Technical Report for DA5401-2024-ML-Challenge (Multi-Label Classification using LLM Embeddings)

- Preethi MM21B051

1. Introduction

The ICD10 classification task involves automating the assignment of diagnostic codes to medical records, formulated as a multi-label classification problem. Each record may correspond to one or more ICD10 codes. The objective was to develop a model capable of accurately predicting these labels using feature embeddings derived from pre-trained large language models (LLMs).

The dataset provided consisted of embeddings of outpatient medical charts from the surgery specialty of a hospital. Due to the high dimensionality and complex nature of these embeddings, a range of neural network architectures and optimization techniques were explored.

2. Dataset and Problem Context

The dataset used comprised:

- **Feature Embeddings**: Two files, embeddings_1.npy and embeddings_2.npy, containing 1024-dimensional embeddings.
- **Labels**: Corresponding ICD10 codes from icd_codes_1.txt and icd_codes_2.txt, covering approximately 1,400 unique ICD10 classes.
- **Data Size**: A total of around 200,000 samples were included.

The evaluation metric chosen for the competition was the average micro-F2 score, given its relevance in multi-label classification contexts.

3. Data Preprocessing

Several preprocessing steps were applied to ensure data quality and address potential issues:

- **Data Loading and Merging**: Embeddings and labels from both files were combined into a unified dataset.
- **Handling Class Imbalance**: Oversampling of minority classes and class weighting in the loss function were employed to mitigate the effects of imbalance.
- **Label Binarization**: The ICD10 codes were converted into a multi-hot encoded format using MultiLabelBinarizer.

4. Model Development and Experimentation

A range of neural network architectures and optimization techniques were tested to identify the best-performing model.

4.1. Baseline Models

Initial experiments focused on simple neural networks with limited depth. These models set a baseline but showed limited success, indicating the need for more complex architectures.

4.2. Advanced Models

Deep neural networks with varied configurations were implemented. Different optimizers, dropout rates, and learning rates were tested:

- **Models 1-15**: Included experiments with diverse combinations of dense layers, dropout rates (0.2-0.4), and optimizers (Adam, Nadam).
- **Model 16**: Achieved the highest performance, featuring dense layers of 2048, 1024, and 512 neurons, with a dropout rate of 0.3. The Adam optimizer was used with a learning rate of 0.0001.

Code for Model 16:

```
# Model 16:
model16 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model16.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005),
loss='binary_crossentropy', metrics=[tf.keras.metrics.AUC(name="AUC", multi_label=True)])
```

Key Features of Model 16:

- **Deep Architecture**: The model has a deep network structure with dense layers of sizes 2048, 1024, and 512 neurons.
- **Dropout Regularization**: Dropout layers with a rate of 0.3 were used to prevent overfitting.
- **Optimizer**: The Adam optimizer with a learning rate of 0.0001 provided stable convergence.
- **Output Layer**: A sigmoid activation was used for multi-label classification.

4.3. Further Exploration (Models 17-40)

After identifying the architecture of Model 16 as the most effective, further experimentation was conducted by adjusting additional hyperparameters to refine the model's performance.

- **Learning Rate**: A detailed investigation was carried out using learning rates of 0.00005 and 0.0001 to determine their impact on convergence and optimization stability.
- **Dropout Rate**: Variations in dropout rates, specifically 0.2 and 0.3, were explored to assess the trade-off between generalization and the risk of overfitting. Minor improvements were observed with a slightly lower dropout rate in specific scenarios.
- **Optimizer Choice**: The performance of the model was evaluated using both the Adam and Nadam optimizers. Although Adam provided strong results, Nadam offered slight enhancements in cases where faster convergence was needed due to its adaptive learning momentum.
- **Activation Function**: The Swish activation function was also implemented in several variants of the model, replacing ReLU in the hidden layers. However, while Swish demonstrated a smoother gradient flow, it did not significantly outperform the ReLU-based architecture of Model 16.
- **Dense Layer Sizes**: Dense layer configurations were tested with initial sizes of 2048 and 1024 neurons to determine the optimal capacity for learning. Larger dense layers (starting at 2048 neurons) generally led to better performance due to their ability to capture complex patterns in the high-dimensional embeddings.

These additional models (Models 17-40) were designed to systematically vary the combinations of the aforementioned parameters. Despite extensive testing, it was observed that the architecture of Model 16 remained the most robust, consistently achieving superior performance. Minor improvements were noted in specific models, particularly those with reduced dropout rates or different optimizers (e.g., Nadam). However, these gains were not substantial enough to warrant changes from the established configuration of Model 16.

The experiments confirmed the efficacy of Model 16's design, highlighting its well-balanced trade-offs between model complexity, regularization, and optimization.

4.4. Threshold Tuning

Threshold tuning was performed to improve prediction performance:

- **Per-Label Threshold Tuning**: Instead of using a fixed threshold (0.5), optimal thresholds were identified for each label individually. This strategy enhanced the micro-F2 score by better accommodating the label distributions.
- **Grid Search Method**: A grid search was conducted over threshold values (0.1 to 0.9) to find the best decision boundaries.

5. Unsuccessful Approaches

Several other approaches were attempted but did not yield improvements:

- **Custom F2 Loss Function**: A custom loss function reflecting the F2 score was defined. However, optimization issues were encountered, and the model often failed to converge due to the non-differentiable components of the loss function.

  Custom F2 Loss Function Code:

  ```
  # Custom F2 loss function
  def f2_loss(y_true, y_pred):
      y_pred = tf.cast(y_pred > 0.5, tf.float32)
      tp = tf.reduce_sum(y_true * y_pred, axis=0)
      fp = tf.reduce_sum((1 - y_true) * y_pred, axis=0)
      fn = tf.reduce_sum(y_true * (1 - y_pred), axis=0)
      f2 = (5 * tp) / (5 * tp + 4 * fn + fp + 1e-8)
      return 1 - tf.reduce_mean(f2)  # 1 - F2 to minimize loss
  ```

  The complexity of the custom F2 loss function and its sensitivity to small changes in predictions led to instability during training.

- **Recurrent Layers (LSTM, GRU)**: Attempts were made to integrate sequential layers (LSTM, GRU) to capture potential temporal patterns. These models showed signs of overfitting without substantial improvements on the validation set.

- **High Dropout Rates**: Increasing dropout beyond 0.3 resulted in underfitting and a decline in predictive performance.

- **Aggressive Learning Rate Scheduling**: Applying aggressive learning rate decay with ReduceLROnPlateau led to premature convergence, hindering the model's ability to optimize effectively.

6. Model Evaluation

The final model, **Model 16**, demonstrated the best performance:

- **Training Micro-F2 Score**: 0.768
- **Validation Micro-F2 Score**: 0.752

The use of threshold tuning contributed to a notable improvement of approximately 3-4% in the micro-F2 score compared to a fixed threshold.

7. Conclusion

A systematic approach was taken to solve the ICD10 classification problem using LLM embeddings. Multiple neural network architectures were explored, and the best-performing model, **Model 16**, was identified after extensive experimentation. Threshold tuning played a key role in optimizing the predictions, resulting in a strong validation micro-F2 score of 0.752.

In [ ]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.preprocessing import MultiLabelBinarizer, StandardScaler
from tensorflow.keras import layers, models

# Load embeddings and labels
embeddings_1 = np.load('embeddings_1.npy')
embeddings_2 = np.load('embeddings_2.npy')
labels_1 = open('icd_codes_1.txt').read().splitlines()
labels_2 = open('icd_codes_2.txt').read().splitlines()

# Combine embeddings and labels
embeddings = np.concatenate([embeddings_1, embeddings_2], axis=0)
labels = labels_1 + labels_2

# scaler=StandardScaler()
# embeddings=scaler.fit_transform(embeddings)

# Extract unique ICD10 codes and binarize labels
all_labels = [set(l.split(';')) for l in labels]
mlb = MultiLabelBinarizer()
multi_hot_labels = mlb.fit_transform(all_labels)

# Check number of unique codes (should match ~1400)
assert multi_hot_labels.shape[1] == len(mlb.classes_)

# Split data for training/validation (80-20 split)
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(embeddings, multi_hot_labels, tes
```

In [ ]:
```
print(X_train.shape)
test_data = np.load('test_data.npy')
```

In [ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers

# Common input shape and output classes
input_shape = (1024,)
num_classes = len(mlb.classes_)
fitted_models = {}

# Model 17
model17 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='swish', kernel_regularizer=regularizers.l2(1e-4)
    layers.Dropout(0.3),
    layers.Dense(1024, activation='swish', kernel_regularizer=regularizers.l2(1e-4)
    layers.Dropout(0.3),
    layers.Dense(512, activation='swish'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
```

```
model17.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 18
model18 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='selu'),
    layers.AlphaDropout(0.2),
    layers.Dense(1024, activation='selu'),
    layers.AlphaDropout(0.2),
    layers.Dense(512, activation='selu'),
    layers.AlphaDropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model18.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 19
model19 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model19.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.0001, decay=1

# Model 20
model20 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.1),
    layers.Dropout(0.3),
    layers.Dense(1024),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.1),
    layers.Dropout(0.3),
    layers.Dense(512),
    layers.BatchNormalization(),
    layers.LeakyReLU(alpha=0.1),
    layers.Dense(num_classes, activation='sigmoid')
])
model20.compile(optimizer='nadam', loss='binary_crossentropy', metrics=[tf.keras.me

# Model 21
model21 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
```

```
])
model21.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.0001), loss='

# Model 22
model22 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu', kernel_regularizer=regularizers.l2(1e-4))
    layers.Dropout(0.3),
    layers.Dense(1024, activation='relu', kernel_regularizer=regularizers.l2(1e-4))
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(1e-4)),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dense(num_classes, activation='sigmoid')
])
model22.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 23
model23 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='swish'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='swish'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model23.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 24
model24 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(num_classes, activation='sigmoid')
])
model24.compile(optimizer='nadam', loss='binary_crossentropy', metrics=[tf.keras.me

# List of models and their names
models_list = [
    # (model19, 'model19'),
    (model20, 'model20'), (model21, 'model21'), (model22, 'model22'),
    (model23, 'model23'), (model24, 'model24')
]

# Dictionary to store fitted models
fitted_models = {}

# Loop to train and store each fitted model
for model, name in models_list:
    print(f"Training {name}")
```

```
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20
    fitted_models[name] = model  # Store the fitted model
    # Generate predictions on the test data
    preds = model.predict(test_data)
    # model_name="model16"
    pred_labels = (preds >= 0.5).astype(int)

    # Decode multi-hot predictions back to ICD10 codes
    submission = []
    for pred in pred_labels:
        codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
        codes.sort()  # Sort lexicographically
        label_string = ';'.join(codes).upper()  # Uppercase and format as required
        submission.append(label_string)

    # Generate sequential IDs (e.g., 1 to number of test samples)
    num_test_samples = len(pred_labels)
    ids = range(1, num_test_samples + 1)

    # Create the submission DataFrame
    submission_df = pd.DataFrame({'id': ids, 'labels': submission})

    # Save the submission file
    submission_filename = f'submission_{name}.csv'
    submission_df.to_csv(submission_filename, index=False)
    print(f"Saved {submission_filename}")
    print(f"Fitted model {name} saved.")

# Access any fitted model using fitted_models['model17'], fitted_models['model18'],
```

In [ ]:

In [ ]:

```
import tensorflow as tf
from tensorflow.keras import layers, models

# Common input shape and output classes
input_shape = (1024,)
num_classes = len(mlb.classes_)

# Define 16 models with explicit configurations

# Model 25: lr=0.00005, dropout=0.2, dense_size=2048, optimizer=adam
model25 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model25.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 26: lr=0.00005, dropout=0.2, dense_size=2048, optimizer=nadam
```

```
model26 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model26.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.00005), loss='b

# Model 27: lr=0.00005, dropout=0.2, dense_size=1024, optimizer=adam
model27 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model27.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 28: lr=0.00005, dropout=0.2, dense_size=1024, optimizer=nadam
model28 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model28.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.00005), loss='b

# Model 29: lr=0.00005, dropout=0.3, dense_size=2048, optimizer=adam
model29 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model29.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 30: lr=0.00005, dropout=0.3, dense_size=2048, optimizer=nadam
model30 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.3),
```

```
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model30.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.00005), loss='b

# Model 31: lr=0.00005, dropout=0.3, dense_size=1024, optimizer=adam
model31 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model31.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# Model 32: lr=0.00005, dropout=0.3, dense_size=1024, optimizer=nadam
model32 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model32.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.00005), loss='b

# Model 33: lr=0.0001, dropout=0.2, dense_size=2048, optimizer=adam
model33 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model33.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='bin

# Model 34: lr=0.0001, dropout=0.2, dense_size=2048, optimizer=nadam
model34 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
```

```
    layers.Dense(num_classes, activation='sigmoid')
])
model34.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.0001), loss='bi

# Model 35: lr=0.0001, dropout=0.2, dense_size=1024, optimizer=adam
model35 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model35.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='bin

# Model 36: lr=0.0001, dropout=0.2, dense_size=1024, optimizer=nadam
model36 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model36.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.0001), loss='bi

# Model 37: lr=0.0001, dropout=0.3, dense_size=2048, optimizer=adam
model37 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model37.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='bin

# Model 38: lr=0.0001, dropout=0.3, dense_size=2048, optimizer=nadam
model38 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model38.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.0001), loss='bi
```

```
# Model 39: lr=0.0001, dropout=0.3, dense_size=1024, optimizer=adam
model39 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model39.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='bin

# Model 40: lr=0.0001, dropout=0.3, dense_size=1024, optimizer=nadam
model40 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model40.compile(optimizer=tf.keras.optimizers.Nadam(learning_rate=0.0001), loss='bi


# List of models and their names
models_list = [
    # (model25, 'model25'), (model26, 'model26'), (model27, 'model27'),
    # (model28, 'model28'), (model29, 'model29'), (model30, 'model30'),
    # (model31, 'model31'), (model32, 'model32'), (model33, 'model33'),
    # (model34, 'model34'),
    (model35, 'model35'), (model36, 'model36'),
    (model37, 'model37'), (model38, 'model38'), (model39, 'model39'),
    (model40, 'model40')
]
fitted_models = {}

# Loop to train and store each fitted model
for model, name in models_list:
    print(f"Training {name}")
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20
    fitted_models[name] = model  # Store the fitted model
    # Generate predictions on the test data
    preds = model.predict(test_data)
    # model_name="model16"
    pred_labels = (preds >= 0.5).astype(int)

    # Decode multi-hot predictions back to ICD10 codes
    submission = []
    for pred in pred_labels:
        codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
        codes.sort()  # Sort lexicographically
        label_string = ';'.join(codes).upper()  # Uppercase and format as required
        submission.append(label_string)
```

```
    # Generate sequential IDs (e.g., 1 to number of test samples)
    num_test_samples = len(pred_labels)
    ids = range(1, num_test_samples + 1)

    # Create the submission DataFrame
    submission_df = pd.DataFrame({'id': ids, 'labels': submission})

    # Save the submission file
    submission_filename = f'submission_{name}.csv'
    submission_df.to_csv(submission_filename, index=False)
    print(f"Saved {submission_filename}")
    print(f"Fitted model {name} saved.")
```

In [ ]:
```
model = fitted_models['model37']
preds = model.predict(test_data)
name="model37"
# model_name="model16"
pred_labels = (preds >= 0.6).astype(int)

# Decode multi-hot predictions back to ICD10 codes
submission = []
for pred in pred_labels:
    codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
    codes.sort()  # Sort lexicographically
    label_string = ';'.join(codes).upper()  # Uppercase and format as required
    submission.append(label_string)

# Generate sequential IDs (e.g., 1 to number of test samples)
num_test_samples = len(pred_labels)
ids = range(1, num_test_samples + 1)

# Create the submission DataFrame
submission_df = pd.DataFrame({'id': ids, 'labels': submission})

# Save the submission file
submission_filename = f'submission2_{name}.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"Saved {submission_filename}")
print(f"Fitted model {name} saved.")
```

In [ ]:

In [ ]:
```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers

# Define a common input shape
input_shape = (1024,)
num_classes = len(mlb.classes_)

# Model 1: Baseline Dense Network with Dropout and L2 Regularization
model1 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.001))
```

```
    layers.BatchNormalization(),
    layers.Dropout(0.2),
    layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.001))
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model1.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.metr

# Model 2: Deep Feedforward Network with Skip Connections
input_layer = layers.Input(shape=input_shape)
x = layers.Dense(512, activation='relu')(input_layer)
x = layers.Dropout(0.3)(x)
x = layers.Dense(256, activation='relu')(x)
skip = layers.Concatenate()([input_layer, x])
x = layers.Dense(128, activation='relu')(skip)
x = layers.Dropout(0.3)(x)
output_layer = layers.Dense(num_classes, activation='sigmoid')(x)
model2 = models.Model(inputs=input_layer, outputs=output_layer)
model2.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[tf.keras.m

# Model 3: Wide and Deep Network
input_layer = layers.Input(shape=input_shape)
wide = layers.Dense(256, activation='relu')(input_layer)
deep = layers.Dense(512, activation='relu')(input_layer)
deep = layers.Dropout(0.3)(deep)
deep = layers.Dense(256, activation='relu')(deep)
combined = layers.Concatenate()([wide, deep])
x = layers.Dense(128, activation='relu')(combined)
output_layer = layers.Dense(num_classes, activation='sigmoid')(x)
model3 = models.Model(inputs=input_layer, outputs=output_layer)
model3.compile(optimizer='adam', loss='focal_loss', metrics=[tf.keras.metrics.AUC(n

# Model 4: LSTM-based Network
model4 = models.Sequential([
    layers.Reshape((32, 32), input_shape=input_shape),
    layers.LSTM(128, return_sequences=True),
    layers.LSTM(64),
    layers.Dropout(0.3),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model4.compile(optimizer='nadam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 5: Transformer-based Model
input_layer = layers.Input(shape=input_shape)
x = layers.Reshape((32, 32))(input_layer)
transformer_block = layers.MultiHeadAttention(num_heads=4, key_dim=32)(x, x)
x = layers.GlobalAveragePooling1D()(transformer_block)
x = layers.Dense(128, activation='relu')(x)
x = layers.Dropout(0.3)(x)
output_layer = layers.Dense(num_classes, activation='sigmoid')(x)
model5 = models.Model(inputs=input_layer, outputs=output_layer)
model5.compile(optimizer='adamw', loss='binary_focal_crossentropy', metrics=[tf.ker

# Model 6: 1D CNN-based Network
```

```
model6 = models.Sequential([
    layers.Reshape((32, 32), input_shape=input_shape),
    layers.Conv1D(64, kernel_size=3, activation='relu'),
    layers.MaxPooling1D(pool_size=2),
    layers.Conv1D(128, kernel_size=3, activation='relu'),
    layers.GlobalMaxPooling1D(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model6.compile(optimizer='sgd', loss='hinge', metrics=[tf.keras.metrics.AUC(name="A

# Model 7: Shallow Network with RMSProp Optimizer
model7 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='sigmoid')
])
model7.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=[tf.keras.m

# Model 8: Deep Neural Network with Learning Rate Decay
model8 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    # layers.Dense(128, activation='relu'),
    # layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model8.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001, decay=1e-6)

# layers were initially 3 relu, metric doesn't matter, learning rate, decay, prepro

# Model 9: Ensemble (Voting Classifier using Averaged Predictions)
def ensemble_predict(X):
    preds1 = model1.predict(X)
    preds2 = model2.predict(X)
    preds3 = model3.predict(X)
    preds4 = model4.predict(X)
    preds5 = model5.predict(X)
    preds6 = model6.predict(X)
    preds7 = model7.predict(X)
    preds8 = model8.predict(X)
    return (preds1 + preds2 + preds3 + preds4 + preds5 + preds6 + preds7 + preds8)

# List of models
models_list = [model1, model2, model3, model4, model5, model6, model7, model8]
model_names = ["model1", "model2", "model3", "model4", "model5", "model6", "model7"
```

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers

# Common input shape and output classes
input_shape = (1024,)
num_classes = len(mlb.classes_)

# Model 9: Deeper Network with Increased Dropout
model9 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.4),
    layers.Dense(128, activation='relu'),
    layers.Dense(num_classes, activation='sigmoid')
])
model9.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001), loss='bina

# # Model 10: Residual Connections for Better Gradient Flow
# input_layer = layers.Input(shape=input_shape)
# x = layers.Dense(1024, activation='relu')(input_layer)
# x = layers.Dropout(0.3)(x)
# residual = layers.Dense(512, activation='relu')(x)
# x = layers.Add()([x, residual])
# x = layers.Dense(256, activation='relu')(x)
# x = layers.Dropout(0.3)(x)
# output_layer = layers.Dense(num_classes, activation='sigmoid')(x)
# model10 = models.Model(inputs=input_layer, outputs=output_layer)
# model10.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.m

# Model 10: Residual Connections with Shape Matching
input_layer = layers.Input(shape=input_shape)
x = layers.Dense(1024, activation='relu')(input_layer)
x = layers.Dropout(0.3)(x)

# Residual branch
residual = layers.Dense(1024, activation='relu')(x)  # Match shape to 1024

# Add residual connection
x = layers.Add()([x, residual])
x = layers.Dense(256, activation='relu')(x)
x = layers.Dropout(0.3)(x)

# Output layer
output_layer = layers.Dense(num_classes, activation='sigmoid')(x)
model10 = models.Model(inputs=input_layer, outputs=output_layer)

# Compile the model
model10.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met


# Model 11: Using LeakyReLU for Better Handling of Negative Activations
```

```
model11 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024),
    layers.LeakyReLU(alpha=0.1),
    layers.Dropout(0.3),
    layers.Dense(512),
    layers.LeakyReLU(alpha=0.1),
    layers.Dropout(0.3),
    layers.Dense(256),
    layers.LeakyReLU(alpha=0.1),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model11.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 12: Learning Rate Scheduler with Adam Optimizer
model12 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=0.0001,
    decay_steps=1000,
    decay_rate=0.9
)
model12.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=lr_schedule), loss

# Model 13: Weight Regularization (L1 and L2 Regularization)
model13 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=1
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l1_l2(l1=1
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model13.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 14: Swish Activation Function for Smoother Gradient Flow
model14 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='swish'),
    layers.Dropout(0.3),
    layers.Dense(512, activation='swish'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='swish'),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
```

```
])
model14.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.met

# Model 15: Using Nadam Optimizer with Batch Normalization
model15 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(1024, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),
    layers.Dense(512, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.BatchNormalization(),
    layers.Dropout(0.3),
    layers.Dense(num_classes, activation='sigmoid')
])
model15.compile(optimizer='nadam', loss='binary_crossentropy', metrics=[tf.keras.me

# Model 16:
model16 = models.Sequential([
    layers.Input(shape=input_shape),
    layers.Dense(2048, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(1024, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.2),
    layers.Dense(num_classes, activation='sigmoid')
])
model16.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.00005), loss='bi

# List of new models
new_models = [model9, model10, model11, model12, model13, model14, model15, model16

# # Training loop for new models
# for i, model in enumerate(new_models, 9):
#     print(f"Training Model {i}")
#     history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=
```

In [ ]:
```
# Load test data
test_data = np.load('test_data.npy')

# Generate predictions
threshold = 0.5  # Adjust threshold if needed based on validation performance
```

In [ ]:
```
# List of models
models_list = [model8]
new_models = [model9, model10, model11, model12, model13, model14, model15, model16
model_names = ["model9", "model10", "model11", "model12", "model13", "model14", "mo
# Training loop and submission generation
for i, (model, model_name) in enumerate(zip(models_list, model_names), 1):
    print(f"Training {model_name}")

    # Train the model
```

```
    history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20

    # Generate predictions on the test data
    preds = model.predict(test_data)
    pred_labels = (preds >= 0.3).astype(int)
```

In [ ]:
```
# List of models
models_list = [model16]
model_names = ["model16"]
# models_list = [model13, model14, model15, model16]
# model_names = ["model13", "model14", "model15", "model16"]
# Training loop and submission generation
# for i, (model, model_name) in enumerate(zip(models_list, model_names), 1):
#     print(f"Training {model_name}")

model=model16
# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=20, ba

# Generate predictions on the test data
preds = model.predict(test_data)
# model_name="model16"
pred_labels = (preds >= 0.4).astype(int)

# 0.45 --- 0.438
# 0.475 --- 0.440
# 0.49 --- 0.441
# 0.495 --- 0.442
#  0.5 --- 0.442
#  0.55 --- 0.442
#  0.57 --- 0.442
# 0.6 --- 0.441

# Decode multi-hot predictions back to ICD10 codes
submission = []
for pred in pred_labels:
    codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
    codes.sort()  # Sort lexicographically
    label_string = ';'.join(codes).upper()  # Uppercase and format as required
    submission.append(label_string)

# Generate sequential IDs (e.g., 1 to number of test samples)
num_test_samples = len(pred_labels)
ids = range(1, num_test_samples + 1)

# Create the submission DataFrame
submission_df = pd.DataFrame({'id': ids, 'labels': submission})

# Save the submission file
submission_filename = f'submission2_{model_name}.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"Saved {submission_filename}")

# Ensemble prediction example
# ensemble_predictions = ensemble_predict(test_data)
```

In [ ]:
```
model_name="model16"
# Generate predictions on the test data
preds = model.predict(test_data)
# model_name="model16"
pred_labels = (preds >= 0.44).astype(int)

# 0.45 --- 0.438
# 0.475 --- 0.440
# 0.49 --- 0.441
# 0.495 --- 0.442
#  0.5 --- 0.442
#  0.55 --- 0.442
#  0.57 --- 0.442
# 0.6 --- 0.441

# for new model16, 0.45 - 0.47 yielded 0.460

# Decode multi-hot predictions back to ICD10 codes
submission = []
for pred in pred_labels:
    codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
    codes.sort()  # Sort lexicographically
    label_string = ';'.join(codes).upper()  # Uppercase and format as required
    submission.append(label_string)

# Generate sequential IDs (e.g., 1 to number of test samples)
num_test_samples = len(pred_labels)
ids = range(1, num_test_samples + 1)

# Create the submission DataFrame
submission_df = pd.DataFrame({'id': ids, 'labels': submission})

# Save the submission file
submission_filename = f'submission2_{model_name}.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"Saved {submission_filename}")
```

In [ ]:
```
def ensemble_predict(X):
    # Get predictions from all models
    preds1 = model1.predict(X)
    preds2 = model2.predict(X)
    preds3 = model3.predict(X)
    preds4 = model4.predict(X)
    preds5 = model5.predict(X)
    preds6 = model6.predict(X)
    preds7 = model7.predict(X)
    preds8 = model8.predict(X)

    # Initialize an empty list to store final predictions
    final_predictions = []

    # Iterate through each sample's predictions and take the union of labels
    for i in range(len(preds1)):
        # Collect predictions for the current sample across all models
        sample_preds = set(preds1[i]) | set(preds2[i]) | set(preds3[i]) | set(preds
```

```
                        set(preds5[i]) | set(preds6[i]) | set(preds7[i]) | set(preds

        # Convert the set back to a list and add to final predictions
        final_predictions.append(list(sample_preds))

    return final_predictions
```

In [ ]:
```
# Generate predictions on the test data
preds = model.predict(test_data)
pred_labels = (preds >= threshold).astype(int)

# Decode multi-hot predictions back to ICD10 codes
submission = []
for pred in pred_labels:
    codes = [mlb.classes_[j] for j, val in enumerate(pred) if val == 1]
    codes.sort()  # Sort lexicographically
    label_string = ';'.join(codes).upper()  # Uppercase and format as required
    submission.append(label_string)

# Generate sequential IDs (e.g., 1 to number of test samples)
num_test_samples = len(pred_labels)
ids = range(1, num_test_samples + 1)

# Create the submission DataFrame
submission_df = pd.DataFrame({'id': ids, 'labels': submission})

# Save the submission file
submission_filename = 'submission_ensemble1.csv'
submission_df.to_csv(submission_filename, index=False)
print(f"Saved {submission_filename}")
```

In [ ]:
```
import pandas as pd

# List of submission files
submission_files = [
    'ensemble_submission3.csv',
    'submission2_model8.csv'
]

# Read all submissions into a list of DataFrames
submissions = [pd.read_csv(file) for file in submission_files]

# Initialize an empty DataFrame for the ensemble predictions
ensemble_df = pd.DataFrame()
ensemble_df['id'] = submissions[0]['id']
ensemble_predictions = []

# Iterate through each row by index
for i in range(len(submissions[0])):
    # Initialize a set to store the union of predictions for this row
    combined_predictions = set()

    # Iterate through each model's prediction for the current sample
    for submission in submissions:
        # Get the predicted labels for the current sample, handling NaN values
```

```
        labels_str = str(submission.loc[i, 'labels']).strip()
        if labels_str:  # Check if it's not an empty string
            labels = labels_str.split(';')
            # Add labels to the combined set
            combined_predictions.update(labels)

    # Convert the set back to a sorted list and join using semicolons
    ensemble_prediction = ';'.join(sorted(combined_predictions))
    ensemble_predictions.append(ensemble_prediction)

# Store the ensemble predictions in the DataFrame
ensemble_df['labels'] = ensemble_predictions

# Save the ensemble predictions to a CSV file
ensemble_df.to_csv('ensemble_submission4.csv', index=False)
```

In [ ]:

```
import pandas as pd
from collections import Counter

# List of submission files
# submission_files = [
#     'submission_model1.csv',
#     'submission_model2.csv',
#     'submission_model8.csv',
#     'submission_model4.csv',
#     'submission_model5.csv',
#     'submission_model6.csv',
#     'submission_model7.csv',
#     'submission_model8.csv'
# ]

submission_files = [
    'submission1_model8.csv',
    'submission2_model8.csv'
]

# Read all submissions into a list of DataFrames
submissions = [pd.read_csv(file) for file in submission_files]

# Initialize an empty DataFrame for the ensemble predictions
ensemble_df = pd.DataFrame()
ensemble_df['id'] = submissions[0]['id']
ensemble_predictions = []

# Iterate through each row by index
for i in range(len(submissions[0])):
    # Initialize a Counter to track the frequency of each label
    label_counter = Counter()

    # Iterate through each model's prediction for the current sample
    for submission in submissions:
        # Get the predicted labels for the current sample, handling NaN values
        labels_str = str(submission.loc[i, 'labels']).strip()
        if labels_str:  # Check if it's not an empty string
            labels = labels_str.split(';')
            # Update the counter with the labels from this prediction
```

```
            label_counter.update(labels)

    # Select only labels that appear more than twice (count > 2)
    selected_labels = [label for label, count in label_counter.items() if count > 2

    # Convert the list back to a sorted string of selected labels
    ensemble_prediction = ';'.join(sorted(selected_labels))
    ensemble_predictions.append(ensemble_prediction)

# Store the ensemble predictions in the DataFrame
ensemble_df['labels'] = ensemble_predictions

# Save the ensemble predictions to a CSV file
ensemble_df.to_csv('ensemble_submission3.csv', index=False)
```

In [ ]:
```
# Define the model
model = models.Sequential([
    layers.Input(shape=(1024,)),  # Input layer for 1024-dimensional embeddings
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(len(mlb.classes_), activation='sigmoid')  # Output layer for multi
])

# Compile the model with binary cross-entropy loss for multi-label classification
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.metri

# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, ba
```

In [ ]:
```
import tensorflow as tf
from tensorflow.keras import layers, regularizers

# Define the model
model = tf.keras.Sequential([
    layers.Input(shape=(1024,)),  # Input layer for 1024-dimensional embeddings
    layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.001))
    layers.Dropout(0.3),  # Dropout layer to prevent overfitting
    layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.001))
    layers.Dropout(0.3),  # Dropout layer
    layers.Dense(1400, activation='sigmoid')  # Output layer for multi-label classi
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accura

# Train the model
history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_data=(X_val, y_val))
```

In [ ]:
```
# Load test data
test_data = np.load('test_data.npy')

# Generate predictions
preds = model.predict(test_data)
threshold = 0.5  # Adjust threshold if needed based on validation performance
pred_labels = (preds >= threshold).astype(int)
```

In [ ]:
```
# Decode multi-hot predictions back to ICD10 codes
submission = []
for pred in pred_labels:
    codes = [mlb.classes_[i] for i, val in enumerate(pred) if val == 1]
    codes.sort()  # Sort lexicographically
    label_string = ';'.join(codes).upper()  # Uppercase and format as required
    submission.append(label_string)
```

```
import pandas as pd

# Generate sequential IDs (e.g., 0 to number of test samples - 1)
num_test_samples = len(pred_labels)  # Length of the test predictions
ids = range(1, num_test_samples + 1)

# Create the submission DataFrame
submission_df = pd.DataFrame({'id': ids, 'labels': submission})

# Save the clean submission file
submission_df.to_csv('submission.csv', index=False)
```

In [ ]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers

# Custom F2 loss function
def f2_loss(y_true, y_pred):
    y_pred = tf.cast(y_pred > 0.5, tf.float32)
    tp = tf.reduce_sum(y_true * y_pred, axis=0)
    fp = tf.reduce_sum((1 - y_true) * y_pred, axis=0)
    fn = tf.reduce_sum(y_true * (1 - y_pred), axis=0)
    f2 = (5 * tp) / (5 * tp + 4 * fn + fp + 1e-8)
    return 1 - tf.reduce_mean(f2)  # 1 - F2 to minimize loss

# Model architecture with increased complexity
def create_model():
    model = models.Sequential([
        layers.Input(shape=(1024,)),
        layers.Dense(1024, activation='relu', kernel_regularizer=regularizers.l2(0.
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.0
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.0
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(1400, activation='sigmoid')  # Multi-label output
    ])
    model.compile(optimizer='adam', loss=f2_loss, metrics=['binary_accuracy'])
    return model

# Load data
embeddings_1 = np.load('embeddings_1.npy')
embeddings_2 = np.load('embeddings_2.npy')
labels_1 = pd.read_csv('icd_codes_1.txt', header=None)
labels_2 = pd.read_csv('icd_codes_2.txt', header=None)
test_embeddings = np.load('test_data.npy')

# Combine embeddings and labels for training
```

```
X_train = np.vstack([embeddings_1, embeddings_2])
y_train = pd.concat([labels_1, labels_2], ignore_index=True)

# Convert labels to multi-hot encoding
unique_labels = sorted(set(";".join(y_train[0].values).split(";")))
label_index = {label: i for i, label in enumerate(unique_labels)}

def labels_to_multi_hot(labels, label_index):
    multi_hot = np.zeros((len(labels), len(label_index)), dtype=int)
    for i, label_str in enumerate(labels):
        for label in label_str.split(";"):
            if label in label_index:
                multi_hot[i, label_index[label]] = 1
    return multi_hot

y_train_multi_hot = labels_to_multi_hot(y_train[0], label_index)

# Split data for validation (e.g., 80-20 split)
split_idx = int(0.8 * len(X_train))
X_val, y_val = X_train[split_idx:], y_train_multi_hot[split_idx:]
X_train, y_train = X_train[:split_idx], y_train_multi_hot[:split_idx]

# Create and train the model
model = create_model()
history = model.fit(X_train, y_train, epochs=20, batch_size=128, validation_data=(X
```

In [ ]:

```
from sklearn.metrics import f1_score
import random

# Select a random sample from the validation set
sample_size = 1000  # Adjust based on memory capacity
sample_indices = random.sample(range(len(X_val)), sample_size)
X_val_sample = X_val[sample_indices]
y_val_sample = y_val[sample_indices]

# Predict on the sample
val_preds_sample = model.predict(X_val_sample)

import numpy as np
from sklearn.metrics import fbeta_score

# Penalize higher thresholds and limit threshold search range
thresholds = np.arange(0.1, 0.5, 0.05)  # Restrict to lower values
best_thresholds = []

for i in range(y_val_sample.shape[1]):
    f2_scores = []
    for thresh in thresholds:
        preds = (val_preds_sample[:, i] > thresh).astype(int)

        # Calculate micro-F2 score
        f2 = fbeta_score(y_val_sample[:, i], preds, beta=2, average='micro')

        # Apply a penalty for higher thresholds (example penalty: subtract a factor
        penalty = 0.01 * (thresh - 0.3) if thresh > 0.3 else 0
        f2_scores.append(f2 - penalty)
```

```
    best_thresh = thresholds[np.argmax(f2_scores)]
    best_thresholds.append(best_thresh)
```

In [ ]:
```
# Predictions on test data using a fixed threshold of 0.5
test_preds = model.predict(test_embeddings)
```

In [ ]:
```
print(best_thresholds)
```

In [ ]:
```
from skopt import gp_minimize
from skopt.space import Real
from sklearn.metrics import fbeta_score

# Define function to maximize F2 score
def f2_threshold_objective(thresh_values):
    # Convert array of threshold values to predictions
    preds = (val_preds_sample > np.array(thresh_values)).astype(int)
    micro_f2 = fbeta_score(y_val_sample, preds, beta=2, average='micro')
    return -micro_f2  # Negative because we are minimizing in Bayesian Optimization

# Define search space for thresholds (0.1 to 0.5 for each label)
space = [Real(0.1, 0.5, name=f'thresh_{i}') for i in range(y_val_sample.shape[1])]

# Run Bayesian optimization
opt_result = gp_minimize(f2_threshold_objective, space, n_calls=20, random_state=0)

# Extract best thresholds
best_thresholds = opt_result.x
print(best_thresholds)
```

In [ ]:
```
# test_labels = []
# for i in range(test_preds.shape[0]):
#     labels = [unique_labels[j] for j in range(test_preds.shape[1]) if test_preds[
#     test_labels.append(";".join(sorted(labels)))

# Predictions on test data using a fixed threshold of 0.5
test_preds = model.predict(test_embeddings)
test_labels = []
for i in range(test_preds.shape[0]):
    labels = [unique_labels[j] for j in range(test_preds.shape[1]) if test_preds[i,
    test_labels.append(";".join(sorted(labels)))

# Create submission dataframe
submission_df = pd.DataFrame({'id': range(1, len(test_labels) + 1), 'labels': test_

# Save submission file
submission_df.to_csv('submission.csv', index=False)
```

In [ ]:
```
# Define the model
model = models.Sequential([
    layers.Input(shape=(1024,)),  # Input layer for 1024-dimensional embeddings
    layers.Dense(512, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(256, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(len(mlb.classes_), activation='sigmoid')  # Output layer for multi
])

# Compile the model with binary cross-entropy loss for multi-label classification
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=[tf.keras.metri

# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, ba
```

In [ ]:
```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers, models, regularizers

# Custom F2 loss function
def f2_loss(y_true, y_pred):
    y_pred = tf.cast(y_pred > 0.5, tf.float32)
    tp = tf.reduce_sum(y_true * y_pred, axis=0)
    fp = tf.reduce_sum((1 - y_true) * y_pred, axis=0)
    fn = tf.reduce_sum(y_true * (1 - y_pred), axis=0)
    f2 = (5 * tp) / (5 * tp + 4 * fn + fp + 1e-8)
    return 1 - tf.reduce_mean(f2)  # 1 - F2 to minimize loss

# Model architecture with increased complexity
def create_model():
    model = models.Sequential([
        layers.Input(shape=(1024,)),
        layers.Dense(1024, activation='relu', kernel_regularizer=regularizers.l2(0.
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(512, activation='relu', kernel_regularizer=regularizers.l2(0.0
        layers.BatchNormalization(),
        layers.Dropout(0.5),

        layers.Dense(256, activation='relu', kernel_regularizer=regularizers.l2(0.0
        layers.BatchNormalization(),
        layers.Dropout(0.3),

        layers.Dense(1400, activation='sigmoid')  # Multi-label output
    ])
    model.compile(optimizer='adam', loss=f2_loss, metrics=['binary_accuracy'])
    return model

# Create the model
# model = create_model()
```

```
# Train the model
history = model.fit(X_train, y_train, validation_data=(X_val, y_val), epochs=10, ba
```