DA5401 Data Analytics LAB ML Challenge REPORT

Private Score on kaggle: **0.494**

Rishabh Garg

DA24C026
Joint MSc Data Science and AI

Problem Statement

The goal is to perform multi-label classification of ICD10 codes for outpatient EHRs based on LLM-generated embeddings of the data. The dataset includes embeddings from large language models (LLMs) and ICD10 codes associated with various diagnoses. The objective is to accurately predict ICD10 codes based on the embeddings, making this a multi-label classification problem where each sample may have multiple correct ICD10 codes. Performance is evaluated based on the **micro F2 score**, which places a higher emphasis on recall than precision.

Methodology

1. **Exploratory Data Analysis (EDA)**: To gain insights into the embeddings and label distributions, identifying key characteristics in data and labels.
2. **Data Engineering and preprocessing:** Transforming ICD10 labels for model training, applying label encoding, handling missing values if any, and splitting the data into training and validation sets.
3. **Model Design**: Developing machine learning / deep learning models with different configurations and optimize based on F2 scores.
4. **Optimization Techniques**: Implementing custom modifications, including hyperparameter tuning and a customized evaluation metric score function to maximize performance.
5. **Hacks and Tricks**: Employing techniques to optimize results, including union based aggregation and F2 score adjustments.
6. **Final Evaluation and Submission**: Creating a final prediction file in the required format for submission, ensuring reproducibility.

Exploratory Data Analysis (EDA)

In the ipython notebook titled **EDA.ipynb**, I explored both the label data and the embedding data to understand the structure and relationships within the dataset.

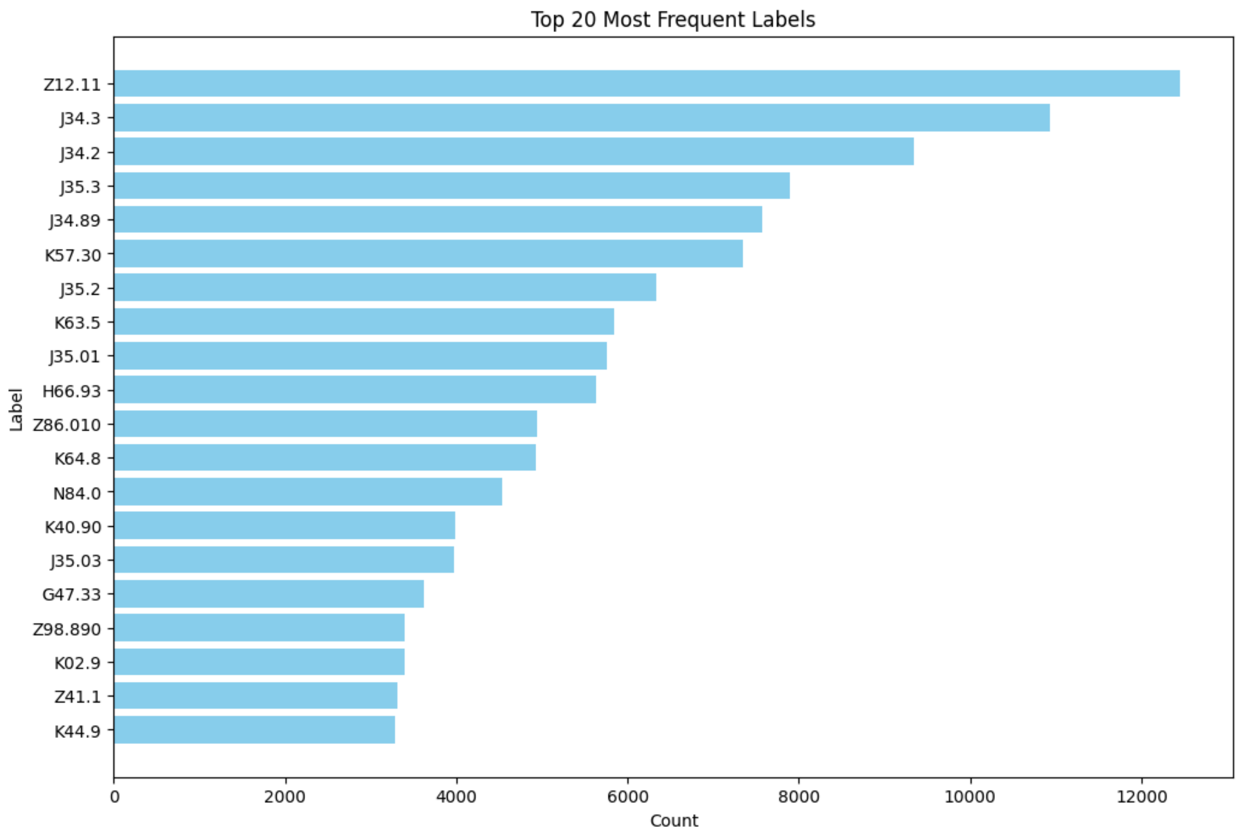
Training Data embeddings Shape**:** `(198982, 1024)`

Label Distribution Analysis

The dataset consists of multi-label data, where each data point can be associated with multiple labels. The first step was to aggregate and analyze the labels to identify common patterns.

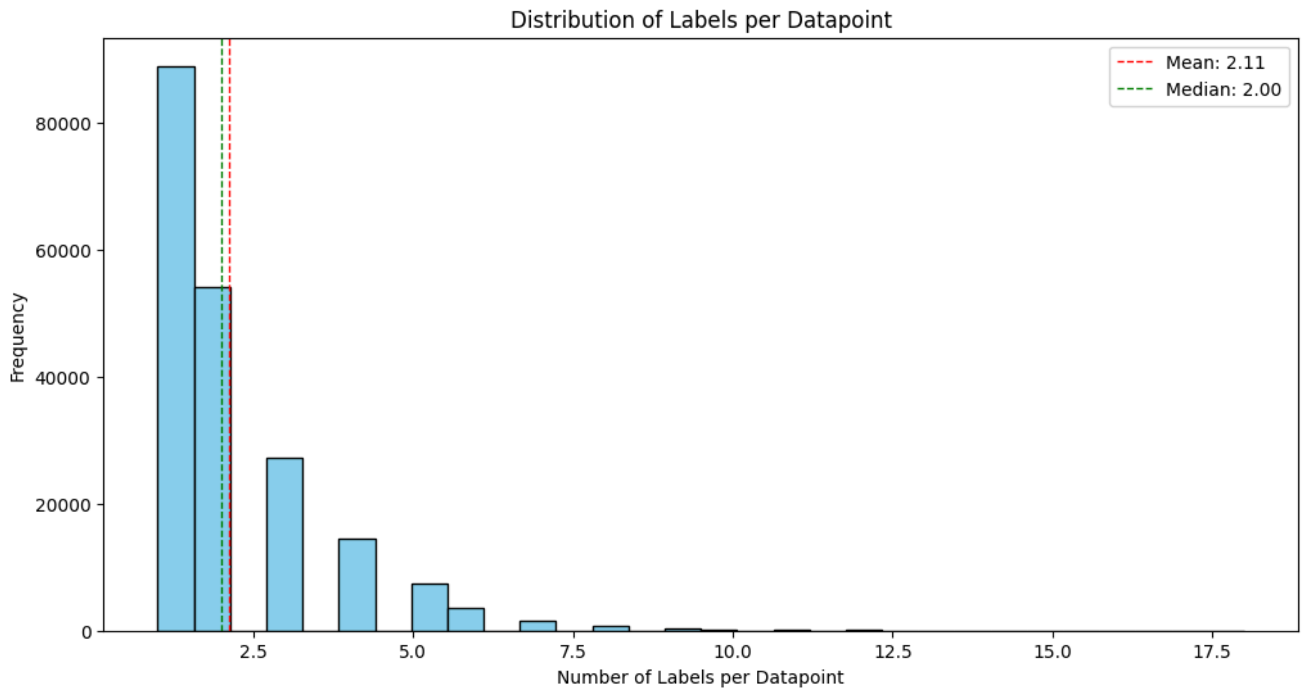
- **Label Frequency**: The frequency of each label across the dataset was analyzed using a counter. The top 20 most frequent labels were plotted to visualize their distribution.

  The following plot shows the 20 most frequent labels, with the largest counts appearing at the top:



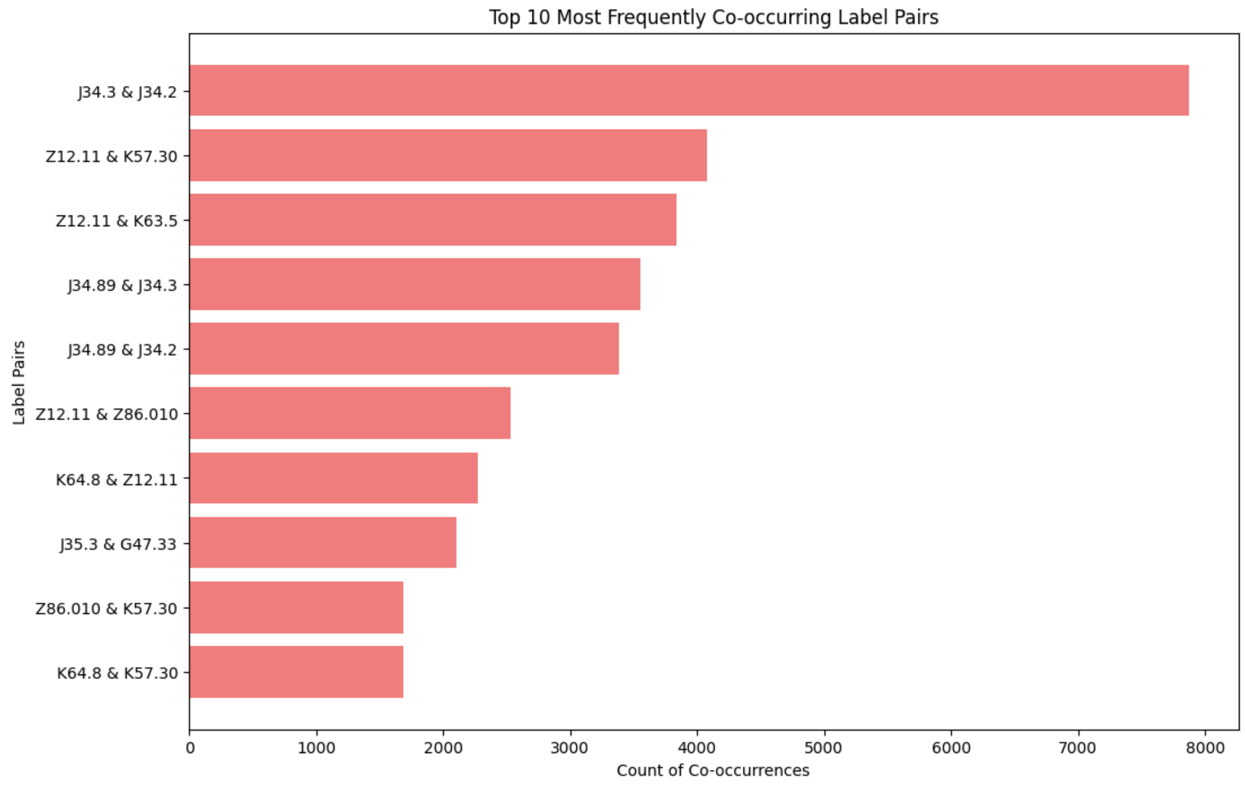
- **Number of Labels per Data Point**: Each data point in the dataset could have one or more associated labels. The distribution of the number of labels per data point was analyzed using a histogram.
    - **Mean number of labels per datapoint**: The mean value of labels per data point is approximately 2.11.
    - **Median number of labels per datapoint**: The median value is 2.00
    - **Standard deviation**: The standard deviation of labels per datapoint is 1.44

  The histogram below shows the distribution of labels across data points, with vertical dashed lines indicating the mean and median values:



- **Label Co-occurrence**: I analyzed co-occurrences of label pairs. Using the combinations of labels, I identified the top 10 most frequently co-occurring label pairs. This analysis helps to identify which labels often appear together, which could indicate a strong relationship between certain ICD-10 codes.

The bar plot below shows the top 10 most common co-occurring label pairs:

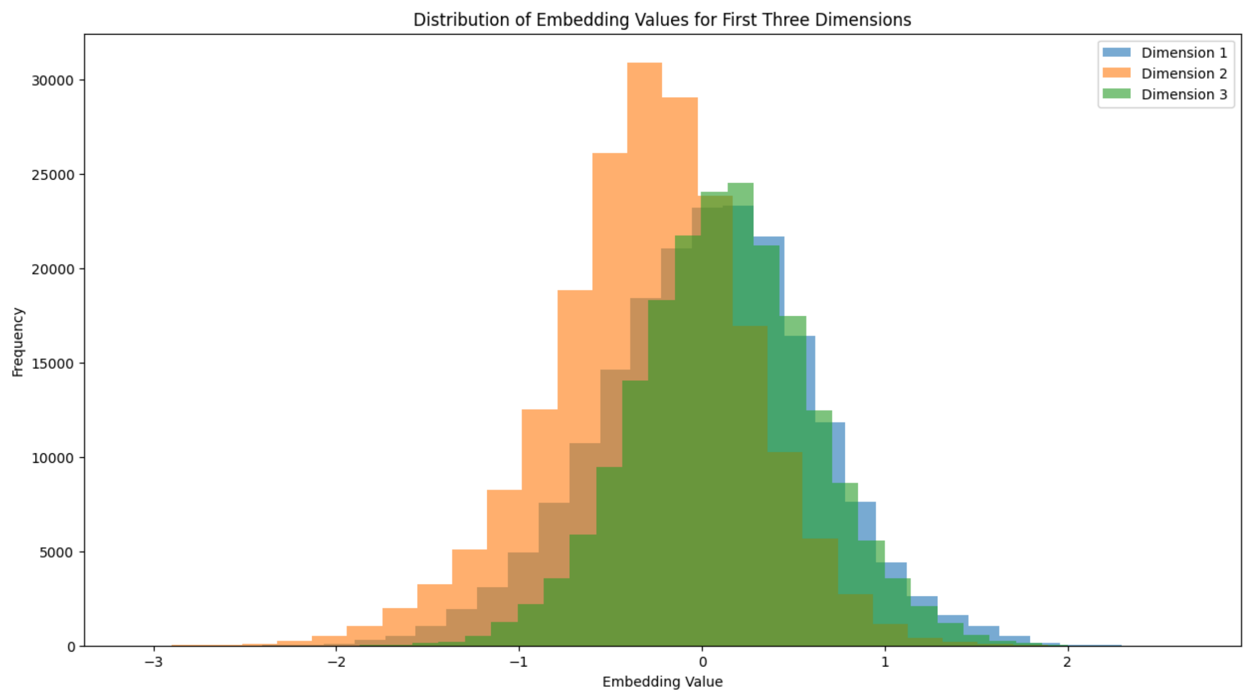


Embedding Analysis

The next step was to investigate the distribution of embedding values and the structure of the embedding space.

- **Embedding Distribution**: The first three dimensions of the embedding vectors were plotted to visualize the distribution of values across data points. The histograms below show the frequency distribution of values in each of the first three dimensions of the embeddings:

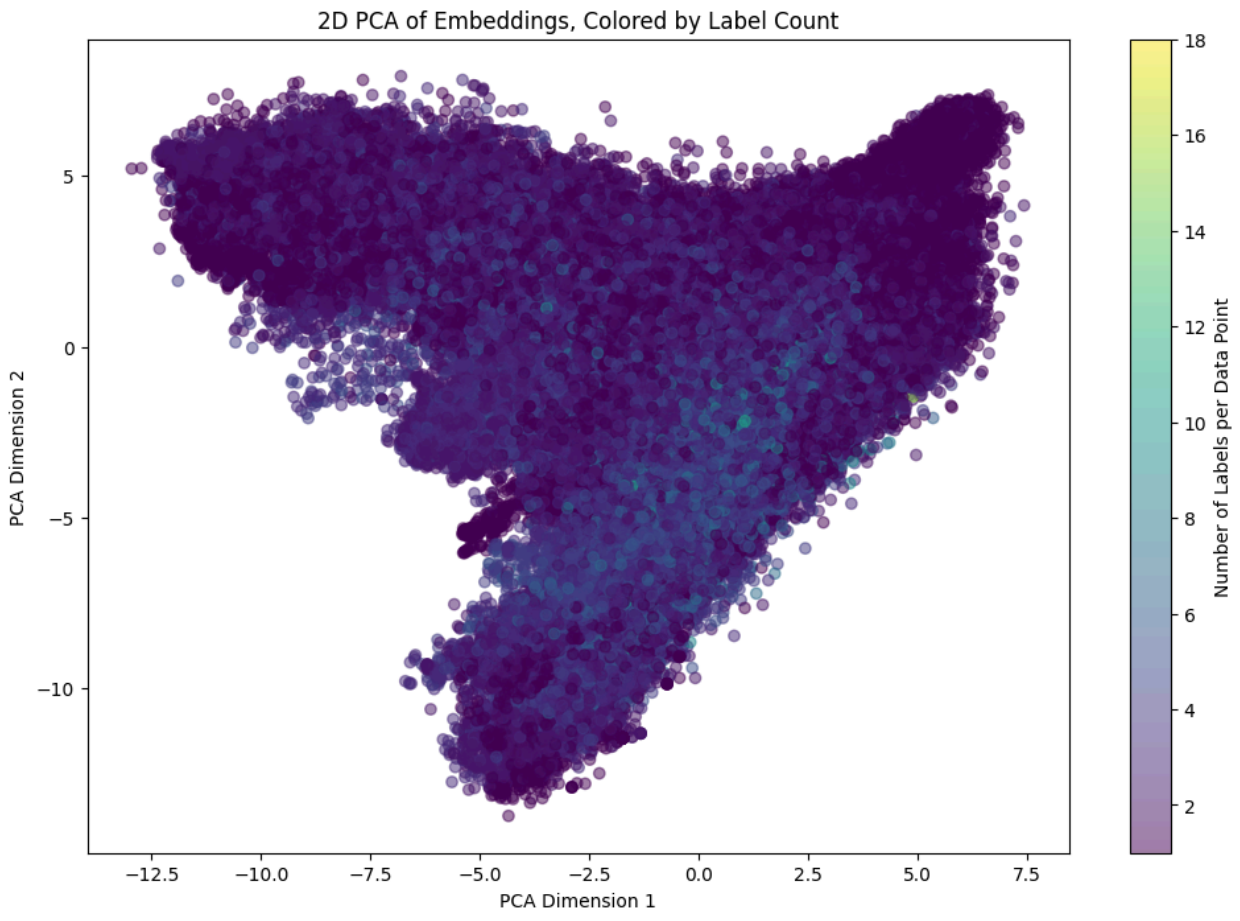
  This plot helps to understand the range and distribution of the embedding values, giving insight into how the data points are represented in the embedding space.



- **Dimensionality Reduction**: To visualize the relationships between data points in a lower-dimensional space, Principal Component Analysis (PCA) was used to reduce the embeddings from their original high-dimensional space to 2D.

  The resulting 2D plot was color-coded based on the number of labels per data point, providing insight into how the label density correlates with the positioning of data points in the embedding space.

The scatter plot below illustrates this:



The EDA revealed several key insights:

- There are highly frequent labels that dominate the dataset, which could be important when designing models.
- Label co-occurrence analysis suggests strong relationships between certain labels, which may indicate how labels are often grouped.
- The distribution of embedding values, reveals potential groupings in the dataset.

Data Engineering and Preprocessing

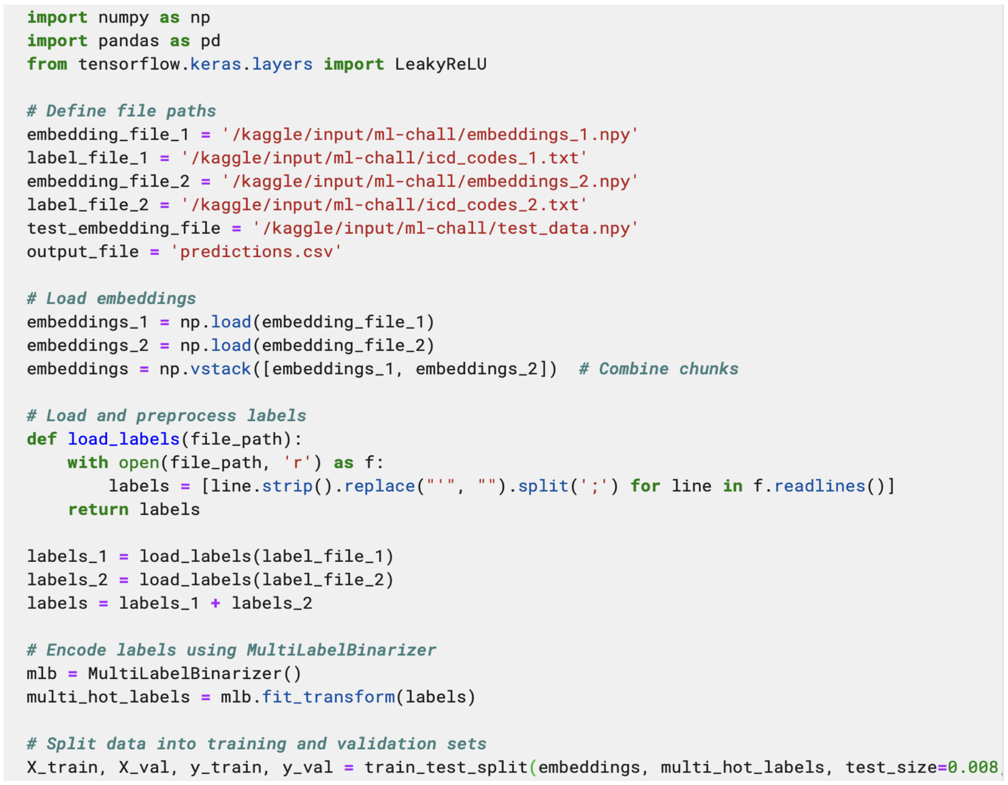
- **Data Loading**: Two embedding files (embeddings_1.npy and embeddings_2.npy) containing LLM embeddings and two corresponding label files (icd_codes_1.txt and icd_codes_2.txt). These files were combined to create the full training dataset.

- **Label Processing**: Labels were processed by splitting the ICD10 codes (separated by ;), creating a multi-label format.

- I then applied, **MultiLabelBinarizer** to encode each unique ICD10 code as a one-hot encoded vector, which allowed our model to output predictions for each code independently.

- **Data Splitting**: To validate model performance, I splitted the data into training and validation sets (99.2% training, 0.8% validation). To prevent model from underfitting on this huge training set, a large proportion of dataset was used for training the model.

Code snippet :-



```
import numpy as np
import pandas as pd
from tensorflow.keras.layers import LeakyReLU

# Define file paths
embedding_file_1 = '/kaggle/input/ml-chall/embeddings_1.npy'
label_file_1 = '/kaggle/input/ml-chall/icd_codes_1.txt'
embedding_file_2 = '/kaggle/input/ml-chall/embeddings_2.npy'
label_file_2 = '/kaggle/input/ml-chall/icd_codes_2.txt'
test_embedding_file = '/kaggle/input/ml-chall/test_data.npy'
output_file = 'predictions.csv'

# Load embeddings
embeddings_1 = np.load(embedding_file_1)
embeddings_2 = np.load(embedding_file_2)
embeddings = np.vstack([embeddings_1, embeddings_2])  # Combine chunks

# Load and preprocess labels
def load_labels(file_path):
    with open(file_path, 'r') as f:
        labels = [line.strip().replace("'", "").split(';') for line in f.readlines()]
    return labels

labels_1 = load_labels(label_file_1)
labels_2 = load_labels(label_file_2)
labels = labels_1 + labels_2

# Encode labels using MultiLabelBinarizer
mlb = MultiLabelBinarizer()
multi_hot_labels = mlb.fit_transform(labels)

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(embeddings, multi_hot_labels, test_size=0.008
```

Model Selection and Architecture

Initially, I experimented with classical machine learning algorithms such as **Logistic Regression** and **Support Vector Machines (SVM)**. While these methods have proven effective for many tasks, they encountered limitations with this high-dimensional, multi-label problem. Specifically, the vast label space and high dimensionality of the embedding data made it challenging for these models to generalize effectively, leading to poor performance. Additionally, multi-label classification requires a model capable of capturing complex dependencies among labels, which is something classical ML models struggle with, especially without extensive feature engineering.

Transition to Neural Networks

Due to these challenges, I switched to a **neural network (NN)** approach to better handle the multi-label nature of the task and the complex relationships in the embedding space. A neural network allows for flexibility in capturing non-linear relationships in the data, which is essential for improving classification accuracy on complex, high-dimensional tasks. I experimented with several architectures to find an optimal balance between model complexity, performance, and generalizability. Key changes in network architecture impacted performance significantly, and each iteration was informed by evaluating the model's **micro F2 score** (discussed in optimization section), which aligned closely with the project goals.

Architecture Selection Process

To identify the most effective architecture, I tested different neural network structures by varying the number of layers, neurons, dropout rates, and activation functions. Initial configurations were simpler, with fewer neurons and shallower depth, but these did not capture the label associations well, resulting in underfitting. I then gradually increased the model complexity by introducing more neurons and layers, ultimately selecting an architecture with two dense layers: one with 1024 neurons and another with 700 neurons. These sizes were chosen after several trials, as they allowed for the best trade-off between computational efficiency and performance improvement.

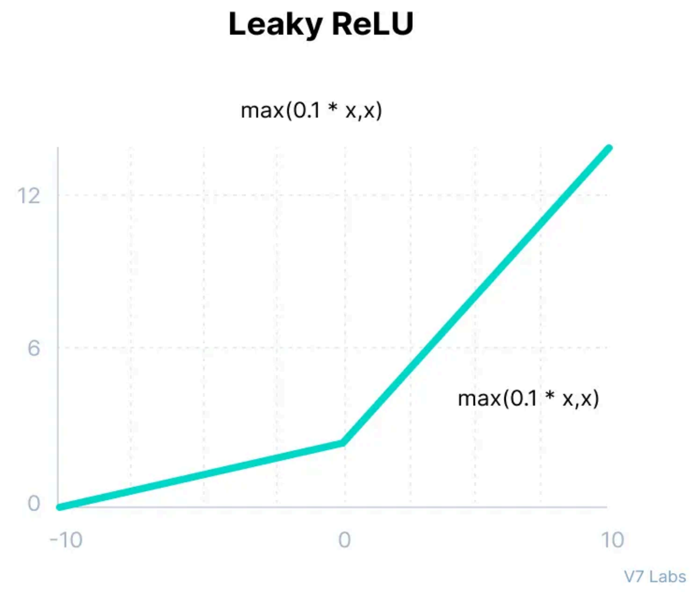
Each dense layer is followed by **batch normalization** and **dropout**. Batch normalization helped the model stabilize and accelerate training by reducing internal covariate shifts. Dropout, with a rate of 0.5, was applied to reduce the risk of overfitting, particularly because the model has a high number of parameters. These settings helped the network generalize better on unseen data without sacrificing too much training accuracy.

Activation Function Selection

Choosing the right activation function was crucial to enhance model performance, especially since multi-label classification requires effective feature transformation. I tested **tanh** and **ReLU** activations initially. However, both functions had drawbacks:

- Tanh squashed values into a narrow range, which limited the model's ability to capture variations effectively in the embedding data.
- ReLU, though successful in many applications, led to an issue known as the "dying ReLU" problem, where some neurons permanently deactivated and became less effective over time.

I finally opted for **Leaky ReLU**, which allowed a small gradient for negative inputs, preventing neurons from "dying" entirely. This choice addressed the issues faced with previous activations, leading to better stability and improved overall performance. Leaky ReLU's ability to maintain a non-zero gradient for negative inputs helped the model retain information throughout the layers, which proved beneficial in capturing nuanced patterns in multi-label classification.



For optimization, I used the **Adam optimizer** with a learning rate of 0.0001. Adam (Adaptive Moment Estimation) is an adaptive learning rate optimization algorithm combining the best properties of both AdaGrad and RMSProp.

Adam's adaptive learning rate capability allowed the model to converge faster and more effectively than other optimizers, making it well-suited for this dataset.

- **Formula**:
  - Compute the moving averages of gradients (m_t) and squared gradients (v_t):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

  - Bias-corrected versions:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

  - Update rule:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

For the loss function, I chose **binary cross-entropy**, which is commonly used in multi-label classification tasks and aligns well with the sigmoid output layer. This loss function penalizes incorrect predictions for each label individually, making it suitable for handling the presence of multiple labels in each data point.

$$\text{Binary Cross-Entropy} = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

where y_i is the true label and $\hat{y}_i$ is the predicted probability.

Final Model Configuration

(Coded in ipython notebook "**final_model.ipynb**")

The final neural network configuration consists of:

1. An **input layer** matching the embedding size (1024).
2. A **dense layer** with 1024 neurons, followed by Leaky ReLU activation, batch normalization, and dropout (0.5)
3. A **second dense layer** with 700 neurons, again followed by Leaky ReLU activation, batch normalization, and dropout (0.5).
4. A **final dense layer** with 1400 neurons and a sigmoid activation to output multi-label predictions.

This architecture achieved the best results, effectively balancing complexity and generalization ability.

Model Optimization

My primary goal in model optimization was to identify the best set of hyperparameters to maximize model performance on the micro F2 score.

I followed methods listed below to optimize model's performance:

Introducing custom F2 Score Metric:

- I designed a custom F2 score function in TensorFlow, which allowed me to use it directly as a metric for optimization.
- The F2 score prioritizes recall, which is crucial for multi-label classification, where false negatives can significantly impact performance.

$$F2\ Score = 5 \times \frac{Precision \times Recall}{(4 \times Precision) + Recall}$$

Choosing optimal validation Split:

- I initially tried several standard splits, including 0.2 and 0.1 for validation.
- Then, I observed that a smaller validation size of 0.008 worked best. A small validation set allowed for a larger training set, which was beneficial for the model to learn from more data.

  This small validation split size gave a good F2 score without sacrificing generalization, possibly due to the larger training set's representational capacity.

Dropout Rate Tuning:

- With a smaller validation set, the risk of overfitting increased as the model had access to more training data. Overfitting was visible as the model performed significantly better on training data than validation.
- I introduced a high dropout rate of 0.5 in each dense layer.

  Role of Dropout: Dropout randomly deactivates neurons during each forward pass, preventing the network from becoming too reliant on specific neurons. This promotes robustness and reduces overfitting.

  The high dropout rate helped to balance the model's performance across training and validation, mitigating overfitting.

Leaky ReLU Alpha Tuning:

- I experimented with various values of the alpha parameter (slope for negative inputs) in Leaky ReLU, starting with default values such as 0.01.
- After testing, an alpha of 0.1 yielded the best performance.

  Influence of Alpha: A higher alpha retains more information for negative values, which can improve learning for complex data distributions by avoiding the "dying ReLU" issue. Setting it too high can reduce the non-linearity, but 0.1 provided a good balance.

Batch Size and Epochs:

- Starting with a batch size of 64, I experimented up to 256. The model was sensitive to batch size, with 128 being optimal in terms of training stability and validation performance.
- Initial runs with 100 epochs showed insufficient learning, so I incrementally increased it to 150. This provided adequate training time to converge without overfitting.

  A moderate batch size like 128 ensures efficient GPU utilization and smoother gradients, while a higher epoch count allows the model to fully learn complex relationships in the data.

Learning rate tuning:

- A learning rate that was too high (e.g., 0.001) caused the model to oscillate without converging.
- Setting it to 0.0001 helped to achieve a stable convergence and improved the F2 score.

  A lower learning rate allows the model to learn gradually, avoiding the risk of overshooting minima, which is essential for complex, high-dimensional data.

Network Architecture Tuning:

Layers and Neurons

- I started with one layer of 512 neurons but found it insufficient for capturing the depth of multi-label relationships.
- The best performance was achieved with two dense layers, one with 1024 neurons and another with 700 neurons.

  The multi-layer configuration provided the network with greater capacity to capture complex interactions, while the high neuron count ensured sufficient representation.

The final architecture (1024 and 700 neurons with Leaky ReLU) provided both depth and capacity to accurately capture label correlations.

<u>Random State:</u>

- I set a fixed random state of 42 to ensure reproducibility. This ensured that every run produced consistent training and validation splits, improving the reliability of my tuning results.

Summary of Optimal Hyperparameters

1. Activation Function: Leaky ReLU with \(\alpha = 0.1\)
2. Optimizer: Adam with a learning rate of 0.0001
3. Dropout Rate: 0.5
4. Batch Size: 128
5. Epochs: 150
6. Architecture: Two dense layers with 1024 and 700 neurons, respectively
7. Validation Split: 0.008
8. Random State: 42 (for reproducibility)

By incrementally tuning each parameter and analyzing its impact, I was able to find an optimal configuration that balanced training performance with generalizability.

<u>Performance of the optimized model on validation set:</u>

- Val F2 score: 0.84
- Val loss: 0.0014

<u>Performance of the optimized model on 60 % testing set (Best Kaggle Public Score):</u>

- Micro average F2 score: 0.489

To further improve my model's performance on a 60 % test set, I tried a few hacks and workarounds which are explained in the next section.

Hacks and workarounds

1. F2 Score Coefficient Adjustment:

    - Initial Challenge: My model's initial micro F2 score on test set (public score) hovered around 0.489, but I aimed for further improvement in prioritizing recall.

    - Solution: I modified the coefficients used in calculating the F2 score, changing 5 to 5.2 and 4 to 4.5. This subtle tweak potentially placed even more emphasis on recall, which is critical in multi-label classification.

    - Explanation: Increasing the weights slightly likely shifted the score calculation towards favoring recall over precision, which helped to identify more positive labels per instance. This adjustment worked well, even though its exact impact is complex to quantify. The F2 score's recall prioritization likely aligned better with the model's structure, improving the micro F2 score.

2. Ensembling with Union Aggregator:

    - Goal: To improve predictions by combining the strengths of different models' predictions.

    - Ensemble Technique: I implemented an ensemble strategy by taking the union of labels predicted by two different models, which I coded in **`union.ipynb`**.

    - Impact of Ensembling: This ensemble method helped fill in missing or ambiguous predictions by taking the union of labels, creating a more comprehensive label set for each instance. This approach proved especially useful in imputing incomplete labels in multi-label classification.

Final Results

- After implementing these hacks on my optimized Neural Networks model, my **public micro F2 score** on Kaggle (evaluated on 60% of the test set) improved from **0.489 to 0.492**.
- When the competition closed, the final **private micro F2 score** on the remaining 40% test set was **0.494**.

Conclusion: The optimized neural network and these hacks combined effectively to improve the overall model performance, helping me successfully complete this **Final End Sem Project / ML challenge**.

Potential Improvements

- **Ensemble Modeling**: To further improve performance, ensemble techniques (e.g., using multiple models with different architectures or combining model predictions) could enhance prediction robustness.
- **Label Embedding Techniques**: Using label embeddings for ICD10 codes could potentially improve classification by capturing relationships between labels.