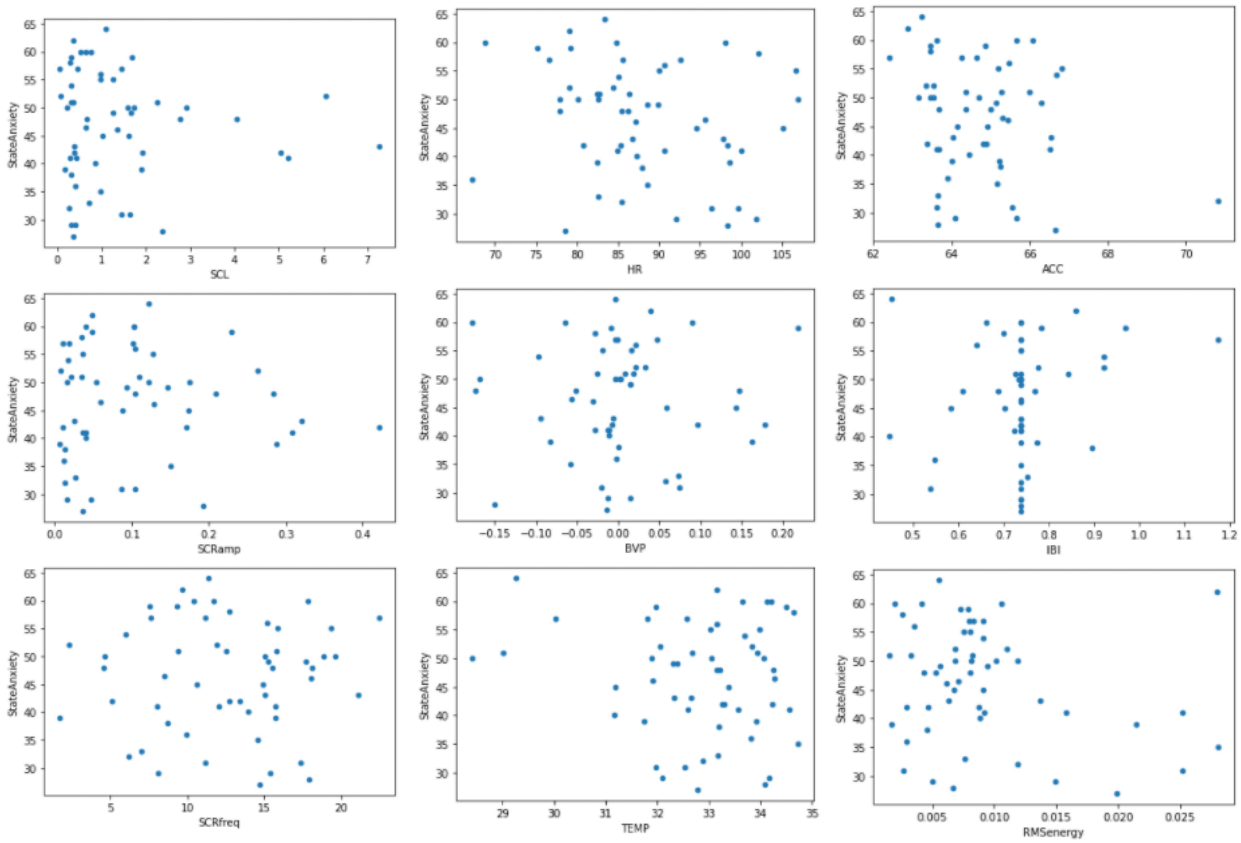
Homework 5 - CSCE633

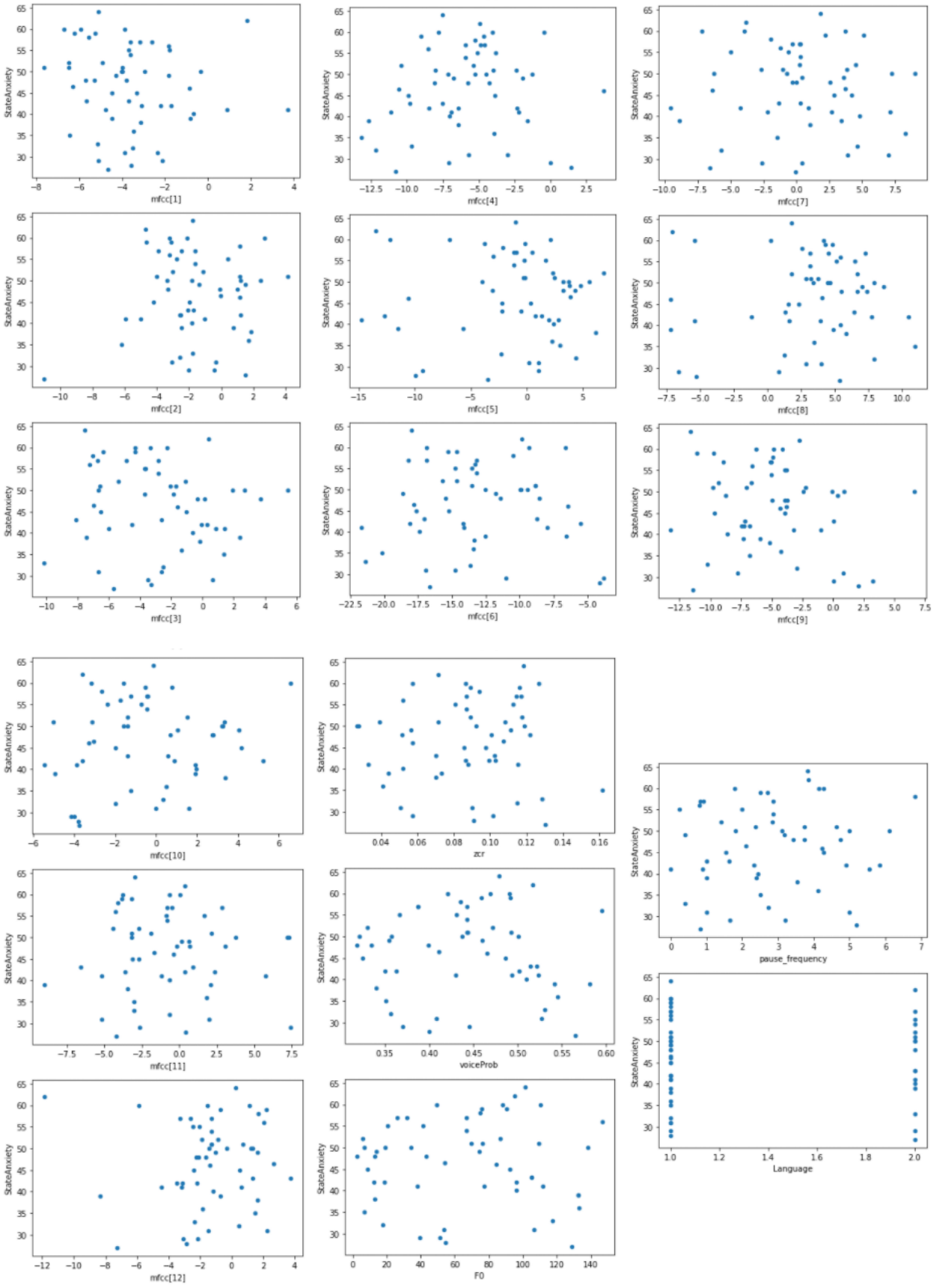
Insoo Chung, Ja-Hun Oh, Sun Yul Lee, Sagar Adhikari, Natalie Martinez

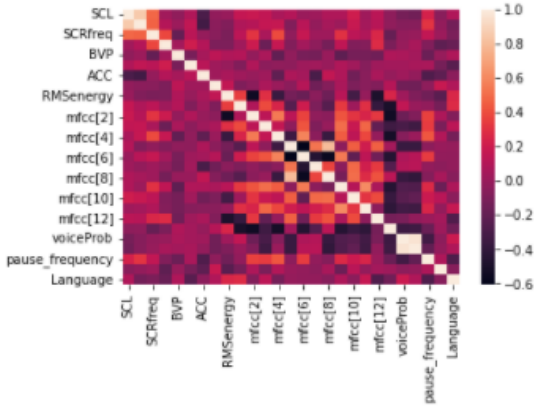
December 14, 2021

(a) Data preprocessing and exploration

In this section, we first tried to replace the missing data values with the corresponding feature mean. After the data imputation, we visualized the data using scatter plots and quantified associations between the features and the label via correlation coefficient. From the correlation coefficient matrix, we could see that RMSenergy has the highest absolute correlation coefficient with the label StateAnxiety




StateAnxiety
mfcc[1]
mfcc[4]
mfcc[7]
mfcc[2]
mfcc[5]
mfcc[8]
mfcc[3]
mfcc[6]
mfcc[9]
mfcc[10]
zcr
pause_frequency
mfcc[11]
voiceProb
Language
mfcc[12]
F0




```
Pearson's correlation coefficient.....
SCL                -0,049805
SCRamp             -0,014174
SCRfreq             0,013870
HR                 -0,217042
BVP                -0,000355
TEMP               -0,120541
ACC                -0,216606
IBI                 0,183156
RMSenergy          -0,236090
mfcc[1]            -0,125903
mfcc[2]             0,081256
mfcc[3]            -0,105996
mfcc[4]             0,096626
mfcc[5]             0,004175
mfcc[6]            -0,017388
mfcc[7]            -0,037644
mfcc[8]             0,058475
mfcc[9]            -0,167732
mfcc[10]            0,125598
mfcc[11]           -0,016609
mfcc[12]            0,045374
zcr                 0,024935
voiceProb          -0,042014
F0                 -0,017982
pause_frequency     0,077460
StateAnxiety        1,000000
Language           -0,065444
```

```python
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_selection import SequentialFeatureSelector as SFS
from sklearn.model_selection import KFold
from skfeature.function.similarity_based import fisher_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.regularizers import l2
from sklearn.linear_model import LinearRegression
import time
```

Imported libraries used in (a), (b), (e-i), and (f-i)

```python
def load_data():
    data = pd.read_csv('/content/gdrive/MyDrive/Homework5/data.csv', sep=',')
    data = data.drop(columns=['PID'])
    headers = data.columns
    data = data.to_numpy(na_value=np.nan)
    return data, headers

def imputation(strategy):
    data, headers = load_data()
    imp = SimpleImputer(missing_values=np.nan, strategy=strategy)
```

```
    imp.fit(data)
    data = imp.transform(data)
    df = pd.DataFrame(data, columns=headers)
    return df
```

Functions used for data pre-processing.

```
def visualization(data):
  print("Plotting 2D scatter plots between the numerical attributes and the outcome\n")
  for col in data.columns:
    if col != 'StateAnxiety':
      data.plot.scatter(x=col, y='StateAnxiety')
      plt.show()
  print("Pearson's correlation coefficient.....")
  sns.heatmap(data.corr(method='pearson'))
  print(data.corr(method='pearson')['StateAnxiety'])
```

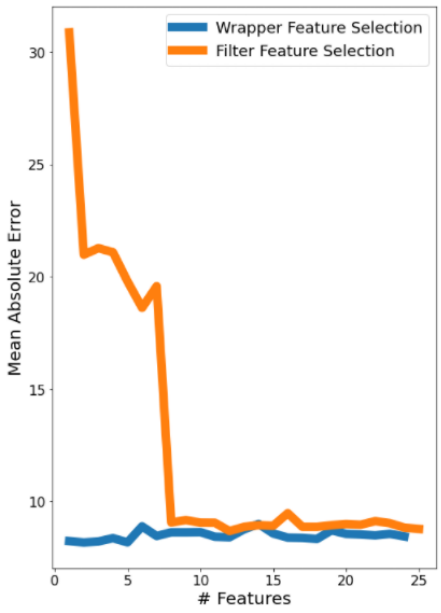
Function used for data visualization and quantification.

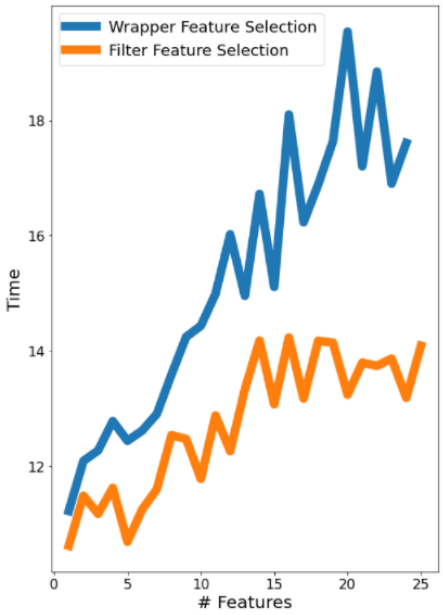
(b) Feature selection

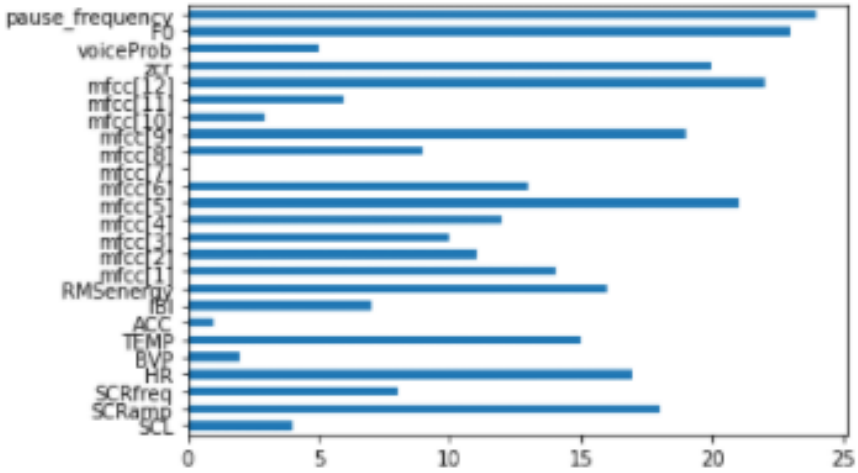
In this section, we explored two feature selection methods. We chose Forward Feature Selection method from Wrapper category and Fisher's Criterion Feature Selection method from Filter category. Using the methods, we achieved subset of the features for each number of features (1 25). Then, loaded dataset was transformed with the feature subset. WE used feedforward neural network regression model with 5-fold cross-validation for each subset and measured the performance of model by calculating the absolute error between the actual and predicted label values. The structures of FNN with each feature selection method are the same. We constructed FNN model with 3 hidden layers. Each hidden layers contains 32, 64, and 32 hidden features. We applied l2 regularization for the output of each hidden layers with learning rate of 0.0001. The activation we used for each hidden layer is ReLu function. For each model training, we calculated computation time for future comparison.

With the small number of features, Fisher's Criterion Feature Selection method produces bad performance. However, as the number of features increases, the performance becomes better. When the number of features is greater or equal to 8, FNN models with two methods have the similar performance. When we compared the computation time with two feature selection methods, FNN with Fisher's Criterion Feature Selection method has shorter computation time than FNN with Forward Feature Selection method. We also analyzed the selected features from two methods. For Forward Feature Selection method, selected features always changes. However, ACC frequently selected for the subset. For Fisher's Criterion Feature Selection method, the order of important features stayed the same. The most important feature was pause frequency.

Method	# Features	Avg Abs Err	Time
Wrapper	2	8.175	12.088
Filter	12	8.684	12.258








```
def forward_selection(data, classifier, num_features):
    y = data["StateAnxiety"]
    X = data.drop(columns=['StateAnxiety', 'Language'])
    sfs = SFS(classifier, n_features_to_select=num_features, direction='forward',
              scoring='r2')
    sfs.fit(X, y)
    return sfs.transform(X), sfs.get_support(indices=True)

def fisher(data):
    y = data["StateAnxiety"]
    X = data.drop(columns=['StateAnxiety', 'Language'])
    cols = X.columns
    X = X.to_numpy()
    y = y.to_numpy()
    ranks = fisher_score.fisher_score(X, y)
    idx = np.argsort(ranks, 0)[::-1]
```

```
    return idx
```

Functions used for selecting features with two methods: Forward Selection Feature Selection and Fisher's Criterion Feature Selection.

```
def fisher_visualize(data):
    y = data["StateAnxiety"]
    X = data.drop(columns=['StateAnxiety', 'Language'])
    cols = X.columns
    X = X.to_numpy()
    y = y.to_numpy()
    ranks = fisher_score.fisher_score(X, y)
    feat_importances = pd.Series(ranks, cols)
    feat_importances.plot(kind='barh')
    plt.show()
```

Function used for visualizing fisher score of each bio-behavioral features.

```
wrapper_scores = []
wrapper_features = []
wrapper_times = []
for d in range(1, len(data.columns) - 2):
  time_start = time.perf_counter()
  print("Number of Features: %d ----------------------------" % d)
  data_set, features = forward_selection(data, LinearRegression(), d)
  wrapper_features.append(data.columns[features])
  cv = KFold(n_splits=5, shuffle=False)
  errors = 0.0
  folds = 0
  for train_index, test_index in cv.split(data_set):
    folds += 1
    print("Fold: %d --------------------------" % folds)
    train_x, test_x = data_set[train_index], data_set[test_index]
    train_y, test_y = data['StateAnxiety'].iloc[train_index], data['StateAnxiety'].
 ↪iloc[test_index]
    train_y, test_y = train_y.to_numpy(), test_y.to_numpy()

    model = Sequential(
        [Dense(32, activation='relu', input_shape=(d,), activity_regularizer=l2(0.
 ↪0001)),
         Dense(64, activation='relu', activity_regularizer=l2(0.0001)),
         Dense(32, activation='relu', activity_regularizer=l2(0.0001)),
         Dense(1)]
    )
    optimizer = RMSprop(0.001)
    model.compile(optimizer=optimizer, loss='mse',metrics=['mse'])
    model.fit(train_x, train_y, epochs=30)
    preds = model.predict(test_x)
    error = np.mean(abs(test_y - preds))
    errors += error
  avg_error = errors / 5
  time_end = (time.perf_counter() - time_start)
  wrapper_scores.append(avg_error)
  wrapper_times.append(time_end)
```

Training FNN with Forward Feature Selection method.

```
filter_scores = []
filter_features = []
filter_times = []
fisher_visualize(data)
for d in range(1, len(data.columns) - 1):
  time_start = time.perf_counter()
  print("Number of Features: %d ----------------------------" % d)
  idx = fisher(data)
  filtered_data = data.drop(columns=['StateAnxiety', 'Language']).to_numpy()
  filter_features.append(data.columns[idx[0:d]])
  data_set = filtered_data[:, idx[0:d]]
  cv = KFold(n_splits=5, shuffle=False)
  errors = 0.0
  folds = 0
  for train_index, test_index in cv.split(data_set):
    folds += 1
    print("Fold: %d --------------------------" % folds)
    train_x, test_x = data_set[train_index], data_set[test_index]
    train_y, test_y = data['StateAnxiety'].iloc[train_index], data['StateAnxiety'].
 ↪iloc[test_index]
    train_y, test_y = train_y.to_numpy(), test_y.to_numpy()

    model = Sequential(
        [Dense(32, activation='relu', input_shape=(d,), activity_regularizer=l2(0.
 ↪0001)),
         Dense(64, activation='relu', activity_regularizer=l2(0.0001)),
         Dense(32, activation='relu', activity_regularizer=l2(0.0001)),
         Dense(1)]
    )
    optimizer = RMSprop(0.001)
    model.compile(optimizer=optimizer, loss='mse',metrics=['mse'])
    model.fit(train_x, train_y, epochs=30)
    preds = model.predict(test_x)
    error = np.mean(abs(test_y - preds))
    errors += error
  avg_error = errors / 5
  time_end = (time.perf_counter() - time_start)
  filter_scores.append(avg_error)
  filter_times.append(time_end)
```

Training FNN with Fisher's Criterion Feature Selection method.

```
plt.figure(figsize=(8,12))
plt.xlabel('# Features', fontsize=20)
plt.ylabel('Mean Absolute Error', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.plot(range(1, len(data.columns) - 2), wrapper_scores,
         label='Wrapper Feature Selection', linewidth=10.0)
plt.plot(range(1, len(data.columns) - 1), filter_scores,
         label='Filter Feature Selection', linewidth=10.0)
plt.legend(prop={"size":18})
```

```
plt.show()

plt.figure(figsize=(8,12))
plt.xlabel('# Features', fontsize=20)
plt.ylabel('Time', fontsize=20)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
plt.plot(range(1, len(data.columns) - 2), wrapper_times,
         label='Wrapper Feature Selection', linewidth=10.0)
plt.plot(range(1, len(data.columns) - 1), filter_times,
         label='Filter Feature Selection', linewidth=10.0)
plt.legend(loc=2, prop={"size":18})
plt.show()
```

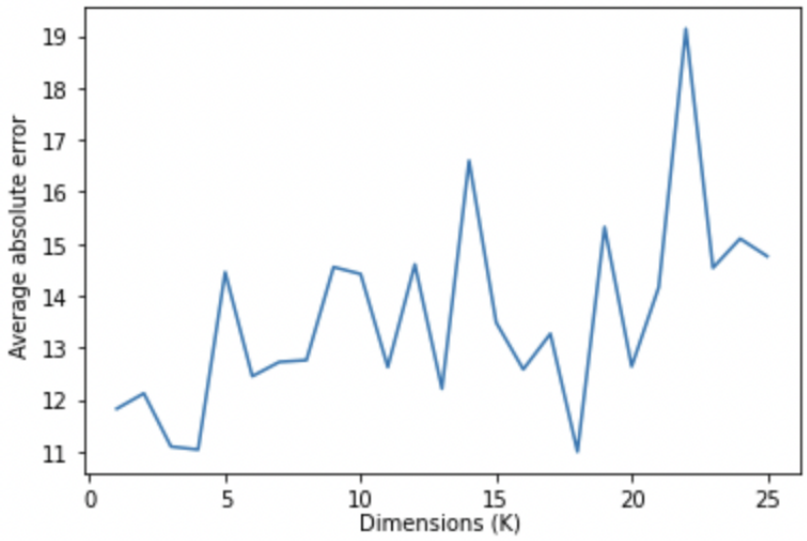
Plotting the performance of FNN models and computation times for each number of features and feature selection method.

```
wrapper_idx = np.argmin(wrapper_scores)
filter_idx = np.argmin(filter_scores)
print("Wrapper")
print("# Features: %d" % len(wrapper_features[wrapper_idx]))
print("Features:")
print(wrapper_features[wrapper_idx])
print("Average Absolute Error: %f" % wrapper_scores[wrapper_idx])
print("Time: %f" % wrapper_times[wrapper_idx])
print()
print("Filter")
print("# Features: %d" % len(filter_features[filter_idx]))
print("Features:")
print(filter_features[filter_idx])
print("Average Absolute Error: %f" % filter_scores[filter_idx])
print("Time: %f" % filter_times[filter_idx])
```

Displaying the number of features, performance, and computation time of the best model for each feature selection method.

(c) Feature transformation

In this section, we used Principal Component Analysis (PCA) to reduce the dimensionality of the bio-behavioral features from D to K. We experimented with different values of k ranging from 1 to 25. The best results were observed for k = 18 which achieved the average absolute error of 10.49. With k=18 we were able to cover 98% variance of the data.




```
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from keras.regularizers import l2


def train_valid_split(data):
    train, test = train_test_split(data, test_size=1 / 5.0, random_state=0)
    return train.drop(columns=['StateAnxiety', 'Language']), test.
 ↪drop(columns=['StateAnxiety', 'Language']), train[['StateAnxiety', 'Language']],␣
 ↪test[['StateAnxiety', 'Language']]

def five_fold_split(data):
    #shuffle and split
    shuffled = data.sample(frac=1)
    fivefold = np.array_split(shuffled, 5)
    return fivefold

def FNN(train_x, train_y, test_x, test_y, input_features):
    model = Sequential()
    model.add(Dense(64, activation='relu',
                    input_shape= (input_features,),
                    kernel_regularizer=l2(0.01)))
    model.add(Dropout(0.1))
    model.add(Dense(32, activation='relu',
                    kernel_regularizer=l2(0.01)))
    model.add(Dropout(0.1))
    model.add(Dense(64, activation='relu',
                    kernel_regularizer=l2(0.01)))
```

```
    model.add(Dropout(0.1))
    model.add(Dense(1))
    model.compile(optimizer='sgd', loss=tf.keras.losses.MeanAbsoluteError())
    es = tf.keras.callbacks.EarlyStopping(monitor="loss", mode="min", verbose=0,␣
 ↪patience=5, restore_best_weights=True)
    fnn = model.fit(train_x, train_y, epochs=100, batch_size=2, verbose=0, callbacks =␣
 ↪[es])
    preds = model.predict(test_x)

    print("k:", input_features, "MeanAbsoluteError", np.mean((abs(preds-test_y))))
    print("-----------------------------------------------------")
    return np.mean((abs(preds-test_y)))

def transform(data):
    scaler = StandardScaler()
    fivefold = five_fold_split(data)

    x = []
    y = []
    for k in range(1, 26):
      errorList = []
      for i in range(5):
        train_list = []
        for j in range(0,5):
          if i!=j :
            train_list.append(fivefold[j])
        train = pd.concat(train_list)
        test = fivefold[i]


        train_x = train.drop(columns=['StateAnxiety', 'Language'])
        test_x = test.drop(columns=['StateAnxiety', 'Language'])
        train_y = train[['StateAnxiety']]
        test_y = test[['StateAnxiety']]

        scaler.fit(train_x)
        train_x = scaler.transform(train_x)
        test_x = scaler.transform(test_x)
        pca = PCA()
        pca.fit(train_x)
        eigenvalues = pca.explained_variance_
        eigenvectors = pca.components_

        transformation_matrix = eigenvectors[0:k, :]
        reduced_train = np.dot(transformation_matrix, train_x.T).T
        reduced_test = np.dot(transformation_matrix, test_x.T).T
        error = FNN(reduced_train, train_y[['StateAnxiety']], reduced_test,␣
 ↪test_y[['StateAnxiety']], k)
        errorList.append(error)
        print(pca.explained_variance_ratio_.cumsum())

      averageError = sum(errorList)/5.0
      x.append(k)
```

```
      y.append(averageError)
    for i in range(0, len(x)):
      print(x[i], y[i])
    plt.plot(x, y)
    plt.xlabel("Dimensions (K)")
    plt.ylabel("Average absolute error")
    plt.show()
```

(d) Working with time series

(d-i) FNN approach

In this task, we adopted FNN model by pulling out bias and slope from timeseries obtained from HR and EDA respectively. By plotting the relationship between time and output(HR, EDA), we concluded that using linear regression model to derive bias and slope is better than that of non-linear model. It is because for many dataset, we can find gradually decreasing or increasing pattern in the output. For the training network model, we used FNN regression model with 5 fold cross-validation and calculated the absolute error between the actual and predicted label values to evaluate the model's accuracy which lead close to 8 of MAE. The FNN model consists of 3 hidden layers, each hidden layers contains 32, 64, and 32 hidden features respectively. We also applied l2 regularization for the output of each hidden layers with learning rate of 0.01. For the optimizer, we adopted stochastic gradient descent.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import preprocessing
```

Imports needed to pull out parameters from the time series using linear and non-linear regression models. The parameters of the FNN model composed of the bias and slope of the linear regression.

```
#Load EDA, get slope and bias
def eda_linear():
    data = pd.read_csv('Homework5/data.csv', sep=',')
    files = os.listdir('Homework5')

    data_eda = pd.DataFrame()
    for file in files:
        if "EDA" in file:
            pid = file[8:12]
            eda = pd.read_excel(os.path.join('Homework5', file))
            times = eda[['Time (s)']]
            edas = eda['EDA']

            # adopt linear model
            model = LinearRegression().fit(times, edas)
            slope = model.coef_[0]
            bias = model.intercept_
            anxiety = data.loc[data['PID'] == pid]['StateAnxiety'].tolist()[0]
            new_row = {'PID': pid, 'EDA_Slope': slope, 'EDA_Bias': bias,
```

```
                        'StateAnxiety': anxiety}
            data_eda = data_eda.append(new_row, ignore_index=True)
    return data_eda

def hr_linear():
    data = pd.read_csv('Homework5/data.csv', sep=',')
    files = os.listdir('Homework5')

    data_hr = pd.DataFrame()
    for file in files:
        if "HR" in file:
            pid = file[7:11]
            eda = pd.read_excel(os.path.join('Homework5', file))
            times = eda[['Time (s)']]
            edas = eda['HR']

            # adopt linear model
            model = LinearRegression().fit(times, edas)
            slope = model.coef_[0]
            bias = model.intercept_
            anxiety = data.loc[data['PID'] == pid]['StateAnxiety'].tolist()[0]
            # language
            language = data.loc[data['PID'] == pid]['Language'].tolist()[0]
            new_row = {'PID': pid, 'HR_Slope': slope, 'HR_Bias': bias,
                       'StateAnxiety': anxiety, 'Language': language}
            data_hr = data_hr.append(new_row, ignore_index=True)
            data_hr.dropna(axis=1)
    return data_hr
```

```
def main():
    data = preprocessing.imputation("mean")
    data_eda = preprocessing.load_eda()
    data_hr = preprocessing.load_hr()
    return data, data_eda, data_hr

if __name__ == "__main__":
    data,_,_ = main()               # import data
    print(data.head(3))


    data_eda = eda_linear()
    data_hr = hr_linear()
    result = pd.concat([data_eda, data_hr], axis=1, join='inner')
    result = result[['PID', 'Language', 'EDA_Bias', 'EDA_Slope', 'HR_Bias',␣
 ↪'HR_Slope', 'StateAnxiety']]
    result = result.loc[:, ~result.columns.duplicated()]
    X = result.drop('StateAnxiety',1)
    y = result['StateAnxiety']
    print(result.head(3))
```

```
        SCL    SCRamp    SCRfreq         HR       BVP       TEMP        ACC  \
0  0.353695  0.049017   9.649805  79.043212  0.039718  33.160052  62.892356
1  0.424881  0.040277  12.075472  90.674728 -0.012490  33.560587  66.507156
2  0.164890  0.006639   1.730769  82.393494  0.162948  31.742308  65.215674
```

```
        IBI  RMSenergy   mfcc[1]  ...   mfcc[9]  mfcc[10]  mfcc[11]  \
0  0.859414   0.027972  1.812113  ... -2.724729 -3.590818  0.390383
1  0.738355   0.025150  0.895732  ... -0.994851 -5.456365 -1.185723
2  0.774074   0.021411 -0.813331  ... -7.322685 -4.966899 -8.985444

    mfcc[12]       zcr  voiceProb          F0  pause_frequency  StateAnxiety  \
0 -11.855134  0.071547   0.516881   95.473209         3.857143          62.0
1  -4.423544  0.088058   0.493523   77.454661         0.888889          41.0
2  -8.322641  0.073272   0.581680  132.855822         1.000000          39.0

   Language
0       2.0
1       1.0
2       2.0

[3 rows x 27 columns]
    PID  Language  EDA_Bias  EDA_Slope     HR_Bias  HR_Slope  StateAnxiety
0  P001       2.0  0.313593   0.000413  101.657069 -0.234657          62.0
1  P003       1.0  0.244437   0.001512   88.307235  0.019845          41.0
2  P004       2.0  0.164597   0.000004   96.652131 -0.091484          39.0
```

Preprocessing data by scaling and correlating it to the Anxiety level which would be the label of this network.

```
import torch
import torch.nn as nn
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torch.utils.data as data_utils
from torch.autograd import Variable
from sklearn.model_selection import train_test_split
```

```
result.dropna(subset = ["StateAnxiety"], inplace=True)
print(result.describe())
print(result['StateAnxiety'].nunique())
```

```
        Language   EDA_Bias  EDA_Slope     HR_Bias   HR_Slope  StateAnxiety
count  54.000000  54.000000  54.000000   54.000000  54.000000     54.000000
mean    1.314815   0.997413   0.003290   88.111185  -0.008873     46.481481
std     0.468803   1.407795   0.005763   13.576665   0.114897      9.912334
min     1.000000  -0.382165  -0.004240   64.172381  -0.303229     27.000000
25%     1.000000   0.244811   0.000197   77.416728  -0.089958     40.250000
50%     1.000000   0.587617   0.001783   85.833860  -0.004810     48.000000
75%     2.000000   1.265215   0.003085   98.234889   0.054935     54.750000
max     2.000000   7.471220   0.029220  119.938652   0.232711     64.000000
30
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

```
import tensorflow.keras.optimizers
from tensorflow.keras.regularizers import l2
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import time
```

```
import torch
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedShuffleSplit

from torch.utils.data import Dataset, DataLoader
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from keras.regularizers import l2


def train_valid_split(data):
    train, test = train_test_split(data, test_size=1 / 5.0, random_state=0)
    return train.drop(columns=['StateAnxiety', 'Language']), test.
 ↪drop(columns=['StateAnxiety', 'Language']), train[['StateAnxiety', 'Language']],␣
 ↪test[['StateAnxiety', 'Language']]

def five_fold_split(data):
    #shuffle and split
    shuffled = data.sample(frac=1)
    fivefold = np.array_split(shuffled, 5)
    return fivefold

def FNN(train_x, train_y, test_x, test_y, input_features):
    model = Sequential()
    model.add(Dense(32, activation='relu',
                    input_shape= (input_features,),
                    kernel_regularizer=l2(0.01)))
    model.add(Dense(64, activation='relu',
                    kernel_regularizer=l2(0.01)))
    model.add(Dense(32, activation='relu',
                    kernel_regularizer=l2(0.01)))
    model.add(Dense(1))
    model.compile(optimizer='sgd', loss=tf.keras.losses.MeanAbsoluteError())
    es = tf.keras.callbacks.EarlyStopping(monitor="val_loss", mode="min", patience=5,␣
 ↪restore_best_weights=True)
    fnn = model.fit(train_x, train_y, epochs=100, batch_size=16, verbose=0,
                    callbacks=[es], validation_data=(test_x, test_y))
    preds = model.predict(test_x)
```

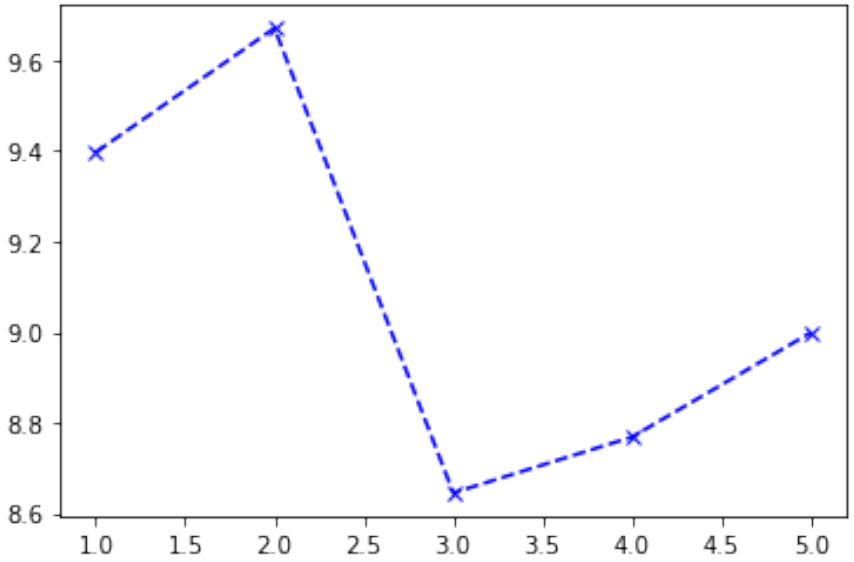
```
    print("MAE: ", np.mean((abs(preds-test_y))))


    return np.mean((abs(preds-test_y)))

def transform(data):
    scaler = MinMaxScaler()
    fivefold = five_fold_split(data)

    x = []
    y = []
    errorList = []
    for i in range(5):
        train_list = []
        for j in range(0,5):
            if i!=j : train_list.append(fivefold[j])
        train = pd.concat(train_list)
        test = fivefold[i]

        train_x = train.drop(columns=['StateAnxiety', 'Language'])
        test_x = test.drop(columns=['StateAnxiety', 'Language'])
        train_y = train[['StateAnxiety']]
        test_y = test[['StateAnxiety']]

        scaler.fit(train_x)
        train_x = scaler.transform(train_x)
        test_x = scaler.transform(test_x)

        error = FNN(train_x, train_y[['StateAnxiety']], test_x,␣
 ↪test_y[['StateAnxiety']], 4)
        errorList.append(error)

        x.append(i+1)
        averageError = sum(errorList)/(i+1)
        y.append(averageError)
        print(f'\n--------Average Error :{averageError}--------')

    plt.plot(x, y, 'bx--')
    plt.show()
```

Training model using FNN model with 3 hidden layers and using relu for the activation function. Adopting early stopping method to lower the loss.

```
data = result.drop(columns='PID')
transform(data)
```

```
MAE:  StateAnxiety    9.397027
dtype: float64

--------Average Error :StateAnxiety    9.397027
dtype: float64--------
MAE:  StateAnxiety    9.944966
```

```
dtype: float64

--------Average Error :StateAnxiety    9.670996
dtype: float64--------
MAE:  StateAnxiety    6.593185
dtype: float64

--------Average Error :StateAnxiety    8.645059
dtype: float64--------
MAE:  StateAnxiety    9.138472
dtype: float64

--------Average Error :StateAnxiety    8.768412
dtype: float64--------
MAE:  StateAnxiety    9.920831
dtype: float64

--------Average Error :StateAnxiety    8.998896
dtype: float64--------
```




```
native_data = data[data['Language']==1]
native_data.describe()
```

(d-ii) RNN approach

```
import math
import matplotlib.pyplot as plt

from statistics import mean

import tensorflow as tf
import numpy as np
```

```
from sklearn.model_selection import train_test_split

from lime import explanation
from lime import lime_base
from lime_timeseries import LimeTimeSeriesExplainer

import preprocessing
```

Imports required for RNN experiments throughout this problem and relevant portions in **(e)** and **(f)**.

```
def mean_per_window(x, window_size):
    res = []
    for i in range(0, len(x), window_size):
        res.append(mean(x[i:i + window_size]))
    return res

def process_xlsx_data(df_X, df_y, window_size, shuffle=True):
    # Process data
    X, y = [], []
    max_x_len = 0
    max_x, min_x = 0, 1e9
    for i in range(len(df_X)):
        x = mean_per_window(df_X[i], window_size)
        if len(x) > max_x_len:
            max_x_len = len(x)
        X.append(x)
        max_x = max(max_x, max(x))
        min_x = min(min_x, min(x))
        y.append(list(df_y[i]))

    for i, x in enumerate(X):
        # pad with trailing nan
        X[i] = x + [0] * (max_x_len - len(x))

    X = np.array(X)
    X = np.expand_dims(X, axis=-1)
    y = np.array(y)
    if shuffle:
        shuffler = np.random.permutation(X.shape[0])
        X = X[shuffler]
        y = y[shuffler]

    X_mask = (X > 0).astype(np.float32)
    mean_y = float(int(np.nanmean(y, axis=None)))
    stddev_y = np.nanstd(y, axis=None)
    max_y = np.nanmax(y, axis=None)
    min_y = np.nanmin(y, axis=None)
    y[np.where(np.isnan(y))] = mean_y  # replace nan with mean

    # Manual mean, stddev calculation due to padding
    X_lengths = (X_mask).astype(np.float32).sum(axis=1)
    mean_x = X.sum(axis=1) / X_lengths
    stddev_x = np.sqrt(
        np.sum(
```

```
            ((X - mean_x[:, None]) ** 2) * X_mask,
            axis=1) / X_lengths
    )
    mean_x = np.expand_dims(mean_x, axis=-1)
    stddev_x = np.expand_dims(stddev_x, axis=-1)

    return {
        "X": X,
        "X_mask": X_mask,
        "y": y,
        "mean_x": mean_x,
        "stddev_x": stddev_x,
        "mean_y": mean_y,
        "stddev_y": stddev_y,
        "min_x": min_x,
        "max_x": max_x,
        "min_y": min_y,
        "max_y": max_y
    }

def process_xlsx_data_merged(data_eda, data_hr, sec_frame,
                             eda_elem_per_sec, hr_elem_per_sec):
    data_e = process_xlsx_data(data_eda["EDA"], data_eda["StateAnxiety"],
                           eda_elem_per_sec * sec_frame, shuffle=False)
    data_h = process_xlsx_data(data_hr["HR"], data_hr["StateAnxiety"],
                           hr_elem_per_sec * sec_frame, shuffle=False)
    assert np.all(data_e["y"] == data_h["y"])
    return {
        "X": np.concatenate([data_e["X"], data_h["X"]], axis=-1),
        "X_mask": np.concatenate([data_e["X_mask"], data_h["X_mask"]], axis=-1),
        "mean_x": np.concatenate([data_e["mean_x"], data_h["mean_x"]], axis=-1),
        "stddev_x": np.concatenate([data_e["stddev_x"], data_h["stddev_x"]], axis=-1),
        "y": data_e["y"],
        "mean_y": data_e["mean_y"],
        "stddev_y": data_e["stddev_y"],
        "max_y": data_e["max_y"],
        "min_y": data_e["min_y"]
    }
```

Data is processed to numpy matrices for `tf.keras` models.

```
def get_ith_split_k_fold(data, i, k=5):
    data_cnt = data.shape[0]
    fold_len = data_cnt // k
    data_train = np.concatenate(
        [data[:i * fold_len], data[(i + 1) * fold_len:]], axis=0)
    data_val = data[i * fold_len: (i + 1) * fold_len]
    return data_train, data_val

def preprocess_data(data, normalize_x, standardize_x, standardize_y):
    X = data["X"]
    y = data["y"]
    if normalize_x:
        assert not standardize_x
```

```
        X_mask = data["X_mask"]
        mean_x = data["mean_x"]
        stddev_x = data["stddev_x"]
        X = ((X - mean_x) / stddev_x) * X_mask
    elif standardize_x:
        assert X.shape[-1] == 1 # only support single feature X
        min_x, max_x = data["min_x"], data["max_x"]
        X = (X - min_x) / (max_x - min_x)
    if standardize_y:
        # y is standardized to match the sigmoid range
        min_y, max_y = data["min_y"], data["max_y"]
        y = (y - min_y) / (max_y - min_y)

    return {"X": X, "y": y}

def postprocess_data(data, standardize_y):
    y = data["y"]
    if standardize_y:
        min_y, max_y = data["min_y"], data["max_y"]
        y = y * (max_y - min_y) + min_y

    return {"y": y}

def init_model(input_shape, rnn_layer_fn, rnn_hidden_size,
               num_rnn_layers, regression, dropout, decay_lr=False):
    layers = [tf.keras.layers.Masking(mask_value=0.,
                                      input_shape=input_shape)]
    for i in range(num_rnn_layers - 1):
        layers.append(rnn_layer_fn(rnn_hidden_size,
                                   recurrent_dropout=dropout,
                                   return_sequences=True))
    layers.append(rnn_layer_fn(rnn_hidden_size, recurrent_dropout=dropout))
    layers.append(tf.keras.layers.Dropout(dropout))

    if regression:
        layers.append(tf.keras.layers.Dense(units=1, activation="sigmoid"))
        loss = "msle"
    else:
        layers.append(tf.keras.layers.Dense(units=100, activation="softmax"))
        loss = "sparse_categorical_crossentropy"
    model = tf.keras.models.Sequential(layers)
    optimizer = "adam"
    if decay_lr:
        optimizer = tf.keras.optimizers.Adam(
            learning_rate=tf.keras.optimizers.schedules.ExponentialDecay(
                                0.0001, decay_steps=1, decay_rate=0.97))
    model.compile(loss=loss, optimizer=optimizer)
    return model

def train_model(model, X_train, y_train, X_val, y_val,
                batch_size, epochs, verbose):
    es = tf.keras.callbacks.EarlyStopping(monitor="val_loss", mode="min",
                                          verbose=verbose, patience=5,
```

```
                                        restore_best_weights=True)
    # Train RNN
    model.fit(X_train, y_train,
                epochs=epochs, batch_size=batch_size,
                validation_data=(X_val, y_val),
                callbacks=[es],
                verbose=verbose)
    return model

def exp_rnn_k_fold(data, epochs,
                    batch_size=128,
                    rnn_layer_fn=tf.keras.layers.LSTM,
                    rnn_hidden_size=128,
                    num_rnn_layers=1,
                    k=5,
                    regression=True,
                    normalize_x=True,
                    standardize_x=False,
                    standardize_y=True,
                    dropout=0.0,
                    verbose=False):
    pre_data = preprocess_data(data, normalize_x, standardize_x, standardize_y and␣
 ↪regression)
    X = pre_data["X"]
    y = pre_data["y"]
    min_y, max_y = data["min_y"], data["max_y"]

    models = []
    errs = []
    for i in range(k):
        X_train, X_val = get_ith_split_k_fold(X, i, k)
        y_train, y_val = get_ith_split_k_fold(y, i, k)
        input_shape = (X.shape[1], X.shape[2])
        # Initialize model
        model = init_model(input_shape, rnn_layer_fn, rnn_hidden_size,
                           num_rnn_layers, regression, dropout)
        model = train_model(model, X_train, y_train, X_val, y_val,
                           batch_size, epochs, verbose)
        y_pred = model.predict(X_val)
        y_pred = postprocess_data({"y": y_pred, "min_y": min_y, "max_y": max_y},
                                  standardize_y and regression)["y"]
        y_val = postprocess_data({"y": y_val, "min_y": min_y, "max_y": max_y},
                                  standardize_y and regression)["y"]
        if not regression:
            y_pred = np.argmax(y_pred, axis=1)

        err = y_val.flatten() - y_pred.flatten()
        err = np.sum(np.abs(err)) / err.shape[0]
        if verbose:
            print("Fold {}".format(i))
            print("y:\t{}".format(y_val.flatten()))
            print("y(pred):\t{}".format(y_pred.flatten()))
            print("Error:\t{}".format(err))
```

```
            print("")
        models.append(model)
        errs.append(err)

    return mean(errs)
```

Methods used for k-fold training of different RNN configurations.

```
def do_full_search(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec):
    print("--- EDA training ---")
    for sec_frame in [1, 4]:
        data = process_xlsx_data(data_eda["EDA"], data_eda["StateAnxiety"],
                                 eda_elem_per_sec * sec_frame, shuffle=False)
        for rnn in [tf.keras.layers.SimpleRNN, tf.keras.layers.LSTM]:
            for regression in [True, False]:
                for hidden in [32, 64, 128, 256]:
                        err = exp_rnn_k_fold(
                            data, 100, rnn_layer_fn=rnn, rnn_hidden_size=hidden)
                        print("{}, {}, window size (sec): {} hidden size: {} -> {}".
 ↪format(
                            "Regression" if regression else "Classification",
                            rnn.__name__, sec_frame, hidden, err))
    print("--------------------")

    print("---- HR training ---")
    for sec_frame in [16]:
        data = process_xlsx_data(data_hr["HR"], data_hr["StateAnxiety"],
                                 hr_elem_per_sec * sec_frame, shuffle=False)
        for rnn in [tf.keras.layers.SimpleRNN, tf.keras.layers.LSTM]:
            for regression in [True, False]:
                for hidden in [32, 64, 128, 256]:
                        err = exp_rnn_k_fold(
                            data, 100, rnn_layer_fn=rnn, rnn_hidden_size=hidden)
                        print("{}, {}, window size (sec): {} hidden size: {} -> {}".
 ↪format(
                            "Regression" if regression else "Classification",
                            rnn.__name__, sec_frame, hidden, err))
    print("--------------------")

    print("--- EDA + HR training ---")
    for sec_frame in [1, 4]:
        data = process_xlsx_data_merged(data_eda, data_hr, sec_frame,
                                        eda_elem_per_sec, hr_elem_per_sec)

        for rnn in [tf.keras.layers.SimpleRNN, tf.keras.layers.LSTM]:
            for regression in [True, False]:
                for hidden in [32, 64, 128, 256]:
                        err = exp_rnn_k_fold(
                            data, 100, rnn_layer_fn=rnn, rnn_hidden_size=hidden)
                        print("{}, {}, window size (sec): {} hidden size: {} -> {}".
 ↪format(
                            "Regression" if regression else "Classification",
                            rnn.__name__, sec_frame, hidden, err))
```

Model	Avg. Abs. Error
RNN	7.12
LSTM	7.35

Algorithm	Data	Avg. Abs. Error
Classification	**EDA**	**7.14**
	HR	7.48
	EDA+HR	7.45
Regression	EDA	7.08
	HR	7.22
	EDA+HR	**7.06**

Top-4 Training Conf.

Data	Algorithm	RNN type	# lzayers	Window (sec)	Hidden Size	Avg. Abs. Error
EDA	Regression	SimpleRNN	1	4	128	5.97
EDA + HR	Classification	SimpleRNN	1	4	256	6.18
EDA + HR	Regression	SimpleRNN	1	4	128	6.31
HR	Classification	SimpleRNN	1	4	64	6.39

```
    print("-------------------------")

# -- Load data --
eda_elem_per_sec = 4
data_eda = preprocessing.load_eda()  # 1 second per 4 rows
hr_elem_per_sec = 1
data_hr = preprocessing.load_hr() # 1 second per 1 row

# -- Full search --
do_full_search(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec)
```

We did total of 96 iterations of RNN training with 5-fold validation and early stopping criterion. In preliminary investigation we found input normalization crucial for correctly training our RNN models. Without normalization models would often resort to outputting near-mean values regardless of input for low loss. Our visual assessment of input showed very different range of EDA and HR serieses of values per participant. We assume that the fluctuation and tendency per person are a better indicator of hireability than the actual magnitude, thus normalized inputs working better.

We tuned 5 knobs in our model search. The types of output (classification output or regression output), hidden size (32<=h<=256), types of RNN cells (RNN or LSTM), window size (1 second or 4 seconds), and types of data (EDA, HR, EDA + HR). We tried using multi-layer RNN cells instead of a single-layer RNN cell but it didn't help much.

- *Type of output:* Formulating problems as regression showed better results in terms of average absolute error of 5-fold training. Also, classification training was very noisy and hard to reproduce even with same configurations. This should be natural as the the output (i.e. hireability) is ordinal numbers instead of classes. Show regression should be a better fit.
- *Data:* Among classifier models, only using EDA data showed best performance. The regression models showed best performance when utilizing both EDA and HR models, but EDA results closely followed next (by 0.02 difference in loss). Also using 4 second frames instead of 1 second frames showed better average validation loss. We presume that EDA is a better indicator of hireability. But with correct type of output (i.e. regression), model can effectively take hint from HR serieses as well. Our examination of input data showed high noise and this could be the reason for our RNN models better performing with 4 seconds windows.
- *Model:* SimpleRNNs show lower average absolute error than LSTMs and larger hidden size generally show better results. This could be due to the fact that our problem is a simple n-to-1 problem not benefitting from the complexity of LSTM.

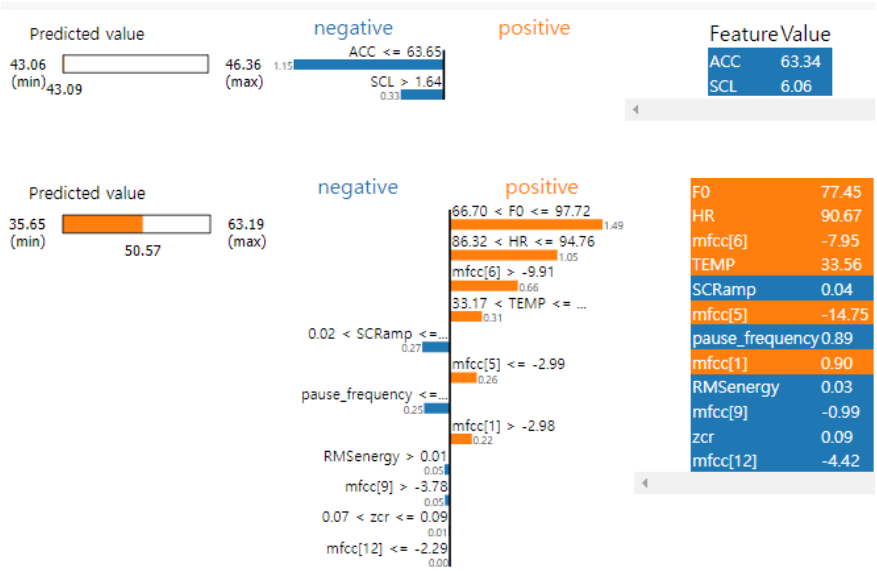
Although we report general trend as above, please note that using same configuration did not guarentee

similar results. Training seemed to be highly affected from the randomness in each training (e.g. initialization, data split), thus low reproducibility. Full results can be found here.

(e) Interpreting the model decisions

(e-i) Model from (b)

We implemented LIME to interpret the model decisions. To see the difference between the feature selection methods, we ran the machine learning algorithm on FNN models with two methods. We used feature subsets that produced the best performance from (b). With the feature subsets, we transformed dataset and separated it into train dataset and test dataset with the ratio of 0.8. Then, we retrained the FNN model with the same structure from (b). The evaluation was taken for one sample from test dataset.
The results show which feature impact the most for the model decision. From (b), selected features with Forward Feature Selection method was ACC and SCL. We could see that ACC gave the most influence to the model decision. In the case for FNN with Fisher's Criterion Feature Selection, F0, which is speech fundamental frequency, gave the most influence to the model decision.




```
import lime
import lime.lime_tabular
from sklearn.model_selection import train_test_split
```

Imported libraries used in (e-i).

```
X = data[wrapper_features[wrapper_idx]]
y = data['StateAnxiety']

train_x, test_x, train_y, test_y = train_test_split(X, y, train_size=0.80)
X = data[wrapper_features[wrapper_idx]]
y = data['StateAnxiety']
cols = X.columns
train_x, test_x, train_y, test_y = train_test_split(X, y, train_size=0.80)

model_wrapper = Sequential(
        [Dense(32, activation='relu',␣
 ↪input_shape=(len(wrapper_features[wrapper_idx]),), activity_regularizer=l2(0.0001)),
```

```
        Dense(64, activation='relu', activity_regularizer=l2(0.0001)),
        Dense(32, activation='relu', activity_regularizer=l2(0.0001)),
        Dense(1)]
)
optimizer = RMSprop(0.001)
model_wrapper.compile(optimizer=optimizer, loss='mse',metrics=['mse'])
model_wrapper.fit(train_x, train_y, epochs=30)
explainer = lime.lime_tabular.LimeTabularExplainer(train_x.to_numpy(),␣
 ↪feature_names=cols, class_names=['StateAnxiety'], verbose=True, mode='regression')
i = 1
exp = explainer.explain_instance(test_x.to_numpy()[i], model_wrapper.predict,␣
 ↪num_features=25)
exp.show_in_notebook(show_table=True)
exp.as_list()
```

Interpreting FNN model with Forward Feature Selection method.

```
X = data[filter_features[filter_idx]]
y = data['StateAnxiety']
cols = X.columns
train_x, test_x, train_y, test_y = train_test_split(X, y, train_size=0.80)

model_filter = Sequential(
        [Dense(32, activation='relu', input_shape=(len(X.columns),),␣
 ↪activity_regularizer=l2(0.0001)),
        Dense(64, activation='relu', activity_regularizer=l2(0.0001)),
        Dense(32, activation='relu', activity_regularizer=l2(0.0001)),
        Dense(1)]
)
optimizer = RMSprop(0.001)
model_filter.compile(optimizer=optimizer, loss='mse',metrics=['mse'])
model_filter.fit(train_x, train_y, epochs=30)
explainer = lime.lime_tabular.LimeTabularExplainer(train_x.to_numpy(),␣
 ↪feature_names=cols, class_names=['StateAnxiety'], verbose=True, mode='regression')
i = 1
exp = explainer.explain_instance(test_x.to_numpy()[i], model_filter.predict,␣
 ↪num_features=25)
exp.show_in_notebook(show_table=True)
exp.as_list()
```

Interpreting FNN model with Fisher's Criterion Feature Selection method.

(e-ii) Model from (d-i)

```
native_data = data[data['Language']==1]
native_data.describe()
```

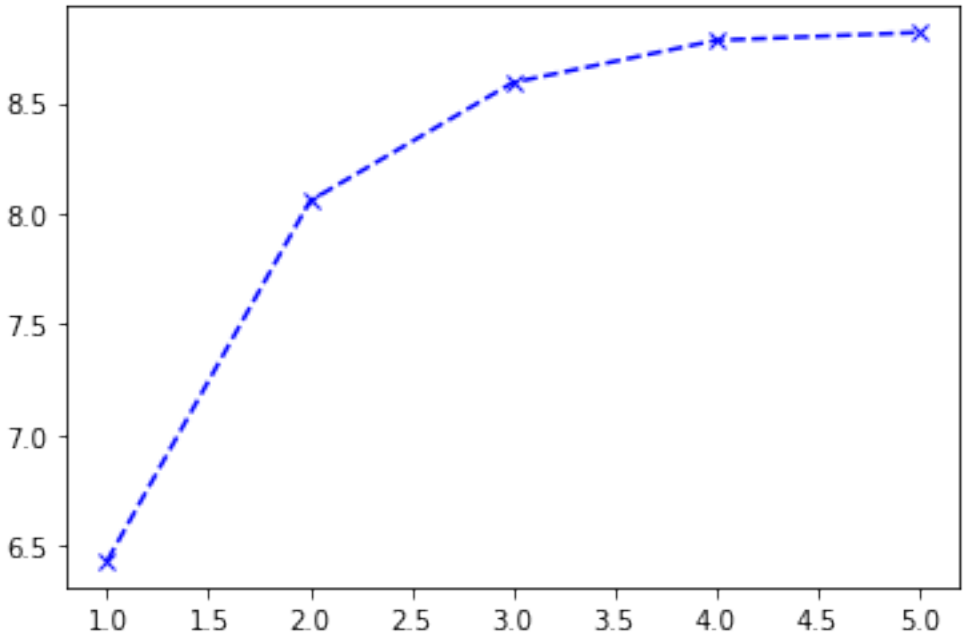
	Language	EDA_Bias	EDA_Slope	HR_Bias	HR_Slope	StateAnxiety
count	37.0	37.000000	37.000000	37.000000	37.000000	37.000000
mean	1.0	0.906025	0.003317	86.562266	0.015142	46.918919
std	0.0	1.146405	0.004820	13.708596	0.098533	10.031557
min	1.0	-0.382165	-0.004240	64.172381	-0.165596	28.000000
25%	1.0	0.244437	0.000675	75.541023	-0.044740	41.000000

50%	1.0	0.549956	0.001940	84.889225	0.005524	48.000000
75%	1.0	1.288176	0.003376	93.608242	0.084634	56.000000
max	1.0	4.996777	0.018874	119.300015	0.205357	64.000000

Unlike result from RNN model, non native speaker dataset showed increase in average error rate. This can lead to a speculation that bias and slope is more correlated to anxiety label of native speakers than non-native. This result might have been derived from the fact that native's anxiety level is reflected with greater covariance than that of non-native dataset.

```
transform(native_data)
```

```
MAE:  StateAnxiety    6.43127
dtype: float64

--------Average Error :StateAnxiety    6.43127
dtype: float64--------
MAE:  StateAnxiety    9.682018
dtype: float64

--------Average Error :StateAnxiety    8.056644
dtype: float64--------
MAE:  StateAnxiety    9.660625
dtype: float64

--------Average Error :StateAnxiety    8.591304
dtype: float64--------
MAE:  StateAnxiety    9.353468
dtype: float64

--------Average Error :StateAnxiety    8.781845
dtype: float64--------
MAE:  StateAnxiety    8.960542
dtype: float64

--------Average Error :StateAnxiety    8.817585
dtype: float64--------
```



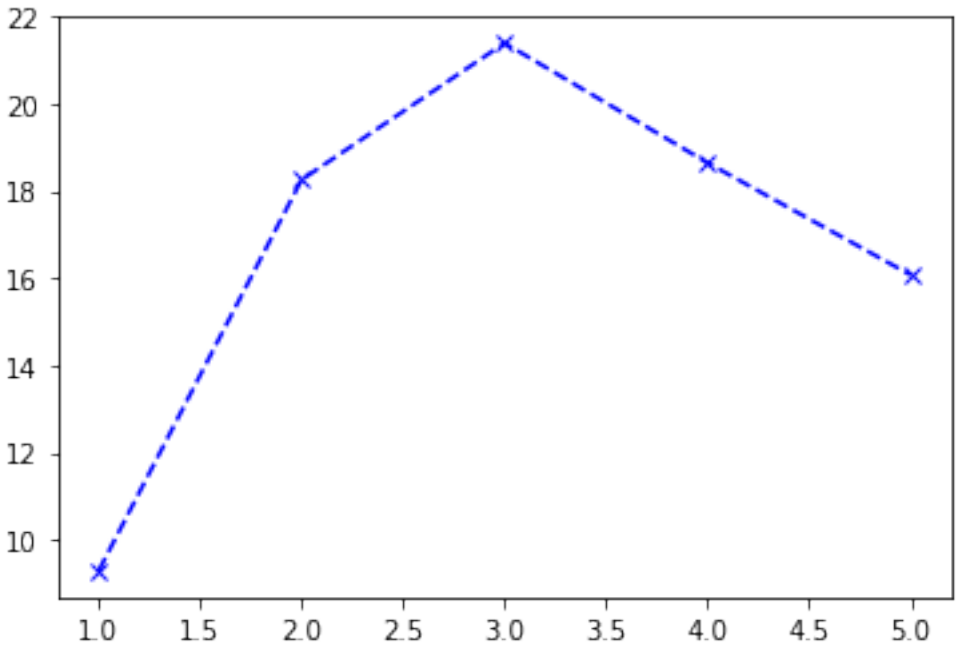

```
non_native_data = data[data['Language']==2]
non_native_data.describe()
```

	Language	EDA_Bias	EDA_Slope	HR_Bias	HR_Slope	StateAnxiety
count	17.0	17.000000	17.000000	17.000000	17.000000	17.000000
mean	2.0	1.196317	0.003231	91.482362	-0.061139	45.529412
std	0.0	1.883243	0.007599	13.043143	0.132847	9.881281
min	2.0	-0.056622	-0.003062	75.166550	-0.303229	27.000000
25%	2.0	0.245933	-0.000019	80.323317	-0.123001	40.000000
50%	2.0	0.625279	0.000610	94.867555	-0.085330	48.000000
75%	2.0	1.192657	0.002600	101.657069	0.022573	52.000000
max	2.0	7.471220	0.029220	119.938652	0.232711	62.000000

```
transform(non_native_data)
```

```
MAE:  StateAnxiety    9.282338
dtype: float64

--------Average Error :StateAnxiety    9.282338
dtype: float64--------
MAE:  StateAnxiety    27.240793
dtype: float64

--------Average Error :StateAnxiety    18.261566
dtype: float64--------
MAE:  StateAnxiety    27.681932
dtype: float64

--------Average Error :StateAnxiety    21.401688
dtype: float64--------
```

```
MAE:  StateAnxiety    10.398513
dtype: float64

--------Average Error :StateAnxiety    18.650894
dtype: float64--------
MAE:  StateAnxiety    5.787387
dtype: float64

--------Average Error :StateAnxiety    16.078193
dtype: float64--------
```



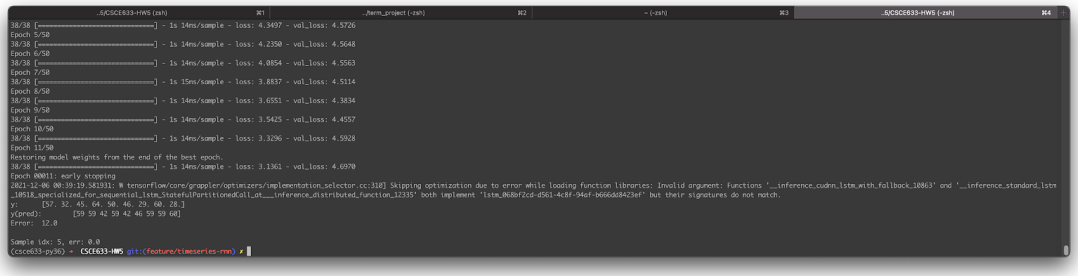
(e-iii) Model from (d-ii)

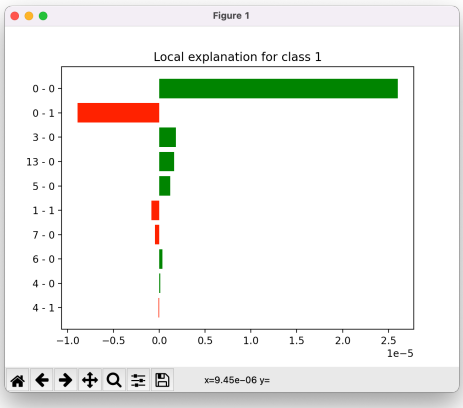
Our code context continues from **(d-ii)**, rather than the previous problem code. And although we show better results with regression results, we conduct our training using the classifier objective as `lime-for-time` only support classifier outputs. Also, `lime-for-time` code has been minimally changed as this.

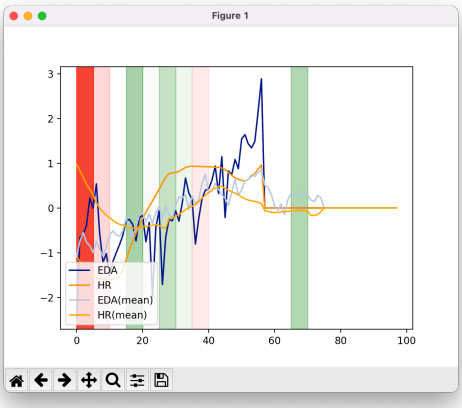
```
def exp_rnn(data, epochs,
            batch_size=128,
            rnn_layer_fn=tf.keras.layers.LSTM,
            rnn_hidden_size=128,
            num_rnn_layers=1,
            regression=True,
            normalize_x=True,
            standardize_x=False,
            standardize_y=True,
            dropout=0.0,
            verbose=False):
    pre_data = preprocess_data(data, normalize_x, standardize_x, standardize_y and␣
 ↪regression)
```

```
    X = pre_data["X"]
    y = pre_data["y"]
    min_y, max_y = data["min_y"], data["max_y"]

    X_train = X[:int(X.shape[0] * 0.7)]
    y_train = y[:int(X.shape[0] * 0.7)]
    X_val = X[int(X.shape[0] * 0.7):int(X.shape[0] * 0.85)]
    y_val = y[int(X.shape[0] * 0.7):int(X.shape[0] * 0.85)]
    X_test = X[int(X.shape[0] * 0.85):]
    y_test = y[int(X.shape[0] * 0.85):]

    input_shape = (X.shape[1], X.shape[2])
    # Initialize model
    model = init_model(input_shape, rnn_layer_fn, rnn_hidden_size,
                       num_rnn_layers, regression, dropout)
    model = train_model(model, X_train, y_train, X_val, y_val,
                        batch_size, epochs, verbose)
    y_pred = model.predict(X_test)
    y_pred = postprocess_data({"y": y_pred, "min_y": min_y, "max_y": max_y},
                              standardize_y and regression)["y"]
    y_test = postprocess_data({"y": y_test, "min_y": min_y, "max_y": max_y},
                              standardize_y and regression)["y"]
    if not regression:
        y_pred = np.argmax(y_pred, axis=1)

    err = y_test.flatten() - y_pred.flatten()
    if verbose:
        print("y:\t{}".format(y_test.flatten()))
        print("y(pred):\t{}".format(y_pred.flatten()))
        print("Error:\t{}".format(np.sum(np.abs(err)) / err.shape[0]))
        print("")

    return model, err, X_test

def lime(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec):
    # Train best classification configuration
    time_frame = 4
    hidden_size = 256
    rnn_layer_fn = tf.keras.layers.SimpleRNN
    data = process_xlsx_data_merged(data_eda, data_hr, time_frame,
                                    eda_elem_per_sec, hr_elem_per_sec)
    model, err, X_test = exp_rnn(data, 50, rnn_layer_fn=rnn_layer_fn,
                                 rnn_hidden_size=hidden_size, regression=False,␣
 ↪verbose=True)

    # Take the better classified example
    num_slices = 24
    num_features = 10
    err = abs(err)
    sample_idx = np.argmin(err)
    series = X_test[sample_idx]
    print("Sample idx: {}, err: {}".format(sample_idx, err[sample_idx]))
    explainer = LimeTimeSeriesExplainer()
```

```
    exp = explainer.explain_instance(
        series, model.predict, num_features=num_features,
        num_samples=100, num_slices=num_slices, replacement_method="total_mean")

    # Plot impact of subseries ver.1
    exp.as_pyplot_figure()  # show the figure
    plt.show()

    # Plot impact of subseries ver.1
    values_per_slice = math.ceil(len(series) / num_slices)
    plt.plot(series[:, 0], color='navy', label='EDA')
    plt.plot(series[:, 1], color='darkorange', label='HR')
    plt.plot(X_test[:, :, 0].mean(axis=0), color='lightsteelblue', label='EDA(mean)')
    plt.plot(X_test[:, :, 1].mean(axis=0), color='orange', label='HR(mean)')
    plt.legend(loc='lower left')
    for i in range(num_features):
        feature, weight = exp.as_list()[i]
        feature = int(feature.split('-')[0].strip())
        start = feature * values_per_slice
        end = start + values_per_slice
        color = 'red' if weight < 0 else 'green'
        plt.axvspan(start , end, ymin=-100, ymax=100, color=color, alpha=abs(weight *␣
↪2 * 1e5))
    plt.show()

lime(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec)
```







We use 4-seconds window size EDA+HR, a simple RNN, 256 hidden size, classifier configuration, closely following one of the better working classification configuration. Note that we are using randomized train-valid-test split rather than 5-fold cross validation we used in **(d-ii)**. So, error is not directly comparable as test sets are different and highly vary per split as the data size is very small.

We show that our RNN models take hints from different regions of the input series for its emitted output. For batching, we used trailing padding and we show that our model mainly focus on non-padding part of the series. Full results can be found here.

(f) Examining individual differences

(f-i) Model from (d-i)

After adopting explainable model to interpret how our FNN model with linear regression inputs affect predicting the label, the value of feature's influence on prediction changed drastically, not conveying similar results for numbers of executions. It has come to question whether slope and bias are really effective in predicting the label. So I have trained a new model with language feature(whehter participant is native speaker or not) and the result of fascinating. Language feature resulted more to affecting the label than any bias and slope combined.

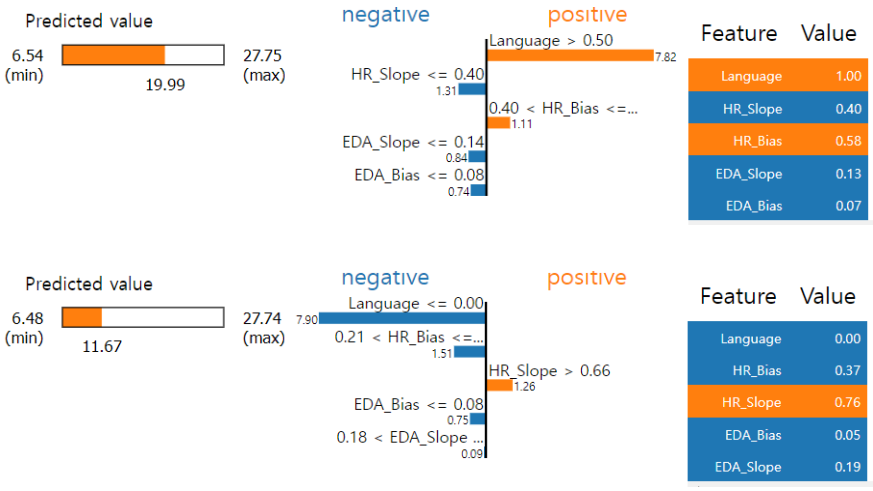
```
#lime
from lime import lime_tabular
from tensorflow.keras.optimizers import RMSprop

scaler = MinMaxScaler()

X = result.drop(columns=['PID','StateAnxiety'])
print(X.describe())
scaler.fit(X)
X = scaler.transform(X)
X = pd.DataFrame(X, columns = ['Language', 'EDA_Bias', 'EDA_Slope' ,'HR_Bias',␣
 ↪'HR_Slope'])
print(X.describe())

y = result['StateAnxiety']
train_x, test_x, train_y, test_y = train_test_split(X, y, train_size=0.80)

model_filter = Sequential(
        [Dense(32, activation='relu', input_shape=(len(X.columns),),␣
 ↪activity_regularizer=l2(0.0001)),
         Dense(64, activation='relu', activity_regularizer=l2(0.01)),
         Dense(32, activation='relu', activity_regularizer=l2(0.01)),
         Dense(1)]
)
optimizer = RMSprop(0.001)
model_filter.compile(optimizer=optimizer, loss='mse',metrics=['mse'])
model_filter.fit(train_x, train_y, epochs=30, verbose=0)
```

```
       Language   EDA_Bias  EDA_Slope     HR_Bias   HR_Slope
count  54.000000  54.000000  54.000000   54.000000  54.000000
mean    1.314815   0.997413   0.003290   88.111185  -0.008873
std     0.468803   1.407795   0.005763   13.576665   0.114897
min     1.000000  -0.382165  -0.004240   64.172381  -0.303229
25%     1.000000   0.244811   0.000197   77.416728  -0.089958
50%     1.000000   0.587617   0.001783   85.833860  -0.004810
```

```
75%      2.000000   1.265215   0.003085   98.234889   0.054935
max      2.000000   7.471220   0.029220  119.938652   0.232711
        Language   EDA_Bias  EDA_Slope    HR_Bias   HR_Slope
count  54.000000  54.000000  54.000000  54.000000  54.000000
mean    0.314815   0.175667   0.225053   0.429270   0.549234
std     0.468803   0.179260   0.172227   0.243457   0.214384
min     0.000000   0.000000   0.000000   0.000000   0.000000
25%     0.000000   0.079835   0.132613   0.237497   0.397938
50%     0.000000   0.123486   0.180020   0.388433   0.556814
75%     1.000000   0.209767   0.218909   0.610808   0.668290
max     1.000000   1.000000   1.000000   1.000000   1.000000
```

```
<keras.callbacks.History at 0x1e6e5459370>
```

```
explainer = lime_tabular.LimeTabularExplainer(train_x.to_numpy(),␣
 ↪feature_names=train_x.columns, class_names=['StateAnxiety'], verbose=True,␣
 ↪mode='regression')
for i in range(5):
    exp = explainer.explain_instance(test_x.to_numpy()[i], model_filter.predict,␣
 ↪num_features=5)
    exp.show_in_notebook(show_table=True)
    exp.as_list()
```



(f-ii) Model from (d-ii)

Our code context continues from **(d-ii)**, rather than the previous problem code.

```
def native_v_non_native(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec):
    time_frame = 4
    hidden_size = 256
    rnn_layer_fn = tf.keras.layers.SimpleRNN
    for lang in [0, 1, 2]:
        errs = []
        data_eda = preprocessing.load_eda(filter_language=lang)
        data_hr = preprocessing.load_hr(filter_language=lang)
        data = process_xlsx_data_merged(data_eda, data_hr, time_frame,
                                        eda_elem_per_sec, hr_elem_per_sec)
        for _ in range(10):
```

```
            _, err, _ = exp_rnn(data, 50, rnn_layer_fn=rnn_layer_fn,
                                rnn_hidden_size=hidden_size, regression=True,␣
 ↪verbose=False)
            errs.append(np.mean(np.abs(err)))
        lang = "native" if lang == 1 else ("non-native" if lang == 2 else "all")
        print("Language: {}, errs: {}".format(lang, mean(errs)))

native_v_non_native(data_eda, data_hr, eda_elem_per_sec, hr_elem_per_sec)
```

Model & Data	Avg. Abs. Err.
All	9.87
Native	12.08
Non-native	7.34

We use 4-seconds window size EDA+HR, a simple RNN, 256 hidden size, classifier configuration, closely following one of the better working classification configuration. Note that we are using randomized train-valid-test split rather than 5-fold cross validation we used in **(d-ii)**. So, error is not directly comparable as test sets are different and highly vary per split as the data size is very small.

For each all/natives/non-natives model, we ran 20 training iterations and averaged the absolute errors of each training run. The non-natives model show decreased loss as compared to all speakers model. But the natives model show increased average absolute error. This indicate that natives are harder to predict from timeseries data only. While there is not much data to pin down an exact reason, we conjecture that as non-native speakers exhibit generally high base anxiety as due to their inconfidence in lack of language skills and additional anxiety from their preparedness, which invites neural networks make predictions solely on the preparedness. However, for native English speakers, the base anxiety level may vary greatly depending on their characters, thus making it harder for the networks to discern their anxiety from preparedness. Full results can be found here.

(g) E-poster

Please refer to the attached E-poster pdf file.

(h) Reporting other teams' work

(1) Team 13
This group found that the wrapper method performed better in terms of computing time and average absolute error.
When working with time-series, their FNN model achieved an average absolute error of 10.35. Their LSTM model for EDA had a average absolute error of 8.168, and 8.513 for HR.
In their conclusions, they state that splitting the data based on native and non-native speakers is an important factor for achieving better models.

(2) Team 12
The work from this team found that for feature selection, when using FNN to train the models, the wrapper method had a lower average absolute error than the filter method. While, the filter method had a shorter computation time.
When working with time-series, this group found that LSTM works best using Heart Rate and EDA signals when predicting anxiety levels.
Lastly, this team found differences in the averages of bio-behavioral features between native and non-native speaking groups.

(3) Team 15
The work from this group was mainly divided based on language, native versus non-native speakers. They found that running models for these features separately gave the best results. For example, the top features that affected the anxiety levels for non-native speakers included speech voice probability which was not included in the top features affecting native speakers.

(4) Team 17
This group found that there is low correlation between all of the bio-behavioral features and the anxiety level. In their feature selection, when comparing the wrapper and filter methods they found that the models performed similarly. Both methods also recommended the same four features, HR, ACC, RIMSEnergy, and IBI.
When working with time-series, this team compared FNN models with increasing polynomial orders. However, there was no significant improvement found between the 3.