Final Project Report

Thomas Katraouras 03215
Ioannis Kalamakis 03225
Alexandros Betsas 03271

ECE418 Neuro-Fuzzy Computing
2023-24

University of Thessaly

Contents

Introduction

The task of text classification involves categorizing text documents into one or more classes or categories, making it a fundamental task in natural language processing (NLP). The complexity of language poses significant challenges in accurately understanding and predicting categories, especially when dealing with a large number of potential classes and subclasses, as is the case with 17 main classes and 109 subclasses in this project.

Data Collection and Preparation

The first step in any machine learning project is to collect or find a dataset that matches the task's requirements. For text classification, this typically involves a dataset with text documents and their corresponding labels. The dataset used in this project contains news articles categorized in level 1 and level 2 categories. Initially, to understand the data we have to deal with, we calculated the number of unique words across all articles in it and the average number of words per article. We also calculated the standard deviation for level 1 and level 2, to understand how balanced our dataset is.

We got these results:

Average number of words per text: 637.44
Number of unique words in all texts: 303777
Standard Deviation for Category Level 1: 227.027
Standard Deviation for Category Level 2: 0.904457

Based on the number of average words per text and number of unique words across all texts, we defined our two constants, MAX_SEQUENCE_LENGTH and MAX_NUM_WORDS, as 600 and 300000 respectively.

The standard deviation for Level 1 suggests that the distribution of texts within Level 1 is quite high, signifying a potential imbalance of the dataset for this category.

On the other hand, standard deviation of texts within Level 2 is low, therefore this category is most likely quite balanced.

Data Loader and Preprocessor

0.1 DataLoader Class

We implemented a custom class DataLoader, in order to keep data loading a standalone part of the project, so any kind of data can be fed into the preprocessor and neural network with little amounts of modifications. The class is designed to facilitate the loading and splitting of text data into training and testing sets.

The constructor initializes the class attributes, which are the path to the dataset, the name of the column containing text data, the name of the column containing level 1 labels, the name of the column containing level 2 labels and the desired train / test split size. While the load_data method handles the actual loading and preprocessing of the dataset. The class reads the dataset from the path specified and extracts the necessary features. It separates the data into text features (X) and two sets of labels (y1 and y2), and identifies the unique labels for each category. It then splits the data into training and test sets using scikit-learn's function train_test_split and according to the specified proportions, and returns all the values.

0.2 TextPreprocessor Class

The TextPreprocessor class implements several steps to clean and prepare the text data, including lowercasing, removing punctuation, digits, and stopwords, and handling contractions.

Firstly, the constructor method initializes the class attributes. These are the maximum number of words an input can have (usually defined as a constant), the maximum number of words to be kept in the Tokenizer's dictionary (also usually defined as a constant), a dictionary storing all punctuation and special characters for feature cleaning with Tokenizer, the Tokenizer itself (initialized with the custom filters and the maximum number of words in its dictionary) and a stopwords dictionary, which uses the NLTK punkt corpus to store stopwords used for preprocessing. More specifically, the Keras Tokenizer is a tool for converting text into sequences of integers, so it can be fed into a neural network. The stopwords module from the NLTK (Nature Language Toolkit) library is to access a predefined list of common English stopwords that occur frequently in test but often do not carry significant meaning to the understanding of the text.

The remove_stopwords method is responsible for removing common stopwords from the given text. It iterates over each world in the text after splitting it, and for each word, it checks if the word is not in the set of stopwords. If the word is not a stopword it is included in the list. Finally, it joins the selected words back into a single string, separated by spaces and returns it.

The preprocess_text function is responsible for applying a series of preprocessing steps to the text data. It takes a list of texts as input and applies the preprocessing steps to every text in the list. First of all, it turns all characters in the text to lowercase. This helps with reducing the vocabulary size by treating capital and lowercase words as identical. re.sub then replaces all hyphens and em/en dashes in the text with a space. It then removes all digits in the text, as they do not hold any significant value in the specific natural language processing task. contractions_dict is a dictionary that maps contractions to their expanded forms, helping in text preprocessing by converting shortened word forms to their full equivalents (e.g., "can't" to "cannot"). It includes all variation of apostrophes. Then, we perform a substitution replacing the contracted form with its expanded form. The flag re.IGNORECASE ensures that the substitution is case-insensitive, allowing the code to handle contractions regardless of their capitalization (although we have already converted the text to lowercase). Then again, we remove standalone possessive " 's " and its variations that have not already been expanded above. We then split the text into standalone sentences using NLTK's sent_tokenize module and remove stopwords from each sentence using the function remove_stopwords mentioned above. After removing stopwords and joining the split sentences, we remove any single character words that might be left over from the previous preprocessing steps. Finally, we append the current preprocessed text to a list and proceed with the next text until we are done, and return the preprocessed texts list.

The fit_on_texts method is the one we call to preprocess our text, which in turn calls preprocess_text described above to prepare the text and fits the Tokenizer with the preprocessed text by calling its method fit_on_texts.

The preprocess_text_data method is designed to preprocess a collection of texts to be fed into a neural network model. It converts each text in the input into a sequence of integers, using the Tokenizer's texts_to_sequences method. Then, it pads them to ensure they all have the same length, and returns them.

The plot_token_concentration method plots the concentration bar plot for the N most frequent tokens in dictionary of the Tokenizer.

Dataset Loading and Preprocessing

We initiate our data preprocessing by utilizing the two classes (DataLoader and TextPreprocessor). This preparation work includes important tasks like cleaning up the text, tokenizing, padding and shuffling the data. Shuffling the dataset serves a crucial purpose, ensuring that our neural network encounters a diversity of categories during training. By preventing the network from becoming biased towards specific categories, we foster a more balanced learning process. In order to emulate real-world scenarios, we fit the tokenizer exclusively with the training data. This decision ensures that our testing data remains unknown to both the tokenizer and the neural network, simulating real-world unseen data. Furthermore, we enrich our understanding of the tokenized vocabulary by visualizing the distribution of the 30 most frequent tokens post-tokenization. Then we encode the labels for both of our classes and subclasses and turn them to categorical variables.

Neural Networks

0.3 Level 1 Classification

For the neural network we chose to implement a convolutional neural network.

ARCHITECTURE:

- Embedding Layer: The first layer of the model responsible for converting one-hot encoded vectors into a lower-dimensional, continuous vector space, where similar words are closer in the vector space. Its input dimensions are equal to the vocabulary size of the Tokenizer, the output dimensions are equal to 100 (this parameter can be changed), its input length is equal to the maximum sequence length (usually a constant) and it is trainable, since we train our own embeddings instead of using pretrained ones like Word2Vec.

- Convolutional Layer: It applies 640 convolutional filters to the embedded word vector to extract local patterns, each filter slides over the word vectors, capturing features from pairs of adjacent words. The kernel size 2 means each filter covers two words at a time.

- BatchNormalization Layer: It comes right after the convolutional layer and before its activation. It makes sure these patterns are evenly spread out before they go to the next step, which helps the model learn faster and more smoothly. This makes it easier for the model to pick up complex patterns without getting stuck.

- LeakyReLU Activation Function: The LeakyReLU activation function introduces a slight twist to the model's learning process by allowing a small, positive flow for negative values, rather than cutting them off completely. It helps ensure that the model doesn't ignore or lose minor yet potentially important patterns during learning.
- GlobalMaxPooling Layer: It reduces the dimensionality of the model by taking the maximum value over the time dimension for each feature map and effectively captures the most important signal from each feature map.
- Dropout Layer: Randomly setting a fraction of the input unit to 0 at each update during training it forces the model to learn more robust features and prevents overfitting. This Dropout Layer (after the convolutional layer) has a rate of 0.4 .
- First Dense Layer: This layer further processes the features extracted by the convolutional and pooling layers. It has 640 units, matching the convolutional filters and is followed by batch normalization and LeakyReLU activation, similar to the convolutional setup.
- Dropout Layer: This Dropout Layer (after the first Dense Layer) has a rate of 0.3 .
- Second Dense Layer: The second and final (output) Dense Layer maps these features to the 17 possible classes. It has 17 units, corresponding to the 17 classes and uses the softmax activation function to produce a probability distribution over the classes.

TRAINING:

To train the model we used Categorical Crossentropy as our loss function, RMSProp with a learning rate of 0.0007 and a RHO of 0.5 as our optimizer. For training / validation metrics we calculate the categorical accuracy, precision and recall. We trained for 25 epochs and a batch size of 32.

Additionally, we split 10% of the training dataset as the validation dataset. The reason we do this instead of using Keras' validation_split , is to introduce randomness to the validation set.

We trained the first model and counted the training time to be **91.98 seconds** for the above configuration.

The graphs (Figure 1, Figure 2) depict the training process of a neural network.

The first graph (Figure 1) shows the model loss, we can clearly see that the training loss marked by the blue line starts quite high and decreases sharply, indicating that the model is learning and improving its predictions on the training data over time. The smooth and steady decline suggests good convergence behavior of the model during the training process. The validation loss marked by the orange line follows a similar sharp decrease, mirroring the training loss. However, as epochs increase, the loss flattens out and starts to increase slightly, suggesting the model begins to overfit and there is no need for additional epochs.

The second graph (Figure 2) shows the model accuracy on the training and validation datasets. Marked by the blue line, the train accuracy of the model, it starts low and rapidly increases, leveling off as the model begins to fit the training data well. The high training accuracy is expected and indicates that the model can learn the classification task. Marked by the orange line, the validation accuracy also improves quite quickly at first before hitting a plateau, indicating that the model cannot further generalize beyond this point and it might begin to overfit.

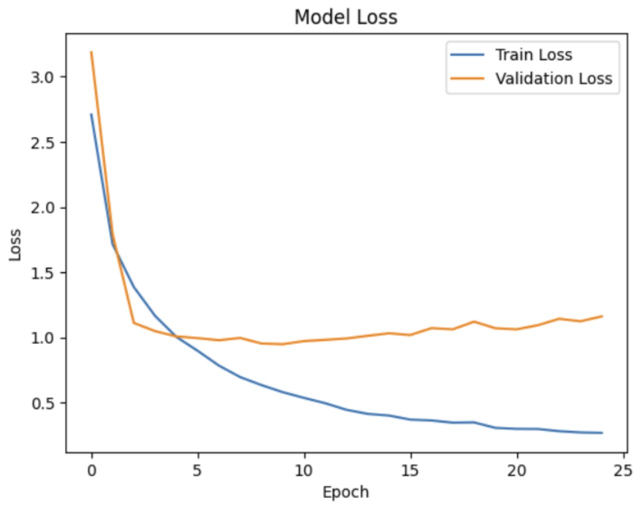

Figure 1: First Model (Level 1 Classifier) Loss Plot

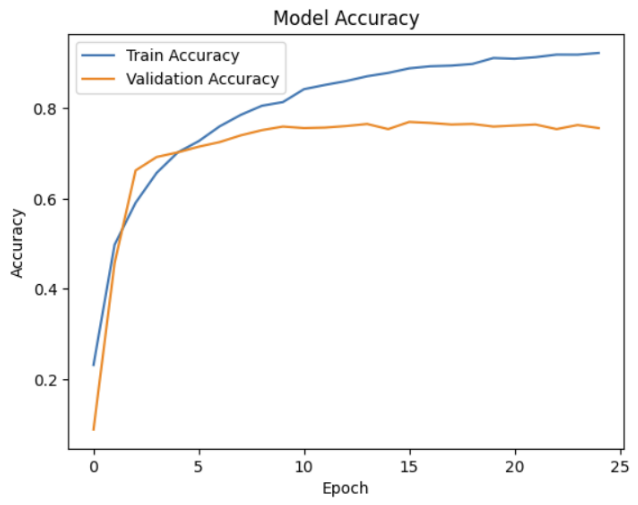

Figure 2: First Model (Level 1 Classifier) Accuracy Plot

FINAL WEIGHTS:

In our Jupyter Notebook, we printed the final weights and biases for each layer of our model (the ones which actually have weights). Due to their large volume we cannot include these directly in our report.

PREDICTIONS:

We then use the trained model to make predictions on the test dataset.

For us to be able to calculate metrics related to the performance of our model we have to convert predictions back to text. We accomplish that by first transforming predictions to class labels, and then reversing predicted class labels back to text with the LabelEncoder's inverse_transform method. We do the same for the true labels (y_test), by converting one-hot encoded labels back to categorical labels and reversing them to text.

Our metrics are:

- Precision: measures the proportion of true positive predictions among all positive predictions made by the classifier.

$$Precision = \frac{True\ positives}{True\ Positives\ +\ False\ Positives}$$

- Recall: measures the proportion of true positive instances that were correctly identified by the classifier out of all actual positive instances in the dataset.

$$Recall = \frac{True\ positives}{True\ Positives\ +\ False\ Negatives}$$

- F-score: is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall, giving equal weight to both metrics.

$$F - score = 2 * \frac{Precision\ x\ Recall}{Precision\ +\ Recall}$$

- Support: refers to the number of actual occurrences of each class in the dataset. It provides context for precision, recall and F-score by indicating the relative frequency of each class in the dataset.

- Accuracy: measures the proportion of correctly classified instances among all instances in the dataset. It is a global measure of the model's performance across all classes and subclasses.

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

The first model's metrics are calculated and printed in our Jupyter Notebook. Once again, due to the large volume of data, we cannot put them directly into the report without making it hard to read.

However, one we want to highlight is accuracy. The first model achieved an **accuracy of 75.64%** on the test set.

INFERENCE TIME:

To measure inference time, we called the model's .predict method 20 times, each time with a random sample from the test dataset. We counted all times and calculated the average. **Average Inference Time: 35.54 ms** .

CONFUSION MATRIX:

To help us visualize predictions and how our model behaves, we plotted a confusion matrix (Figure 3).

We notice the network gets confused in similar labels, for example it confused 14 (religion and belief) with 12 (society) a lot, which is indeed hard to classify, even for a human, and would probably need a larger sample to improve accuracy.

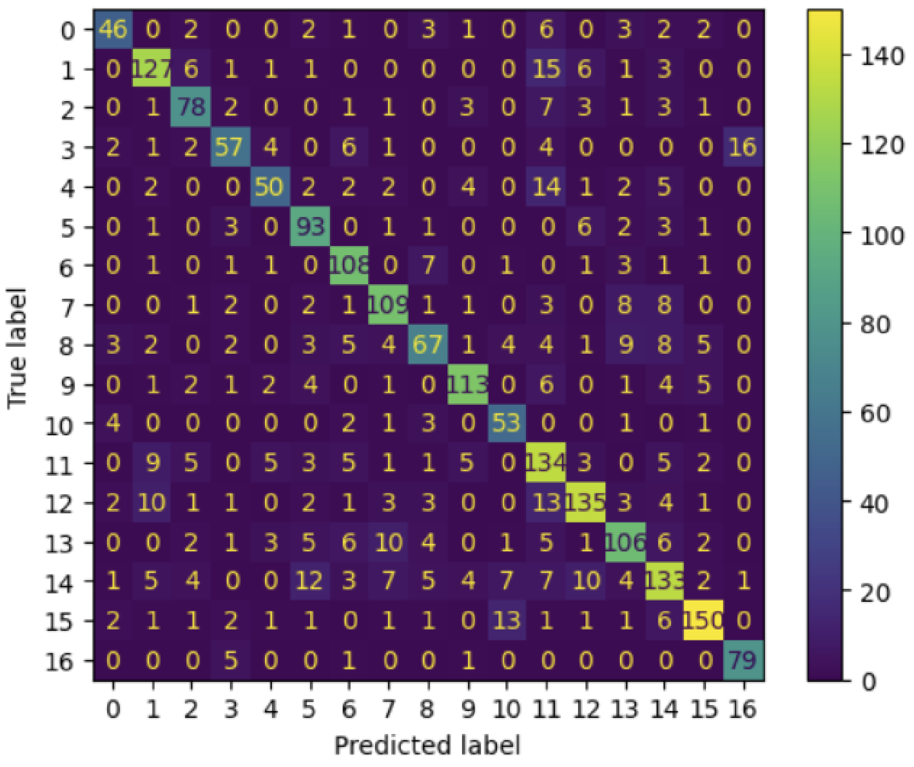

Figure 3: First Model (Level 1 Classifier) Confusion Matrix

0.4 Level 2 Classification

ARCHITECTURE:

The network architecture and parameters for the level 2 (subclass) classification are the same as the first network, with the only difference being in the output Dense Layer, which has 109 units, corresponding to the 109 classes.

TRAINING:

The loss function, optimizer and metrics are the same as the first network.

We trained the second model and counted the training time to be **92.22 seconds** for the above configuration.

The graphs (Figure 4, Figure 5) depict the training process of a neural network. We notice analogous performance to the first model (although not the same numbers).

The first graph (Figure 4) shows how the model gets better over time at making predictions on the training data, indicated by the blue line that goes down smoothly. This means the model is learning well. The orange line shows the model's performance on new data it hasn't seen before (validation data), which also gets better but then levels off and starts to get worse, suggesting the model is starting to memorize the training data too much and isn't learning to predict new data as well.

The second graph (Figure 5) displays the model's accuracy, with the blue line showing it does a great job of correctly predicting the training data over time. This is what we expect. The orange line shows the model's accuracy on the validation data, which improves quickly at first but then doesn't get much better. This indicates the model is having difficulty making correct predictions on new data after reaching a certain point.

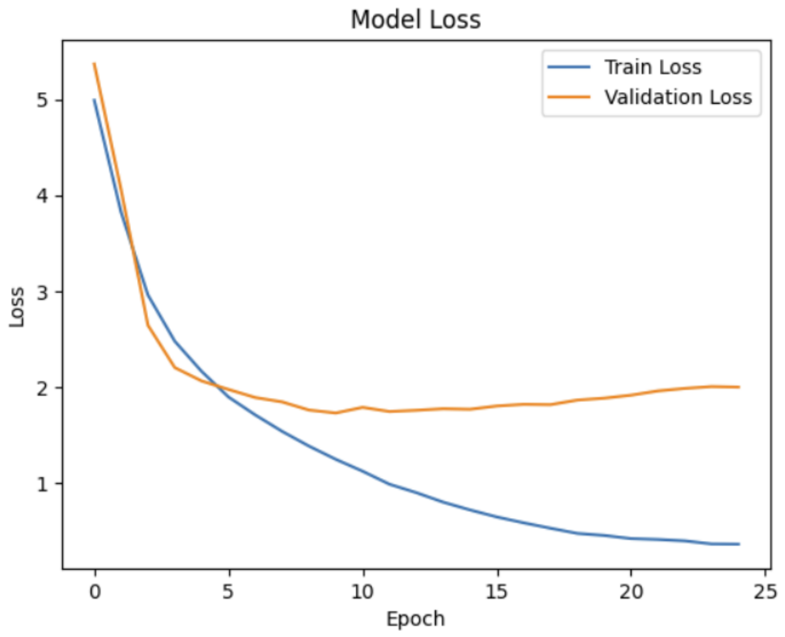

Figure 4: Second Model (Level 2 Classifier) Loss Plot

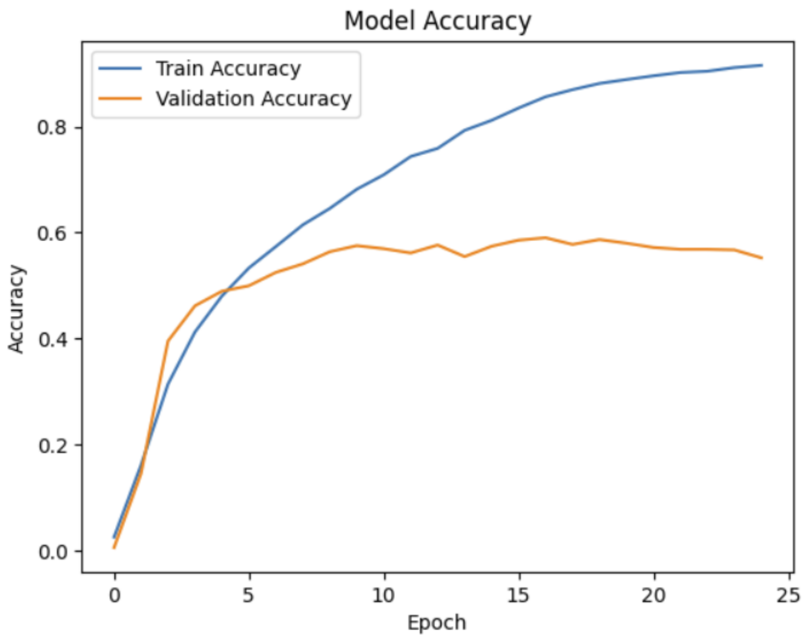

Figure 5: Second Model (Level 2 Classifier) Accuracy Plot

FINAL WEIGHTS:

In our Jupyter Notebook, we printed the final weights and biases for each layer of our model (the ones which actually have weights). Due to their large volume we cannot include these directly in our report.

PREDICTIONS:

We then use the trained model to make predictions on the test dataset, after converting both the predicted and real labels to text.

The first model's metrics are calculated and printed in our Jupyter Notebook. Once again, due to the large volume of data, we cannot put them directly into the report without making it hard to read.

The second model achieved an **accuracy of 58.20%** on the test set.

INFERENCE TIME:

Like on the first model, to measure inference time, we called the model's .predict method 20 times, each time with a random sample from the test dataset. We counted all times and calculated the average. **Average Inference Time: 36.51 ms** .

CONFUSION MATRIX:

Since we have too many classes (109), a confusion matrix would be of no use since it would be too cluttered to read any data.

Transformers Implementation

The results we got above are satisfactory for the dataset we have. However, we wanted to attempt to see how high the accuracy can get. So we decided to utilize transformers, which are pretrained models, for our classification task (essentially we did transfer learning).

We chose the distilbert-base-uncased model, since it's a faster variation of the well-performing bert-base-uncased model.

We did a simple preprocessing by shuffling the dataset, mapping the labels to numbers and tokenizing the data.

We initialized the model with GPU and defined these training parameters:

- learning rate: 2e-5
- optimizer: adamw
- train batch size: 16
- eval batch size: 64
- warmup steps: 500
- weight decay: 1e-5
- train epochs: 15

These were purely based on info we found on Huggingface's website from the model creator and people who had tested the model.

After training and testing the model for both level 1 and level 2 classifications, we got the results:

1st model (Level 1 classification)

- Training Loss = 0.37554
- Accuracy = 78.9 %
- Confusion Matrix in Figure 6

2nd model (Level 2 classification)

- Training Loss = 1.07710
- Accuracy = 62.8 %

There are more metrics on our Jupyter Notebook evaluating the two models' performance.

We notice an improvement compared to our implementation. We believe these powerful pretrained models have potential for even higher accuracy with hyperparameter tuning. Unfortunately, their training time is too high (up to 24 hours on GPU), making it impossible to experiment with them in the context of our project, due to limited time.

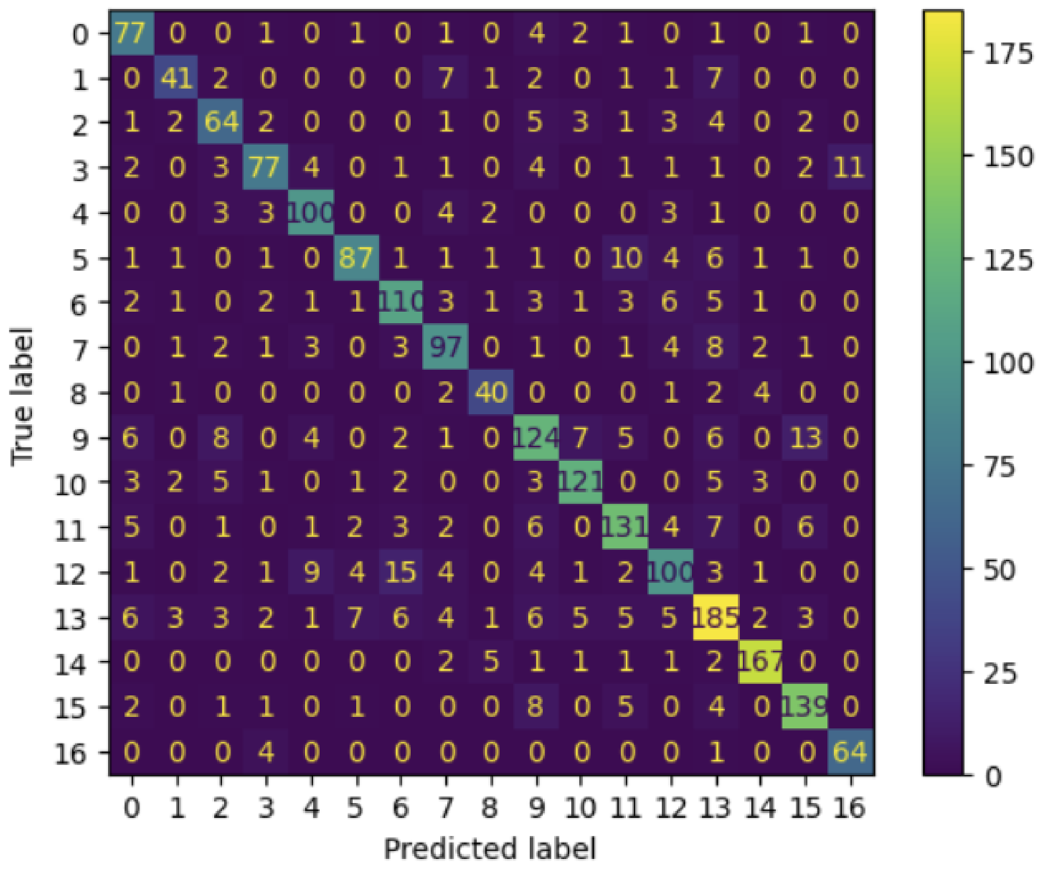

Figure 6: First transformer model confusion matrix