Report on Data Engineering and Model Development

Your Name

November 14, 2024

0.1 Data Engineering

The preprocessing pipeline involves the integration and transformation of large datasets. Below is the code used for data preprocessing and an explanation of each step:

```
import numpy as np
import tensorflow as tf

# Load embedding data
X = np.load('embeddings_1.npy')
X_2 = np.load('embeddings_2.npy')
X = np.concatenate((X, X_2), axis=0)

# Load label data
label_data = []
file_names = ['icd_codes_1.txt', 'icd_codes_2.txt']
for file_name in file_names:
    with open(file_name, 'r') as file:
        label_data.extend(line.strip() for line in file if
           line.strip())

# Create unique vocabulary for StringLookup
unique_codes = set()
for labels in label_data:
    unique_codes.update(labels.split(";"))
unique_codes = sorted(unique_codes)

# Initialize StringLookup
lookup_layer = tf.keras.layers.StringLookup(vocabulary=unique_codes,
   output_mode="multi_hot", mask_token=None, num_oov_indices=0)

# Process labels with tf.data.Dataset
label_data_ds = tf.data.Dataset.from_tensor_slices(label_data)
def encode_labels(labels):
    return lookup_layer(tf.strings.split(labels, sep=";"))
multi_hot_labels_ds = label_data_ds.map(encode_labels,
            num_parallel_calls=tf.data.AUTOTUNE).batch(1000)

# Concatenate results
y = tf.concat(list(multi_hot_labels_ds), axis=0)
print("Shape of y:", y.shape)
```

0.1.1 Explanation of Steps

- **Loading Embedding Data:** The embedding vectors are loaded from two files, `embeddings_1.npy` and `embeddings_2.npy`, and concatenated along the first axis to create a unified dataset. The dataset consists of 1024 features for each sample.
- **Reading Label Data:** Label data are extracted from text files `icd_codes_1.txt` and `icd_codes_2.txt`, where each line contains semicolon-separated labels.
- **Building Vocabulary:** A set of unique ICD-10 codes is created by iterating through all label sets and splitting them by the semicolon delimiter. This set is then sorted to ensure consistent encoding.

- **StringLookup Initialization:** A `StringLookup` layer is initialized with the vocabulary to map each label to a multi-hot encoded format.
- **Efficient Processing:** The `tf.data.Dataset` API is used to preprocess the label data efficiently. Each label set is split and encoded using the `StringLookup` layer, and batching is applied to optimize memory usage.
- **Concatenating Results:** The batched encoded labels are concatenated to form the final multi-hot encoded tensor `y`, which has dimensions `(200000, 1400)`, representing the number of samples and unique labels.

0.2 Feature Engineering

A `StringLookup` layer was initialized using TensorFlow to map the ICD-10 codes to a multi-hot encoded format. This layer utilized the sorted vocabulary, ensuring consistent encoding of label sets. To optimize memory usage, the label data were processed using a `tf.data.Dataset` pipeline, leveraging batching and parallel processing to handle the large-scale dataset efficiently. The final encoded labels, stored in a tensor `y`, had dimensions `(200000, 1400)`, indicating the number of samples and the total unique labels. Each embedding vector consists of 1024 features, representing the high-dimensional nature of the data.

0.3 Sampling

Train test split was used to split the training data and the validation data which is randomly done.

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
    random_state=42)

print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_val.shape)
print("y_test shape:", y_val.shape)
```

Other than this, no other samppling was done because it was not required. As we used gpu to run out neural network model, the code worked sufficiently fast and so, no sampling was used. It was done to capture every information we can.

0.4 Exploratory Data Analysis (EDA)

0.4.1 Visualization

Below is the code snippet used to generate a box plot visualization of the first 10 embedding features:

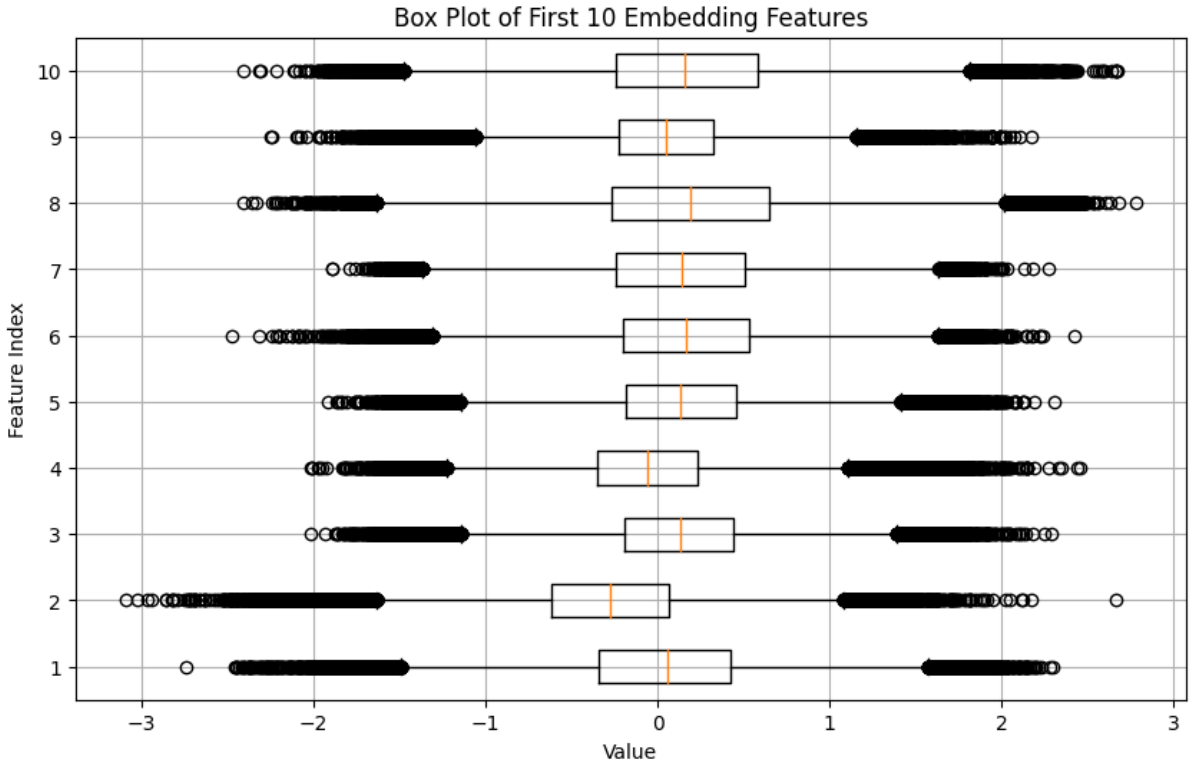
```
    import matplotlib.pyplot as plt

# Box plot for first 10 embedding features
plt.figure(figsize=(10, 6))
```

```
plt.boxplot(X[:, :10], vert=False)
plt.title("Box Plot of First 10 Embedding Features")
plt.xlabel("Value")
plt.ylabel("Feature Index")
plt.grid(True)
plt.show()
```

0.4.2 Explanation of the Visualization

The box plot displays the distribution of values for the first 10 embedding features out of a total of 1024 features. Key takeaways:

- The horizontal orientation makes it easier to compare feature distributions.
- The median, quartiles, and potential outliers for each feature are clearly visible.
- Variability in feature scales provides insights into normalization needs or model expectations.

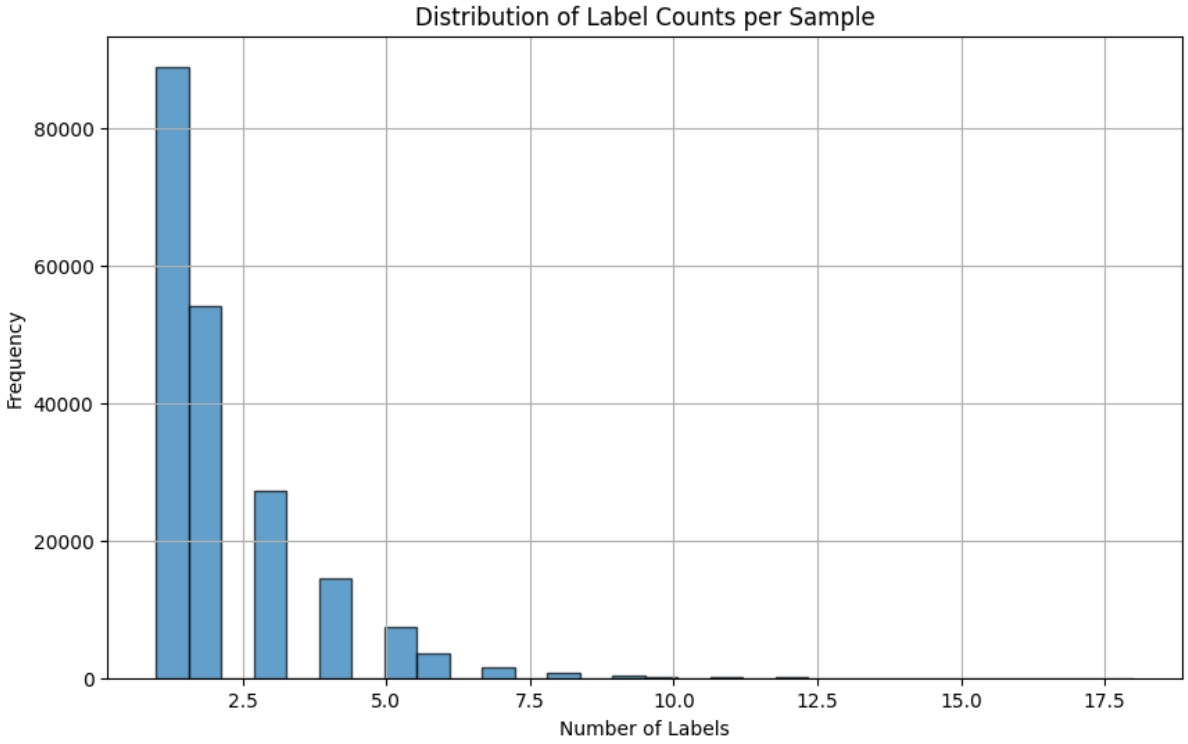


Additionally, the following visualizations provide insights into label data:

```
# Distribution of label counts per sample
plt.figure(figsize=(10, 6))
plt.hist(label_counts, bins=30, edgecolor='k', alpha=0.7)
plt.title("Distribution of Label Counts per Sample")
plt.xlabel("Number of Labels")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()

# Frequency of top 20 most common labels
plt.figure(figsize=(10, 6))
```

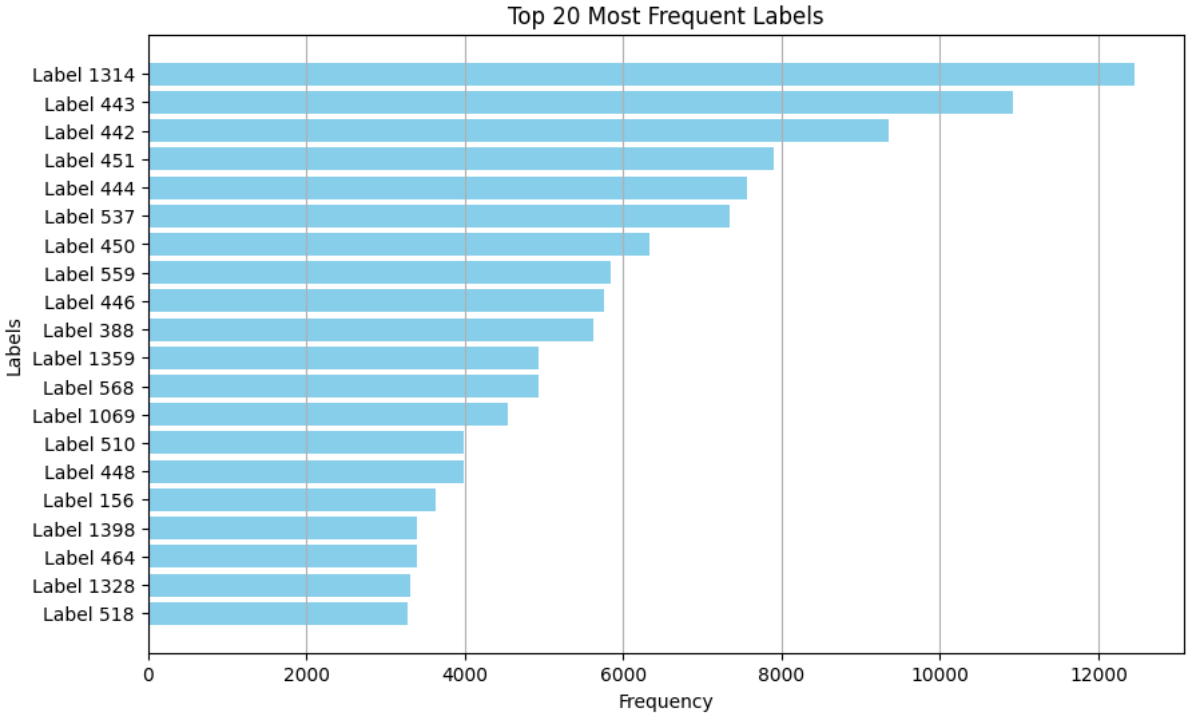
```
plt.barh(range(len(top_label_counts)), top_label_counts,
    color='skyblue')
plt.yticks(range(len(top_label_counts)), [f"Label {i}" for i in
    top_label_indices])
plt.xlabel("Frequency")
plt.ylabel("Labels")
plt.title("Top 20 Most Frequent Labels")
plt.grid(True, axis='x')
plt.show()
```

0.4.3 Distribution of Label Counts



The histogram above illustrates the distribution of label counts per sample. Most samples have a small number of associated labels, while a few have a larger number. This skewed distribution suggests that most instances are sparsely labeled.

0.4.4 Top 20 Most Frequent Labels



This visualization highlights the imbalance in label occurrences, with certain labels appearing significantly more frequently than others.

0.5 Models used

A number of models were considered for this data challenge. The first approach is to consider the mutilabel problem as a multiple binary label classification problem. This process involves training around 1400 models (one for each label) and predict the test data on these 1400 models meaning that each model will predict 0 or 1 for its corresponding label.

The binary classifiers tried for this approach were logistic regression and lightbgm model.

Listing 1: Code for creating seperate label files for multiple-binary classification

```
import pandas as pd
import os
import tensorflow as tf

# Define output folder for CSV files
output_folder = "label_files"
os.makedirs(output_folder, exist_ok=True)

# Save each column as a separate CSV file
for label_index in range(y_train.shape[1]):
    # Extract each column (label) and save as a binary CSV
    label_data = pd.DataFrame(y_train[:, label_index],
        columns=[f'label_{label_index}'])
    label_data.to_csv(f"{output_folder}/label_{label_index}.csv",
        index=False, header=False)

```

```
print("CSV files created in the 'label_files' folder.")
```

the above code will create 1400 csv files of binary output for each label. Each csv will have 0s and 1s indicating the presence of that particular label in each train datapoint.

Listing 2: Code for implenting Logreg models for each label

```
from sklearn.linear_model import LogisticRegression
from joblib import Parallel, delayed
import joblib

# Folder for saving models
model_folder = "models"
os.makedirs(model_folder, exist_ok=True)

# Function to train a model for each label
def train_model_for_label(label_index):
    # Load the binary target data for this label
    y_train = pd.read_csv(f"{output_folder}/label_{label_index}.csv",
        header=None).values.ravel()

    # Initialize and train the model
    model = LogisticRegression(max_iter=100)
    model.fit(X_train, y_train)

    # Save the model
    model_path = f"{model_folder}/model_{label_index}.pkl"
    joblib.dump(model, model_path)

    return model
# Train each model in parallel
Parallel(n_jobs=8)(delayed(train_model_for_label)(i) for i in
    range(y_train.shape[1]))

print("Training complete. Models saved in the 'models' folder.")
```

The above code trains the training dataset on logistic regression and saves 1400 models as output in a seperate folder.

We got 0.427 score (score refers to average micro f2 score) using this model. Nest, the main model for binary classification has been changed from logreg to lightbgm as we searched and found it as a better model in this scenaario. But, later we found that lightgbm model overpredicts (giving abnormally more number of labels for the test data). In fact, we got 0.000 score from using lightgbm.

From here, we switched our concentration to neural network models. Here is a general framework of the neural network model that we used:

Listing 3: sample neural network code

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
    BatchNormalization, Input, LayerNormalization
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint,
    ReduceLROnPlateau
from tensorflow.keras.optimizers import Adam
import numpy as np

# Enhanced model architecture with larger layers and increased dropout
```

```
def create_model(input_shape):
    model = Sequential([
        Input(shape=(input_shape,)),
        Dense(2048, activation='relu'),
        LayerNormalization(),
        Dropout(0.5),
        Dense(1024, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(0.4),
        Dense(256, activation='relu'),
        BatchNormalization(),
        Dropout(0.4),
        Dense(1400, activation='sigmoid')
    ])
    return model

# Initialize the model
model = create_model(X_train.shape[1])

# Compile model with Adam optimizer and custom focal loss
optimizer = Adam(learning_rate=0.0003)
model.compile(optimizer=optimizer, loss='binary_crossentropy',
    metrics=[MicroF2Score()])

# Callbacks for early stopping and learning rate reduction
early_stopping = EarlyStopping(monitor='val_loss', patience=8,
    restore_best_weights=True, mode='min')
model_checkpoint = ModelCheckpoint('best_model.keras',
    save_best_only=True, monitor='val_loss', mode='min')
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
    patience=4, mode='min')

# Train the model
history = model.fit(X_train, y_train,
                    batch_size=32,  # Optimized batch size
                    epochs=100,
                    validation_split=0.1,
                    callbacks=[early_stopping, model_checkpoint,
                       lr_scheduler],
                    verbose=1)
```

Here are the important parts we tuned on the neural network model:

1. **Number of Layers**: Used no layer, 1 layer, 2 layer, 3 layers, 4 layers and 8 layers. It is found that no layer works best among all these (simply a sigmoid function between input and output gave the best results :))

2. **Activation function**: Tried relu, leaky relu, elu and gelu model. Found that leaky relu worked best among all.

3. **Dropout**: Changed dropout between 0.3, 0.4 and 0.5 and no dropout for each layer.

4. **Optimizer**: Tried adam, nadam, RMSprop, Adamax, Adagrad. Different optimizers worked well under different conditions (dependent on other parameters)

5. **Batch size**: Tried batch sizes 32, 64 and 128. Found 32 is ideal for our problem.

6. **Learning rate**:learning rate of 0.0003 was taken for all the models which was found to be ideal after doing trial and error.

   Number of epochs is set to 100. Although, This will not be reached every time as we included early stopping criterion and the model is trained within 100 epochs usually.

   Permutations and combinations were made with all these parameters and the optimal parameters were found which will be discussed with their corresponding scores in the further sections.

0.5.1 Additional things used in the Neural Network

- **Early stopping** : Early stopping criterion was used as we found out after about 50-60 epoch, validation test score was not improving and started declining (model started over fitting). This will automatically stops epoch once it reaches a steady state.

Listing 4: Early stopping

```
        early_stopping = EarlyStopping(monitor='val_loss',
           patience=8, restore_best_weights=True, mode='min')
```

- **Dynamic learning rate change**: The ReduceLROnPlateau learning rate scheduler is a helpful tool for adjusting the learning rate dynamically during training. This setup helps to prevent overfitting and allows the model to converge more effectively by slowing down the learning rate when progress plateaus.

Listing 5: lr scheduler

```
        lr_scheduler = ReduceLROnPlateau(monitor='val_loss',
           factor=0.5, patience=4, mode='min')
```

0.5.2 Metric Used

A custom micro average f2 score was written from scratch and used as a metric in training our neural network model. Following code snippet shows the function we have used.

Listing 6: Code for custom f2 metric

```
import tensorflow as tf

@tf.keras.saving.register_keras_serializable()
class MicroF2Score(tf.keras.metrics.Metric):
    def __init__(self, name='micro_f2_score', beta=2, **kwargs):
        super(MicroF2Score, self).__init__(name=name, **kwargs)
        self.beta = beta
        self.tp = self.add_weight(name='tp', initializer='zeros')
        self.fp = self.add_weight(name='fp', initializer='zeros')
        self.fn = self.add_weight(name='fn', initializer='zeros')

    def update_state(self, y_true, y_pred, sample_weight=None):
        # Threshold y_pred to get binary predictions
```

```
        y_pred = tf.cast(y_pred > 0.5, tf.float32)

        # Cast y_true to float32 to ensure compatibility
        y_true = tf.cast(y_true, tf.float32)

        # Calculate true positives, false positives, and false
            negatives
        true_positive = tf.reduce_sum(y_true * y_pred)
        false_positive = tf.reduce_sum(y_pred * (1 - y_true))
        false_negative = tf.reduce_sum((1 - y_pred) * y_true)

        # Update the corresponding weights
        self.tp.assign_add(true_positive)
        self.fp.assign_add(false_positive)
        self.fn.assign_add(false_negative)

    def result(self):
        precision = self.tp / (self.tp + self.fp +
            tf.keras.backend.epsilon())
        recall = self.tp / (self.tp + self.fn +
            tf.keras.backend.epsilon())
        f_beta = (1 + self.beta**2) * (precision * recall) /
            (self.beta**2 * precision + recall +
            tf.keras.backend.epsilon())
        return f_beta

    def reset_states(self):
        self.tp.assign(0)
        self.fp.assign(0)
        self.fn.assign(0)
```

0.5.3 Prediction and Outputting as csv files

The following code generates the output csv file as required in the submission.

Listing 7: Code for prediction and outputting required csv file in logreg model

```
import tensorflow as tf
import numpy as np
import pandas as pd
import joblib
import os

num_labels = len(unique_codes)  # Total number of labels

# Initialize the StringLookup layer for reverse lookup
lookup_layer = tf.keras.layers.StringLookup(
    vocabulary=unique_codes, invert=True, output_mode="int",
        mask_token=None, num_oov_indices=0
)

# Folder where models are saved
model_folder = "models"

# Array to store binary predictions for each label
y_pred = np.zeros((X_test.shape[0], num_labels), dtype=int)

```

```
# Predict for each label using the corresponding model
for label_index in range(num_labels):
    # Load the saved model for this label
    model_path = f"{model_folder}/model_{label_index}.pkl"
    model = joblib.load(model_path)

    # Predict probabilities for the test set and convert to binary labels
    y_pred_probs = model.predict_proba(X_test)[:, 1]  # Take probabilities for the positive class
    y_pred[:, label_index] = (y_pred_probs > 0.5).astype(int)

# Convert binary predictions to ICD-10 codes
predicted_indices = [np.where(pred_row == 1)[0] for pred_row in y_pred]
predicted_codes = [lookup_layer(indices).numpy() for indices in predicted_indices]
predicted_codes = [[code.decode('utf-8') for code in row] for row in predicted_codes]

# Join ICD-10 codes with semicolons for each test instance
predicted_labels = [';'.join(row) for row in predicted_codes]

# Create the final submission DataFrame
submission_df = pd.DataFrame({
    'id': range(1, len(predicted_labels) + 1),
    'labels': predicted_labels
})

# Save to CSV file
submission_df.to_csv('submission.csv', index=False)
print("Predictions saved to 'submission.csv'.")
```

Listing 8: output for Neural network model

```
import tensorflow as tf
import numpy as np
import pandas as pd

# Load test data
X_test = np.load('test_data.npy')

# Predict probabilities and convert to binary
y_pred_probs = model.predict(X_test)
y_pred = (y_pred_probs > 0.5).astype(int)

# Prepare reverse lookup
lookup_layer = tf.keras.layers.StringLookup(
    vocabulary=unique_codes, invert=True, output_mode="int", mask_token=None, num_oov_indices=0
)

# Convert predictions to ICD10 codes
predicted_indices = [np.where(pred_row == 1)[0] for pred_row in y_pred]
predicted_codes = [lookup_layer(indices).numpy() for indices in predicted_indices]
predicted_codes = [[code.decode('utf-8') for code in row] for row in predicted_codes]
predicted_labels = [';'.join(row) for row in predicted_codes]
```

```
22
23 # Create and save submission DataFrame
24 submission_df = pd.DataFrame({
25     'id': range(1, len(predicted_labels) + 1),
26     'labels': predicted_labels
27 })
28 submission_df.to_csv('submission.csv', index=False)
```

0.5.4 Ensemble

Listing 9: Ensemble methods used

```
1  # Ensemble methods
2  def majority_voting(preds):
3      binary_preds = (preds > 0.5).astype(int)
4      majority_vote = np.mean(binary_preds, axis=0) >= 0.5
5      return majority_vote.astype(int)
6
7  def weighted_average(preds, weights):
8      weighted_preds = np.average(preds, axis=0, weights=weights)
9      return (weighted_preds > 0.5).astype(int)
10
11 def max_pooling(preds):
12     max_preds = np.max(preds, axis=0)
13     return (max_preds > 0.5).astype(int)
14
15 def threshold_adjustment(preds, threshold=0.4):
16     avg_preds = np.mean(preds, axis=0)
17     return (avg_preds > threshold).astype(int)
```

An ensemble of best models was done which turned out to be the best prediction.

0.6 Hacks and Workarounds

- Developed an efficient batching system using TensorFlow `tf.data.Dataset` to preprocess large-scale label data.
- Leveraged `StringLookup` with a custom vocabulary to ensure consistent multi-hot encoding for label sets.
- Designed memory-efficient pipelines to handle datasets with high-dimensional embeddings and large label vocabularies.
- Multiple ensemble models of neural networks were used, each configured with different layers and types of optimizers. Aggregation of predictions was performed using a voting mechanism. Final predictions for the test dataset were fine-tuned with threshold tuning based on the validation set.

0.7 Performance

The final model is an ensemble of various simple neural network models which produced a score of around 0.475 and the logreg model.

The final score we got is 0.490 with a public score of 0.489 placing in the sixth position overall.

As our best model is an ensemble of other models (we just used ensemble techniques on the test outputs), there is no particular training and validation score as no model is seperately done.

However, for the best individual model trained (we got a score of 0.478 with this), the model is a no layer, no dropout adam optimizer neural network model. The training score (custom f2 score used) we obtained from training the model is 0.876 and the validation score is 0.837 for this model.

0.8 Conclusion

It is found that the models we train is very particular for our dataset. We have tried a lot of combinations but we are shocked to find that the more basic our codes became (reduced the complexity of the models), the better our scores were.

Also, domain knowledge also plays an important role in building our model. The idea of using simpler model occured to us when we were searching about the origin of the dataset, how it is produced and understanding more about the features and labels.