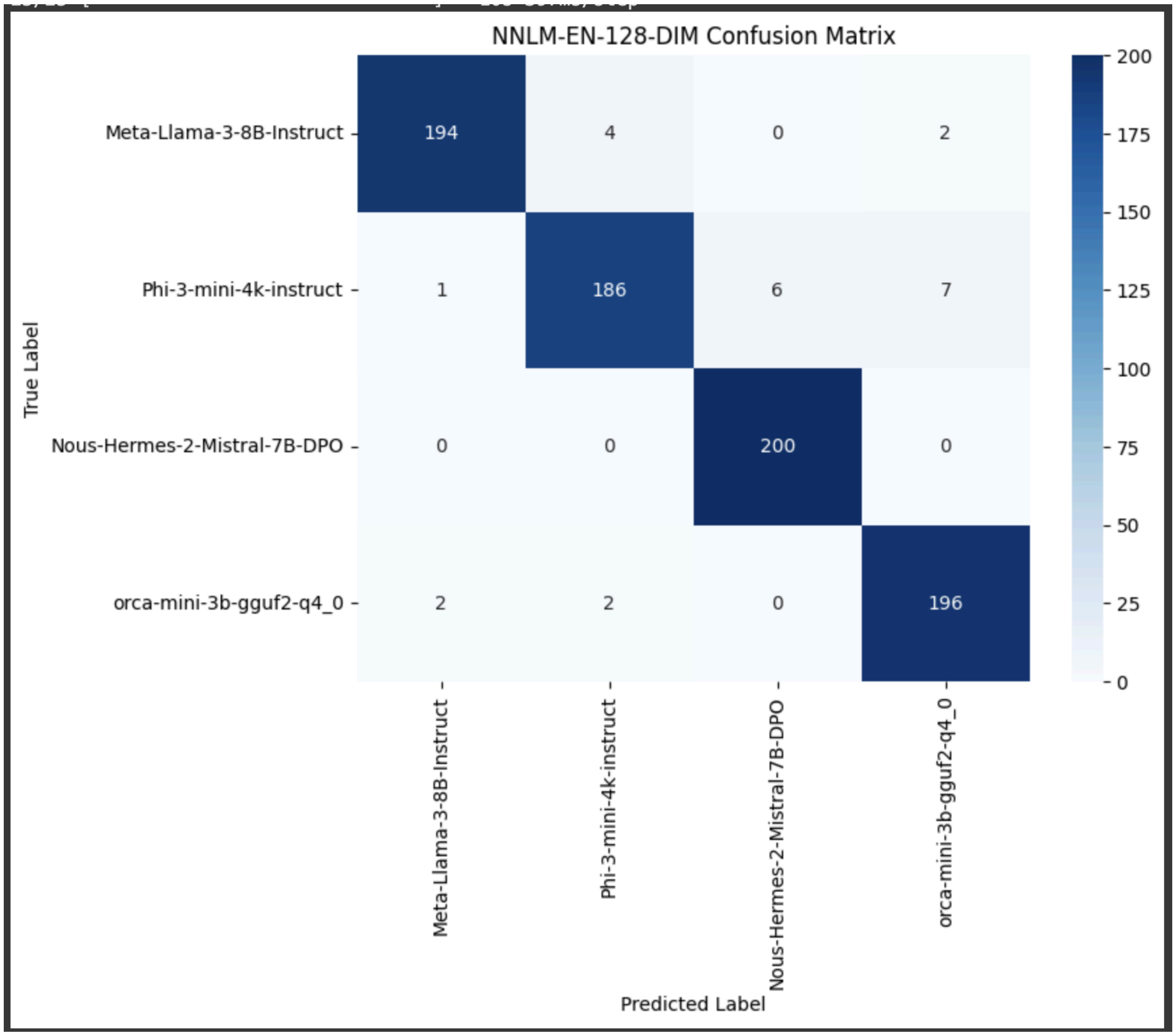


Figure 11

COMPARISON BETWEEN GOOGLE COLAB AND LINUX

Attribute	**Google Colab (cloud T4)**	**Linux machine (local 16 GB)**
The library used for building the model and versions	Most of the libraries used were already installed on Colab, so there is no need to install them again. However, "tensorflow" and "tensorflow-hub" were downgraded to version 2.15	The code was run on Python version 3.9.21 because of TensorFlow version 2.15. The following were installed: "jupyterlab==4.4.0", "notebook==7.4.0", "numpy==2.2.5",

	and 0.15, respectively.	“gpt4all==2.8.2”, “pandas==2.2.3”, “seaborn==0.13.2”, “matplotlib==3.10.1”, “tensorflow==2.15” and “tensorflow-hub==0.15”.
Speed of training Universal Sentence Encoder	It took about 140 seconds per epoch. In total, 35 minutes to train the model.	It took about 70 seconds per epoch. It took 17 minutes and 30 seconds to train the model.
Speed of training nnlm-en-dim50-with-normalization/2	It took about 30 seconds per epoch. It took 7 minutes and 30 seconds to train the model.	It took about 21 seconds per epoch. It took 5 minutes and 15 seconds to train the model.
Speed of training nnlm-en-dim128-with-normalization/2	It took about 76 seconds per epoch. It took 15 minutes and 10 seconds to train the model.	It took about 50 seconds per epoch. In total, it took 10 minutes to train the model.

Regarding training speed, Google Colab showed a more uniform time per epoch, while the local Linux machine had more irregular times per epoch. Training on Colab also took longer than on the local Linux machine.

CONCLUSION

In this report, three Text classification models were built to classify text generated by four open-source Large language models and analyse their performance. The results show that the NNLM models outperformed the Universal Sentence Encoder in classifying these texts. This is due to the overfitting of the Universal Sentence Encoder model, which has too many parameters but very little training data.

The NNLM model with 128 dimensions outperformed the other models with a validation accuracy of 96.25%, followed by the NNLM model with 50 dimensions with a validation accuracy of 95.6%. The Universal sentence encoder had a validation accuracy of 92%, which is decent but poor compared to the NNLM models. The Universal sentence encoder reaches its peak accuracy and loss faster than the rest of the models, which shows that the model learns faster than the rest, but needs more data to work well.

REFERENCE

Moez Ali. (2022, Nov 23). Understanding Text Classification in Python. Retrieved from https://www.datacamp.com/tutorial/text-classification-python
Geeksforgeeks. (2024, Sept 17). Categorical Cross-Entropy in Multi-Class Classification. Retrieved from
https://www.geeksforgeeks.org/categorical-cross-entropy-in-multi-class-classification

Geeksforgeeks. (2025, March 4). What is Adam Optimizer?. Retrieved from https://www.geeksforgeeks.org/adam-optimizer

https://docs.gpt4all.io/gpt4all_python/home.html#chat-session-generation