Multi-class Classification on Normal and Abnormal ECG Based on Machine Learning

Peicong Li, Sijun Yuan, Ting Wu

Introduction

In this project we aim to create and train a model using PTB-XL dataset, which contains ECG classified into five categories: Normal ECG, Myocardial Infarction, ST/T Change, Conduction Disturbance and Hypertrophy. We initially plan to implement linear multi-class classification on it. However, as we dig deeper, this work appears to be much more complicated than we thought. After implementing complex models like Convolutional Neural Networks(CNN) and combination of CNN and LSTM, the accuracy rate always fails to exceed 60%. So this is a failure analysis report, which could provide us some experience for future study in machine learning.

Dataset

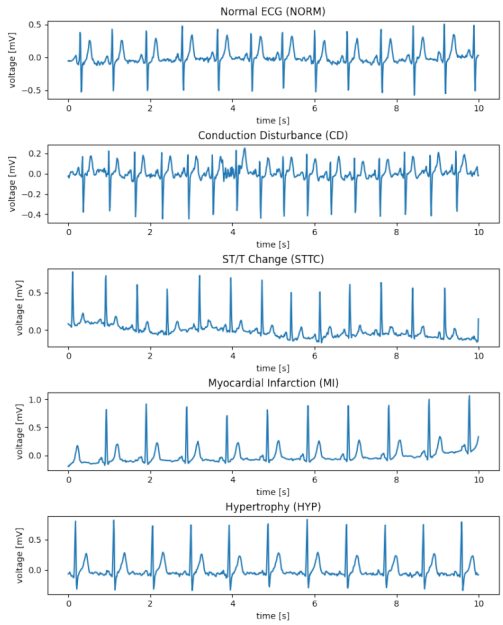
Our dataset is from a website:

https://physionet.org/content/ptb-xl/1.0.3/#files-panel

This website contains the description and the introduction of the dataset we choose, and we downloaded the whole package at the bottom of the page. The PTB-XL dataset is a comprehensive electrocardiography dataset that includes 21,799 clinical 12-lead ECG records from 18,869 patients. The records span a length of 10 seconds and cover a patient age range from 0 to 95 years. The dataset is organized into a structure containing waveform files stored in WFDB format with a precision of 16 bits, and a resolution of 1μV/LSB. Sampling frequencies are available at 500Hz and a downsampled 100Hz. The dataset includes a ptbxl_database.csv file which contains 28 columns of metadata including patient identifiers, demographic and recording metadata, ECG statements, signal quality metadata, and cross-validation folds for model training and testing.

In [6]:
```
from IPython.display import Image
Image(filename="5classesECG.png",width=450,height=300)
```

Out[6]:



Work

By implementing the official code provided on Physionet, we loaded the dataset and then extracted X and Y.

In [6]:
```
import pandas as pd
import numpy as np
import wfdb
import ast

def load_raw_data(df, sampling_rate, path):
    if sampling_rate == 100:
        data = [wfdb.rdsamp(path+f) for f in df.filename_lr]
    else:
        data = [wfdb.rdsamp(path+f) for f in df.filename_hr]
    data = np.array([signal for signal, meta in data])
    return data

path = 'ptb-xl-a-large-publicly-available-electrocardiography-dataset-1.0.1/'

sampling_rate=100

# load and convert annotation data
Y = pd.read_csv(path+'ptbxl_database.csv', index_col='ecg_id')
Y.scp_codes = Y.scp_codes.apply(lambda x: ast.literal_eval(x))
```

```
# Load raw signal data
X = load_raw_data(Y, sampling_rate, path)

# Load scp_statements.csv for diagnostic aggregation
agg_df = pd.read_csv(path+'scp_statements.csv', index_col=0)
agg_df = agg_df[agg_df.diagnostic == 1]

def aggregate_diagnostic(y_dic):
    tmp = []
    for key in y_dic.keys():
        if key in agg_df.index:
            tmp.append(agg_df.loc[key].diagnostic_class)
    return list(set(tmp))

# Apply diagnostic superclass
Y['diagnostic_superclass'] = Y.scp_codes.apply(aggregate_diagnostic)

# Split data into train and test
test_fold = 10
# Train
X_train = X[np.where(Y.strat_fold != test_fold)]
y_train = Y[(Y.strat_fold != test_fold)].diagnostic_superclass
# Test
X_test = X[np.where(Y.strat_fold == test_fold)]
y_test = Y[Y.strat_fold == test_fold].diagnostic_superclass
```

In [7]:
```
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_test shape:", X_test.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (19634, 1000, 12)
y_train shape: (19634,)
X_test shape: (2203, 1000, 12)
y_test shape: (2203,)
```

In [8]:
```
# Flatten the list of lists into a single list
all_diagnostic_classes = [item for sublist in Y['diagnostic_superclass'] for item in sublist]

# Convert to a set to find unique elements
unique_diagnostic_classes = set(all_diagnostic_classes)

# Print unique diagnostic classes
print(unique_diagnostic_classes)

# If you want to count occurrences of each diagnostic class
from collections import Counter
diagnostic_class_counts = Counter(all_diagnostic_classes)
print(diagnostic_class_counts)
```

```
{'NORM', 'STTC', 'HYP', 'CD', 'MI'}
Counter({'NORM': 9528, 'MI': 5486, 'STTC': 5250, 'CD': 4907, 'HYP': 2655})
```

As it is shown above, fold1 to fold9 are used for training while fold 10 is used for testing. X are samples that have 10s ECG records with a sampling frequency of 100Hz, resulting in a length of 1000 time points. 12 means on each time point, 12 leads are recorded. This means the input features should be a 1000*12 2-D array. The output labels are more interesting. Y is multi-label, which means that each label can be represented using one or multiple classes, and that one patient might have multiple heart diseases, making the dataset more complex than those with only one class in each label. So far, it should be very obvious that our original proposal with a linear model, is not realistic at all, as it has to handle 12000 features after flattening per sample. But we still gave it a try.

In [9]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import accuracy_score, classification_report

# Flatten the input data
X_train_flat = X_train.reshape((X_train.shape[0], -1))
X_test_flat = X_test.reshape((X_test.shape[0], -1))

# MultiLabelBinarizer to convert the labels to a suitable binary format
mlb = MultiLabelBinarizer()
y_train_bin = mlb.fit_transform(y_train)
y_test_bin = mlb.transform(y_test)

# Initialize OneVsRestClassifier with LogisticRegression
logreg_model = OneVsRestClassifier(LogisticRegression(max_iter=2000))

# Train the model
logreg_model.fit(X_train_flat, y_train_bin)

# Predict on the test data
y_pred_bin = logreg_model.predict(X_test_flat)

# Evaluate the performance
accuracy = accuracy_score(y_test_bin, y_pred_bin)
print("Accuracy: {:.2f}%".format(accuracy * 100))
print("Classification Report:\n", classification_report(y_test_bin, y_pred_bin, target_names=mlb.classes_))
```

```
Accuracy: 17.25%
Classification Report:
              precision    recall  f1-score   support

          CD       0.24      0.19      0.21       498
         HYP       0.28      0.18      0.22       263
          MI       0.29      0.22      0.25       553
        NORM       0.42      0.43      0.43       964
        STTC       0.28      0.20      0.24       523

   micro avg       0.33      0.28      0.31      2801
   macro avg       0.30      0.25      0.27      2801
weighted avg       0.32      0.28      0.30      2801
 samples avg       0.27      0.29      0.27      2801
```

```
E:\anaconda3\envs\tf2_env\lib\site-packages\sklearn\metrics\_classification.py:1471: UndefinedMetricWarning: Precision and F-score are ill-defined and being set to 0.0 in samples with no predicted labels. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
E:\anaconda3\envs\tf2_env\lib\site-packages\sklearn\metrics\_classification.py:1471: UndefinedMetricWarning: Recall and F-score are ill-defined and being set to 0.0 in samples with no true labels. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

We picked a logistic regression model for training, firstly flatten the data, and then use MultiLabelBinarizer to convert the labels of five different classes to a suitable binary format as output of the model. Obviously, this is super awful result. Then we look into a more complex model for handling input and output of our dataset, which is Convolutional Neural Network (CNN). We constructed a CNN simple CNN model using TensorFlow and scikit-learn. The dataset, comprising training and testing sets denoted as X_train and X_test, was reshaped for CNN compatibility to dimensions (batch_size, height, width, channels), specifically (1000, 12, 1). Labels y_train and y_test were binarized via scikit-learn's MultiLabelBinarizer. This A Keras Sequential CNN model was constructed with two convolutional and max-pooling layers, a flattening step, and two dense layers, employing a sigmoid output activation. The model utilized the RMSprop optimizer, binary crossentropy loss, and was assessed on accuracy

Training over 10 epochs with a batch size of 64, the model validated using 10% of the training data. Post-training, the model's predictions on X_test were thresholded at 0.5 to binary form, then reverted to label format using MultiLabelBinarizer's inverse_transform, yielding y_pred_labels. Model accuracy was quantified by comparing y_test_bin with y_pred using the accuracy_score metric.

In [10]:
```python
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import accuracy_score, classification_report

# Reshape data for CNN input
X_train_CNN = X_train.reshape((-1, 1000, 12, 1))
X_test_CNN = X_test.reshape((-1, 1000, 12, 1))

# Convert labels to binary form
mlb = MultiLabelBinarizer()
y_train_bin = mlb.fit_transform(y_train)
y_test_bin = mlb.transform(y_test)

# Create a CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(1000, 12, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(y_train_bin.shape[1], activation='sigmoid'))


# Compile the model
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# model summary
model.summary()


# Train the model
model.fit(X_train_CNN, y_train_bin, epochs=10, batch_size=64, validation_split=0.1)

# Evaluate the model on the test set
y_pred_prob = model.predict(X_test_CNN)
y_pred = (y_pred_prob > 0.5).astype(int)

# Convert predictions to original labels
y_pred_labels = mlb.inverse_transform(y_pred)

# Calculate subset accuracy
subset_accuracy = accuracy_score(y_test_bin, y_pred)
print("Subset Accuracy: {:.2f}".format(subset_accuracy))
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_2 (Conv2D)           (None, 998, 10, 32)       320

 max_pooling2d_2 (MaxPooling  (None, 499, 5, 32)       0
 2D)

 conv2d_3 (Conv2D)           (None, 497, 3, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 248, 1, 64)       0
 2D)

 flatten_1 (Flatten)         (None, 15872)             0

 dense_2 (Dense)             (None, 128)               2031744

 dense_3 (Dense)             (None, 5)                 645

=================================================================
Total params: 2,051,205
Trainable params: 2,051,205
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
277/277 [==============================] - 5s 14ms/step - loss: 0.4505 - accuracy: 0.5418 - val_loss: 0.4797 - val_accuracy: 0.4374
Epoch 2/10
277/277 [==============================] - 4s 14ms/step - loss: 0.3655 - accuracy: 0.6290 - val_loss: 0.4579 - val_accuracy: 0.5239
Epoch 3/10
277/277 [==============================] - 4s 14ms/step - loss: 0.3271 - accuracy: 0.6659 - val_loss: 0.4034 - val_accuracy: 0.5855
Epoch 4/10
277/277 [==============================] - 3s 12ms/step - loss: 0.2944 - accuracy: 0.6864 - val_loss: 0.4179 - val_accuracy: 0.5708
Epoch 5/10
277/277 [==============================] - 3s 12ms/step - loss: 0.2641 - accuracy: 0.7105 - val_loss: 0.3852 - val_accuracy: 0.6186
Epoch 6/10
277/277 [==============================] - 3s 12ms/step - loss: 0.2344 - accuracy: 0.7352 - val_loss: 0.4323 - val_accuracy: 0.5998
Epoch 7/10
277/277 [==============================] - 3s 12ms/step - loss: 0.2021 - accuracy: 0.7565 - val_loss: 0.4272 - val_accuracy: 0.5896
Epoch 8/10
277/277 [==============================] - 3s 12ms/step - loss: 0.1684 - accuracy: 0.7778 - val_loss: 0.4728 - val_accuracy: 0.6090
Epoch 9/10
277/277 [==============================] - 3s 12ms/step - loss: 0.1345 - accuracy: 0.8034 - val_loss: 0.5593 - val_accuracy: 0.6054
Epoch 10/10
277/277 [==============================] - 3s 12ms/step - loss: 0.1031 - accuracy: 0.8184 - val_loss: 0.6227 - val_accuracy: 0.5932
69/69 [==============================] - 0s 2ms/step
Subset Accuracy: 0.48
```

The result of CNN is much better than Logistic Regression, but still not satisfactory enough. Then an idea came to me, which is adding some Long Short-Term Memory(LSTM) layers on the existing CNN model. Because in my undergraduate study, I used to work on a project using LSTM modules combined with convolutional layers to deal with ECG signal compression and reconstruction, and eventually achieved quite good results. LSTM is especially highly-performed when dealing with time serires signals like ECG. Therefore, we added two LSTM layers after the existing convolutional layers, and tested over and over with different parameters.

In [12]:

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.metrics import accuracy_score, classification_report

# Reshape data for CNN input
X_train = X_train.reshape((-1, 1000, 12, 1))
X_test = X_test.reshape((-1, 1000, 12, 1))

# Convert labels to binary form
mlb = MultiLabelBinarizer()
y_train_bin = mlb.fit_transform(y_train)
y_test_bin = mlb.transform(y_test)

from tensorflow.keras import models, layers

# Model configuration
input_shape = (1000, 12, 1)
num_classes = y_train_bin.shape[1]

# Create the CNN-LSTM model
model = models.Sequential()

# CNN layers
model.add(layers.Conv2D(256, (3, 3), activation='relu', input_shape=input_shape))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(512, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))

# Reshaping output for LSTM layer
new_shape = (model.output_shape[1], model.output_shape[2] * model.output_shape[3])
model.add(layers.Reshape(target_shape=new_shape))

# LSTM layers
model.add(layers.LSTM(128, return_sequences=True))
model.add(layers.Dropout(0.4))
model.add(layers.LSTM(64))
model.add(layers.Dropout(0.5))

# Dense layers
model.add(layers.Dense(128, activation='relu'))
```

```
model.add(layers.Dense(num_classes, activation='sigmoid'))

# Compile the model (You'll need to specify the loss function and optimizer)
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Model summary
model.summary()

# Train the model
model.fit(X_train, y_train_bin, epochs=10, batch_size=64, validation_split=0.1)

# Evaluate the model on the test set
y_pred_prob = model.predict(X_test)
y_pred = (y_pred_prob > 0.5).astype(int)

# Convert predictions to original labels
y_pred_labels = mlb.inverse_transform(y_pred)

# Calculate subset accuracy
subset_accuracy = accuracy_score(y_test_bin, y_pred)
print("Subset Accuracy: {:.2f}".format(subset_accuracy))
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 998, 10, 256)      2560

 max_pooling2d_6 (MaxPooling  (None, 499, 5, 256)      0
 2D)

 conv2d_7 (Conv2D)           (None, 497, 3, 512)       1180160

 max_pooling2d_7 (MaxPooling  (None, 248, 1, 512)      0
 2D)

 reshape_1 (Reshape)         (None, 248, 512)          0

 lstm_2 (LSTM)               (None, 248, 128)          328192

 dropout_2 (Dropout)         (None, 248, 128)          0

 lstm_3 (LSTM)               (None, 64)                49408

 dropout_3 (Dropout)         (None, 64)                0

 dense_6 (Dense)             (None, 128)               8320

 dense_7 (Dense)             (None, 5)                 645

=================================================================
Total params: 1,569,285
Trainable params: 1,569,285
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
277/277 [==============================] - 42s 140ms/step - loss: 0.5414 - accuracy: 0.4256 - val_loss: 0.5227 - val_accuracy: 0.3457
Epoch 2/10
277/277 [==============================] - 37s 134ms/step - loss: 0.4840 - accuracy: 0.4737 - val_loss: 0.5262 - val_accuracy: 0.3585
Epoch 3/10
277/277 [==============================] - 37s 135ms/step - loss: 0.4376 - accuracy: 0.5132 - val_loss: 0.4452 - val_accuracy: 0.4959
Epoch 4/10
277/277 [==============================] - 38s 137ms/step - loss: 0.4056 - accuracy: 0.5754 - val_loss: 0.4270 - val_accuracy: 0.5810
Epoch 5/10
277/277 [==============================] - 42s 153ms/step - loss: 0.3839 - accuracy: 0.6078 - val_loss: 0.3948 - val_accuracy: 0.6003
Epoch 6/10
277/277 [==============================] - 40s 144ms/step - loss: 0.3684 - accuracy: 0.6301 - val_loss: 0.3864 - val_accuracy: 0.6273
Epoch 7/10
277/277 [==============================] - 40s 144ms/step - loss: 0.3551 - accuracy: 0.6441 - val_loss: 0.3664 - val_accuracy: 0.6242
Epoch 8/10
277/277 [==============================] - 40s 144ms/step - loss: 0.3429 - accuracy: 0.6549 - val_loss: 0.3384 - val_accuracy: 0.6533
Epoch 9/10
277/277 [==============================] - 40s 144ms/step - loss: 0.3298 - accuracy: 0.6605 - val_loss: 0.3419 - val_accuracy: 0.6380
Epoch 10/10
277/277 [==============================] - 40s 145ms/step - loss: 0.3206 - accuracy: 0.6660 - val_loss: 0.3241 - val_accuracy: 0.6731
69/69 [==============================] - 3s 32ms/step
Subset Accuracy: 0.57
```

Seen from above, our upgraded model seemed to make some progress compared with the CNN one, but still didn't meet our expectations. After consulting materials and inquiring with experts, we concluded the following two reasons that lead to the bad performance of our work.

1.We ignored data cleaning, which is a crucial step in machine learning, as the performance of the model largely depends on the quality of the input data. "Garbage In, Garbage Out"

2.Our model is still not optimized enough.By reading a paper which handles the same dataset as ours, we found that they can achieve an accuracy of above 80% using a CNN model with entropy.This means that using CNN is the right direction, it's just we are not going through enough distances.It's a great pitty that we don't have enough time to further improve on our design.

Conclusion

Although at last we didn't get desired results, this is still a very rewarding experience. The process of tuning models is time-consuming, but the feeling of seeing the performance rising is wonderful. The imperfections in our work are somehow inspirations for future improvements.