DA5401:Data Analytics Laboratory

ML-Challenge Report

Gokulakrishnan B
DA24M007
M.Tech DSAI - 2024-26

Introduction

The DA5401 end semester ML challenge is a multi-label classification problem, that expects the participants to come up with various deep learning algorithms to cross the baseline score, after proper data preprocessing techniques.

The labels are ICD10 codes. ICD-10 codes are a global system for classifying and coding medical conditions, procedures, and causes of death.

Data Engineering

We are given with 2 npy files named embedings_1.npy and embedings_2.npy that contains input embeddings, which can be loaded with numpy using load function and followed by concatenation as follows

```
data1 = np.load('/kaggle/input/da5401/embeddings_1.npy')
data2 = np.load('/kaggle/input/da5401/embeddings_2.npy')

X = np.concatenate((data1, data2), axis=0)
```

The labels are stored in 2 text files corresponding to 2 spy files. They are named icd_codes_1.txt and icd_codes_2.txt. The number of rows/ lines in these txt files are same as the number of rows in the .npy embeddings files. Each row contains single or multiple rows that are separated by ';'. This can be preprocessed by the following code

```
with open('/kaggle/input/da5401/icd_codes_1.txt', 'r') as file1, open('/
kaggle/input/da5401/icd_codes_2.txt', 'r') as file2:
    y1 = [line.strip().split(';') for line in file1]
    y2 = [line.strip().split(';') for line in file2]

y_combined = y1 + y2
```

Now we have a list named y_combined that have each element as a list that contains the multiple labels. To convert these labels into number for the models to train on, we use multilabel binarizer from sklearn using the code below

```
mlb = MultiLabelBinarizer()
mlb.fit(y_combined)

y_encoded_1 = mlb.transform(y1)
y_encoded_2 = mlb.transform(y2)

y_encoded = np.concatenate((y_encoded_1, y_encoded_2), axis=0)
```

Now, we have X_encoded and y_encoded, that has the numerical input and label matrices.

Now we perform scaling on input data. For this, we import standard scaler module from sklearn.preprocessing. This can be done by following code

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X = scaler.fit_transform(X)
```

Make sure to use the same scaler and mlb_transform object to transform and inverse transform test input and predictions respectively.

We are using PyTorch library to build our deep learning models. So we will convert the numpy matrices as torch tensor and put them in a data loader function, so that it can be passed to the models created.

To convert the numpy matrices into tensor, we use the following code

```
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y_encoded, dtype=torch.float32)
```

To load the tensors into a loader block, we use the following code

```
train_dataset = TensorDataset(X_tensor, y_tensor)
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
```

The batch size that I used is 128. I experimented it with various batch sizes like 16,32,64,256 as well. There are no significant difference in accuracies of 16,32,64 and 128. But using batch size as 256 resulted in significant loss. As the batch size increases, more data are trained in parallel, so less training time. Considering these two factors, I chose batch size as 128.

Sampling

I did not use any frequency specific sampling here. The initial idea was to increase the the rows with rare features with repeating features. Then I realised most of the labels are rare. Thus I dropped the idea.

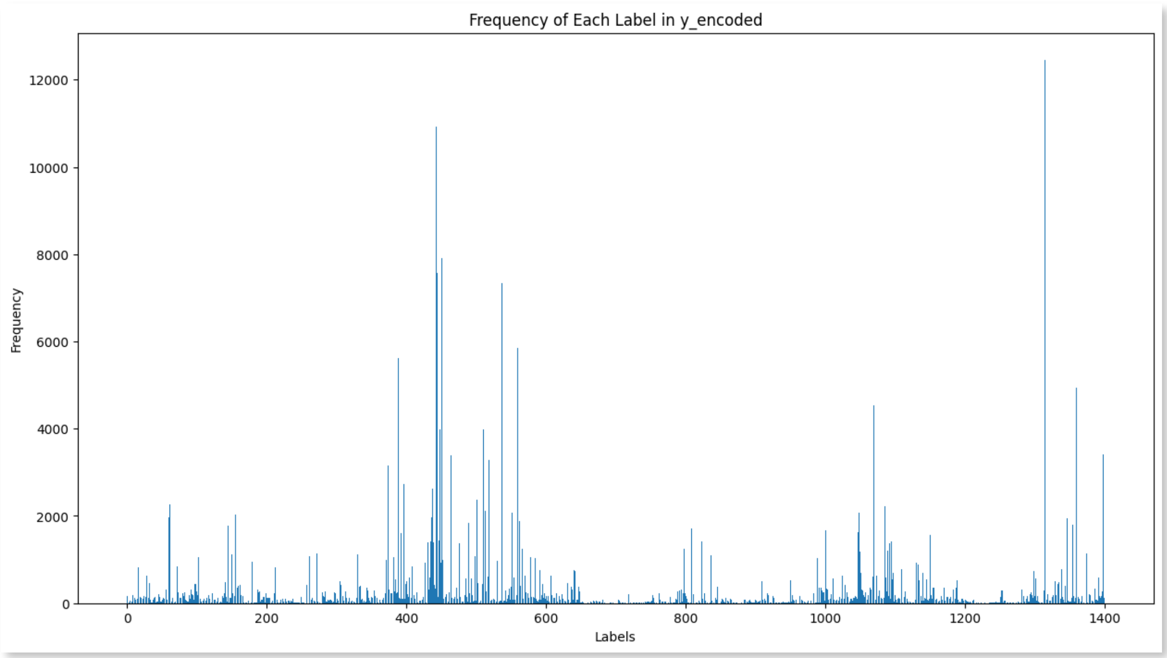
Another crazy idea I had was to repeat embedding 1 after concatenating embeddings 1 and 2. The reason is that, using simple ANN only on embedding 1 gave higher accuracy on test set (on Kaggle) than using embedding 2. So I thought of appending data 1 again. On implementing this approach first time, my accuracy got micro f2 score got increased by 0.02, by then on second run, it decreased. So I decided this is not the right approach.

EDA with visualisation

The dataset consist of 198982 rows and 1024 columns. That means there are 198982 instances of data points with each data point having 1024 features. This makes it quite large.

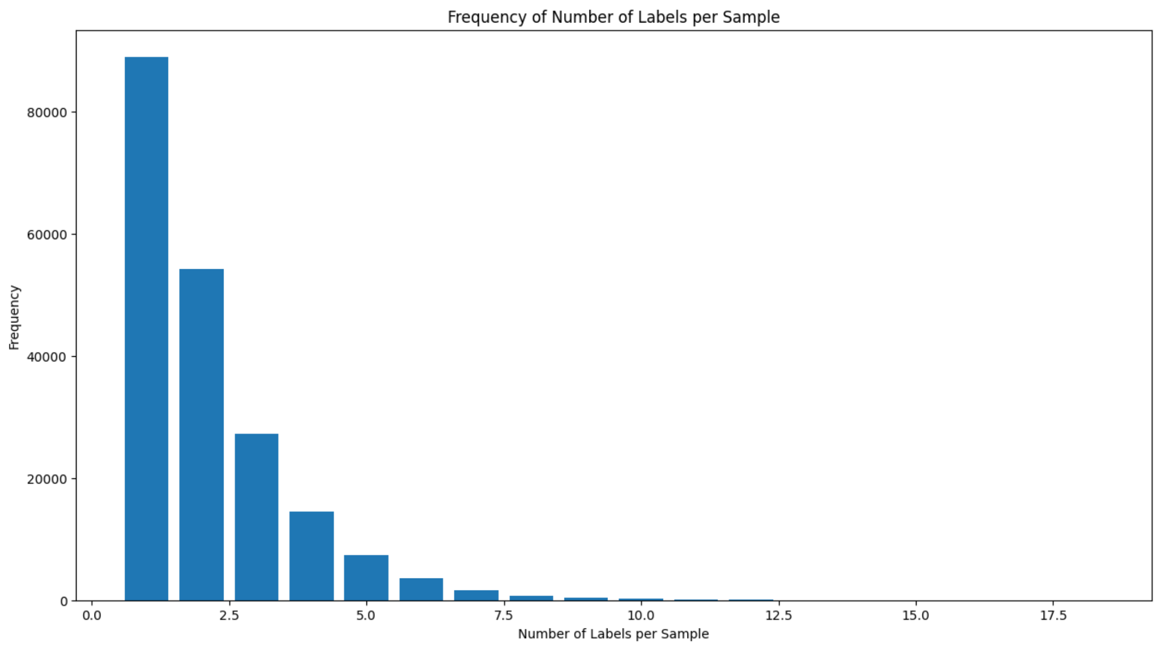
The number of unique labels in the dataset are 1400. The embeddings are generated through GatorTron LLM.

Now, lets plot a bar chart to know about the frequency distribution of the labels.



Here, we can see that few labels have high frequencies, while majority labels have very few occurrences.

Now lets plot the bar chart for the frequency of number of labels.



Here, we can see that there are more number of rows with number of labels between 0 to 8. There are only few occurrences of data where the number of labels is greater than 8.

Model Selection with performance metrics

My first approach was to use a simple ANN, then to use attentionCNN, and siamese network. Each approaches are explained in detail below.

ANN

Artificial neural networks have many layers of neurons and their weighted connections among layers enables it learn complex patterns. Neurons in each layers is activated by input. A single layer can only learn linear pattern, but increasing the number of layers facilitates it to learn non linear pattern as well.

The model that I implemented had 3 layers - 1 input layer, 1 hidden layer and 1 output layer with batch normalisation. With hidden layer having 1024 neurons, and input and output dimensions matching the dimensions accordingly. This model resulted in private score of 0.466.

Without batch normalisation, the accuracy score is reduced to 0.226. Adding batch normalisation between every layer, increases the accuracy score.

In the above model, I used dropout as 0.2. Decreasing the dropout overfits the model, thus training error reduces, but test error increases. Increasing the dropout under-fits the model, thus we get high training bias. So dropout = 0.2 is a sweet spot .

Increasing the number of layers overfits the data, and it results in accuracy of 0.43. Increasing the number of neurons in the single layer, slightly increased the accuracy, but it took more time.

The above mentioned model is implemented with the below code

```
class MultiLabelNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(MultiLabelNN, self).__init__()
        self.fc1 = nn.Linear(input_dim, 1024)
        self.bn1 = nn.BatchNorm1d(1024)
        self.fc2 = nn.Linear(1024, output_dim)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = self.fc1(x)
        x = self.bn1(x)
        x = torch.relu(x)
        x = self.dropout(x)
        x = torch.sigmoid(self.fc2(x))
        return x
```

Attention CNN

The SimpleAttentionCNN model combines convolutional layers with an attention mechanism to focus on important features, enhancing feature extraction for multi-label classification. The attention layer refines feature maps from the CNN, while fully connected layers output the final multi-label predictions.

Attention CNNs have an advantage over traditional ANNs by better capturing spatial patterns through convolutional layers. The attention mechanism helps the model focus on the most relevant features, improving its performance. Additionally, CNNs use fewer parameters, making them more efficient and faster to train.

I used attention CNN with 1 convolutional layer and with similar parameters from above CNN model, I got public score of 0.434 and private score of 0.433.

Although it is faster than ANN, accuracy is slightly poor than vanilla ANN with batch normalisation.

Siamese network

A Siamese network consists of two identical neural networks that share weights and are trained to compare two inputs. It is commonly used for tasks like similarity detection, where the model learns to determine if two inputs are similar or different.

Siamese networks are advantageous over vanilla ANN in multi-label classification because they specialise in learning pairwise relationships between labels, which can improve the model's ability to capture label dependencies. Unlike ANN, which process all labels independently, Siamese networks can better leverage shared representations between similar labels, enhancing accuracy in tasks with complex label correlations.

I used siamese network with fine tuned parameters and got public score 0.454 and private score of 0.449. The accuracy of siamese network is better than attention CNN and comparable with transformer.

The code to implement siamese network is as follows

```
class SiameseFNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SiameseFNN, self).__init__()

        # Shared layers
        self.shared_fc1 = nn.Linear(input_dim, 1024)
        self.shared_bn1 = nn.BatchNorm1d(1024)
        # Output layer
        self.fc_out = nn.Linear(1024, output_dim)

    def forward(self, x):
        # Shared representation layers
        x = self.shared_fc1(x)
        x = self.shared_bn1(x)
        x = F.relu(x)

        # Output layer
        x = torch.sigmoid(self.fc_out(x))

        return x
```

So, at last I submitted 10 submissions with ANN, 5 submissions with attention CNN, and 5 submission with siamese network with various parameters.

Hacks and Workarounds that you have invented

I experimented with duplicating the data and appending to it again. Theoretically it should not affect the performance, but I got improved accuracy in some cases.

Another workaround is, once you complete training the model for the specified number of epochs, change the batch size and train the model on top the same model again. It may sound weird, but I got better results this way.

Training and Validation performance tables of your final model

	Micro F2 score
Training	0.926
Public	0.468
Private	0.466