DATA ANALYSIS & MACHINE LEARNING ASSIGNMENT

NAME: KUNAH MANOHARAN

STUDENT ID: 22260186

PART 1

A. Dataset description

The first part of assignment is about Extrasensory where the data is of 60 users or participants and each individual participant is identified or allotted with a UUID (Universally Unique Identifier). The data is collected every 1 minute from these participants and consists of thousands of examples by means of ExtraSensory mobile application. A majority of these users were students and research assistants from the UCSD campus. These sensors used were of various types and not all the sensors were available all the time, some of these handsets didn't have a particular sensor or two. Once the data was processed a set of cleaned labels were reported by users themselves. Labels with prefix 'OR_', 'LOC_' or 'FIX_' are processed versions of original labels. There are two types of labels which the users can use to report namely: Main activity and secondary activity.

- In Main Activity – The movement or posture of users are described by these labels. This section is mutually exclusive and there are 7 possible outcomes: sitting, walking, standing and moving, running, cycling, stand stationary, lying down.
- In Secondary Activity - 109 additional labels that describes more specific context in different aspects like sports, transportation, basic needs, company, location and so on. These labels can be applied to an example.

Some of these labels were applied on very rare occasion but they are still useful, sometimes, the users may apply a wrong or irrelevant label and sometimes relevant label is missed by users and is not reported. Due to this reason, cleaning of these labels was performed, and these cleaned versions of the labels were provided. Various testing, validation, increasing the training data, testing is the requirement of the assignment. A starter code was provided and using this starter need to find mean, variance and use the same dataset to run for a different model. C Parameter value also needs to be increase or decrease to test and observe the relation between the accuracy value and C parameter value. 5 random user files were used to calculate test accuracy, balance accuracy and to get mean and variance.

B. <u>SECTION 1: Initial setup and starter code</u>

A set of libraries are imported from the screenshot attached below, that will be useful with the code to get an output. One such library is pandas which is an open-source library that provides high performance data manipulation in Python language. To plot graph matplot library is imported and regression model is imported which will be used in the code. Csv is used to read .csv type files in python and all the dataset files are of

type .csv. Range of functionality is systematized using standartscaler library which is a pre-processing step. All these libraries and class are imported to ensure all the library and parameters are defined in the code.

- **Code**

```
#import library
import pandas as pd
import numpy as np
from pathlib import Path
import sklearn.metrics as metrics
import matplotlib.pyplot as plt
import statistics
import csv
import os

from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.metrics import classification_report
```

a. Directory/folder's location or path

Data directory and the file path are defined, and location are mentioned by assigning it to a variable so that the dataset can be used while completion of code. As said earlier every user/participant was given a UUID which will be unique and can be used for calling a specific folder containing user's data. Any file with no data or blank file is deleted with no labels.

- **Code:**

```
# DataSet file path location:
data_dir = 'C:/Users/Kunah/OneDrive/Documents/Dublin City
University/Assignment/DA && ML/sensor_starter/data'
os.chdir(data_dir)

# Utility functions:
def load_data_for_user(uuid):
    return pd.read_csv(data_dir + '/' + (uuid + '.features_labels.csv'))
def get_features_and_target(df, feature_names, target_name):

    # select out features and target columns and convert to numpy
    X = df[feature_names].to_numpy()
    y = df[target_name].to_numpy()

    # remove examples with no label
```

```
has_label = ~np.isnan(y)
X = X[has_label,:]
y = y[has_label]
return X, y
```

b. Data for one user

At first, data is loaded for a single user's file by means of calling function and the user is randomly selected from the dataset. **.head()** is used to select data of the first 5 users likewise **.tail()** is used to get data for the last five users in a dataset. For the provided dataset the required data is shown in the output depending on the matrix size of the dataset. Initially as per requirement data is loaded for a single user as seen in the below code section.

```
# Data for one user:
df = load_data_for_user('99B204C0-DD5C-4BB7-83E8-A37281B8D769')
df.head()
```

The following screenshot is the output for the above code section. All the available columns are displayed for a user.

Out[1135]:

	timestamp	raw_acc:magnitude_stats:mean	raw_acc:magnitude_stats:std	raw_acc:magnitude_stats:moment3	raw_acc:magnitude_stats:moment4	raw_acc:
0	1444339547	0.926982	0.042822	0.097741	0.159221	
1	1444339597	0.925956	0.003253	0.001480	0.004549	
2	1444339657	0.935604	0.101205	0.137324	0.224176	
3	1444339765	0.931602	0.003507	-0.001307	0.005401	
4	1444339826	0.930136	0.003882	0.002859	0.007062	

5 rows × 278 columns

c. Selection of feature by selecting labels

Various labels were used by the users which were set on their own. For label Bicycling when C-params is set to '0.1', balanced as well as train accuracy are 0.989 and 0.7891 respectively.

When the label is set to walking for same parameter value balance and train accuracy are calculated to 0.9285 and 0.6471.

- **Code**

```
# Features can be selected accordingly by selecting labels

# FIX_Walking, #Bicycling , #Sitting

# select targets

# selecting columns for walking

acc_sensors = [p for p in df.columns if
```

```
            p.startswith('raw_acc:') or
            p.startswith('watch_acceleration:')]
target_column = 'label:FIX_walking'
print(f'target_column')
X_train, y_train = get_features_and_target(df, acc_sensors, target_column)
print(f'{y_train.shape[0]} examples with {y_train.sum()} positives')
scaler = StandardScaler()
imputer = SimpleImputer(strategy='mean')
X_train = scaler.fit_transform(X_train)
X_train = imputer.fit_transform(X_train)
```

- **Output:**

```
Training accuracy of trained model: 0.9285

Out[1679]: 0.9146608315098468
```

Balanced Accuracy

```
n [1680]: # Balanced Accuracy calculation:

y_user1_prediction = kun.predict(X_train)
print(f'Balanced accuracy of train: {metrics.balanced_accuracy_scor

Balanced accuracy of train: 0.6471
```

d. Training the Model

Logistic regression is used to predict the categorical dependent variable with the help of independent variables. Logistic Regression method is used in the code where the parameters for solver, maximum iterations and C value is given as input which can be changed if required. Training accuracy of a trained model is determined by dividing the number of correct predictions by the total number of samples. For the given code training accuracy is around 0.9285.

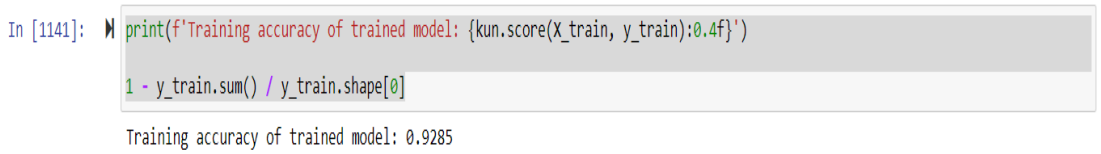
$$\text{Accuracy} = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

- **Code:**

```
# training model:
kun = LogisticRegression(solver='liblinear', max_iter=1000, C=1.0)
kun.fit(X_train, y_train)
```

```
print(f'Training accuracy of trained model: {kun.score(X_train, y_train):0.4f}')
1 - y_train.sum() / y_train.shape[0]
```

- **Output:**



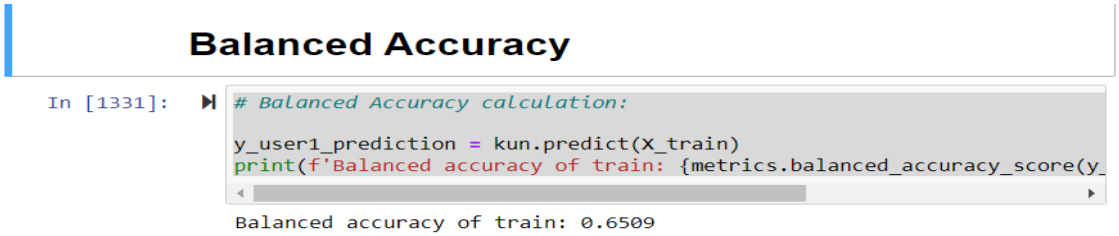
Balanced Accuracy

It is a metric that is used to access the performance of a classification model and is calculated as Balanced Accuracy = (Sensitivity + Specificity)/2. The closer the value of balanced accuracy is to 1 the better the model is able to correctly classify observations.

- **Code**

```
# Balanced Accuracy calculation:
y_user1_prediction = kun.predict(X_train)
print(f'Balanced accuracy of train: {metrics.balanced_accuracy_score(y_train, y_user1_prediction):0.4f}').
```

- **Output:**



Balanced Accuracy of train is 0.7135, which indicates that the training data is not properly evaluated based on testing data. The same code is then tested for a different single user to and to get test accuracy and balanced accuracy for testing purpose.

C. SECTION 2: Improving the Test Set

5 different users are selected in a random order and to test the model on each individually and to find mean and variance. For this all the 5 random users are put into a single string and stored in a list. The list is then appended so that all the values of these 5 users are appended. The data is defined in a loop which prints the Test accuracy and balanced accuracy. The mean is then calculated by using the formula x_mean = x_sum/len(x_list) where x_sum is the sum of list x. For variance use of statistics.variance(x_list) calculates the variance of the entire list and gives an output. The below code and the output are the test accuracy and balanced accuracy also with mean and variance of balanced accuracy. The five users used are

- 7CE37510-56D0-4120-A1CF-0E23351428D2
- 27E04243-B138-4F40-A164-F40B60165CF3
- BEF6C611-50DA-4971-A040-87FB979F3FC1

- A5CDF89D-02A2-4EC1-89F8-F534FDABDD96
- 1DBB0F6F-1F81-4A50-9DF4-CD62ACFA4842

Code:

```
# Selecting 5 different users and testing the model on each indiviually
# Evaluating mean and variance of the balanced accuracy

u_str = [' CCAF77F0-FABB-4F2F-9E24-D56AD0C5A82F', ' 2C32C23E-E30C-498A-8DD2-0EFB9150A02E', 'CF722AA9-2533-4E51-9FEB-9EAC84EE9AAC','00EABED2-271D-49D8-B599-1D4A09240601','59EEFAE0-DEB0-4FFF-9250-54D2A03D0CF2']
x_list = []
def store_list(x) :          # function to store list
    x_list.append(x)
for u in u_str:
    df_test = load_data_for_user(u)
    X_test, y_test = get_features_and_target(df_test, acc_sensors, target_column)
    print("UUID : ", u)
    print(f'{y_train.shape[0]} examples with {y_train.sum()} positives')
    X_test = imputer.transform(scaler.transform(X_test))
    print(f'Test accuracy: {kun.score(X_test, y_test):0.4f}')
    y_pred = kun.predict(X_test)
    print(f'Balanced accuracy (train): {metrics.balanced_accuracy_score(y_test, y_pred):0.4f}')
    x = float(format(metrics.balanced_accuracy_score(y_test, y_pred)))
    store_list(x)
    print(" ")
x_sum = sum(x_list)
x_mean = x_sum/len(x_list)
print("Mean of Balanced Accuracy : ",x_mean)

var = statistics.variance(x_list)
print("Variance of Balanced Accuracy : ",var)
```

- **Output:**

```
UUID :  7CE37510-56D0-4120-A1CF-0E23351428D2
5484 examples with 468.0 positives
Test accuracy: 0.9269
Balanced accuracy (train): 0.6968

UUID :  27E04243-B138-4F40-A164-F40B60165CF3
5484 examples with 468.0 positives
Test accuracy: 0.9300
Balanced accuracy (train): 0.5299

UUID :  BEF6C611-50DA-4971-A040-87FB979F3FC1
5484 examples with 468.0 positives
Test accuracy: 0.9183
Balanced accuracy (train): 0.7453

UUID :  A5CDF89D-02A2-4EC1-89F8-F534FDABDD96
5484 examples with 468.0 positives
Test accuracy: 0.9322
Balanced accuracy (train): 0.5305
```

```
Balanced accuracy (train): 0.5299

UUID :  BEF6C611-50DA-4971-A040-87FB979F3FC1
5484 examples with 468.0 positives
Test accuracy: 0.9183
Balanced accuracy (train): 0.7453

UUID :  A5CDF89D-02A2-4EC1-89F8-F534FDABDD96
5484 examples with 468.0 positives
Test accuracy: 0.9322
Balanced accuracy (train): 0.5305

UUID :  1DBB0F6F-1F81-4A50-9DF4-CD62ACFA4842
5484 examples with 468.0 positives
Test accuracy: 0.8808
Balanced accuracy (train): 0.5847

Mean of Balanced Accuracy :  0.6174306273774179
Variance of Balanced Accuracy :  0.009731636806674503
```

Mean and Variance of balanced accuracy is 0.6174306 and 0.009731 respectively. Kernel used here is "rbf" and for classification – Support Vector classifier is used which is stored in variable kun. Radial Basis function network is more popular kernels in machine learning for non-linear data, it's the most used kernel in SVM (Support Vector Machine).

D. SECTION 3: Increase training data

Combining the same 5 users and adding them to a data frame named df_combined, in this step a new folder named "Kunah_csv.csv" is created which is a combination of all the data for the required 5 user. From sklearn model post importing train_test_split library which is used to help in splitting the data in a required percentage. In this case the data is split into two subsets. The two subsets are – Training set = 80 and Validation set = 20 whereas the test_size = 0.2. C parameter value is set to 1.0 and maximum iteration is valued to 1000 and by using linear logistic regression model training accuracy and balanced accuracy for the larger dataset is calculated. From the above screenshot, training accuracy is calculated at 0.9305 and balance accuracy is 0.6167. The value of balance accuracy seems low but still decent enough to be considered to be

- **Output**

```
In [1349]: print(f'LR with C= 1.0 Training accuracy: {kun.score(X_train, y_train):

LR with C= 1.0 Training accuracy: 0.9305

In [1350]: 1 - y_train.sum() / y_train.shape[0]

Out[1350]: 0.9147481194438113

In [1351]: X_val = scaler.transform(X_val)
           X_val = imputer.transform(X_val)
           y_pred = kun.predict(X_val)
           print(f'Balanced accuracy (train): {metrics.balanced_accuracy_score(y_

Balanced accuracy (train): 0.6167
```

used. The three types of values precision, recall and f1-score where the later is calculated as

F1 Score = 2 * (Precision * Recall) / (Precision + Recall)

Where, precision is the positive predictions relative to total positive predictions which are correct. Recall is the positive predictions relative to total actual positives which are correct. As seen in the below diagram, when precision and recall values are equal to 0.93 and 0.98 respectively. **F1-Score is 2 * (0.9114) / (1.91) = 2 * 0.47717 = 0.96**

- **Output**

LR with C = 1.0 Classification Report

	precision	recall	f1-score	support
0.0	0.93	0.98	0.96	1003
1.0	0.52	0.26	0.34	94
accuracy			0.92	1097
macro avg	0.73	0.62	0.65	1097
weighted avg	0.90	0.92	0.90	1097

Classification report when C = 1.0, f1-score is relatively high for 0.0 there's a significant decrease in f1-score when the C changes from 0 to 1. The F1-Score also varies from 0.96 to 0.34 respectively. The balanced accuracy when C = 1.0 is 0.6167 whereas when C = 0.5, balance accuracy is 0.6546 which clearly states that C-Parameter is inversely proportional to the accuracy. The lower the C-parameter value the higher the accuracy.

- **Output**

LR with C = 0.5 Classification Report

	precision	recall	f1-score	support
0.0	0.94	0.96	0.95	1003
1.0	0.44	0.35	0.39	94
accuracy			0.91	1097
macro avg	0.69	0.65	0.67	1097
weighted avg	0.90	0.91	0.90	1097

E. SECTION 4: Different Kernel types

a. Kernel = Linear and C = 1.0

A Linear SVC's (Support Vector Classifier) goal is to split or categorize the data you supply by producing a "best fit" hyperplane. You may then feed some characteristics to your classifier to get the "predicted" class after acquiring the hyperplane. This makes this particular algorithm—which may be used in a variety of circumstances—pretty ideal for multiple purpose. By using SVC classifier and kernel is set to linear and C-Parameter to 1, training accuracy is 0.9300 and balance accuracy at 0.6232. Since balance accuracy for this model using linear is reduced indicates the model isn't compatible.

- **Code**

```
from sklearn.svm import SVC
kun1 = SVC(C=1.0, kernel='linear', gamma= 1)
kun1.fit(X_train, y_train)
print(f'SVM Training accuracy for C = 1 : {kun1.score(X_train, y_train):0.4f}')
1 - y_train.sum() / y_train.shape[0]
X_val = scaler.transform(X_val)
X_val = imputer.transform(X_val)
y_pred = kun1.predict(X_val)
print(f'Balanced accuracy (train): {metrics.balanced_accuracy_score(y_val, y_pred):0.4f}')
print('SVM with linear kernel Classification Report\n when C = 1')
print(classification_report(y_val, y_pred))
```

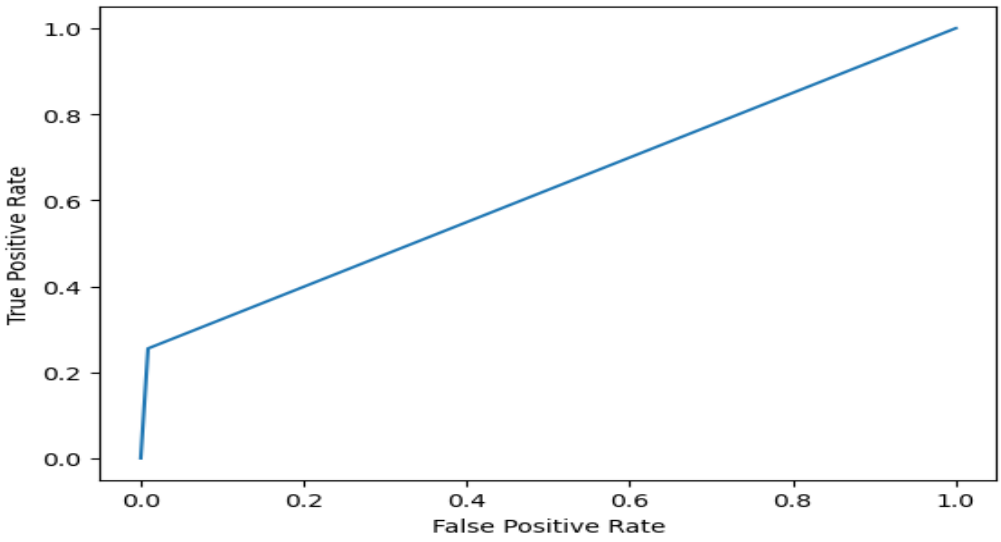
- **Output**

```
Balanced accuracy (train): 0.6232

SVM with linear kernel Classification Report
 when C = 1
              precision    recall  f1-score   support

         0.0       0.93      0.99      0.96      1003
         1.0       0.73      0.26      0.38        94

    accuracy                           0.93      1097
   macro avg       0.83      0.62      0.67      1097
weighted avg       0.92      0.93      0.91      1097
```

- **ROC Graph Curve**
  Code

```
fpr, tpr, _ = metrics.roc_curve(y_val,  y_pred)
auc = metrics.roc_auc_score(y_val, y_pred)
print("AUC for our data is : ",auc)
plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

- **ROC CURVE**



b. Kernel = Poly and C = 0.1

Polynomial kernel in machine learning is a function commonly used with SVMs and other kernelized models, it represents similarity of vectors and can be used in learning non-linear models. The second type of kernel used here is poly with C parameter value set to 0.1. The difference in the value observed is balance accuracy value is increased to 0.7061 from 0.6232. This clearly indicates that the lower the C params value the better the accuracy performance of a model. This also indicates that poly and c = 0.1 is the model is clearly able to classify observations.

- **Code**

```
from sklearn.svm import SVC
kun2 = SVC(C = .1, kernel='poly', gamma= 1)
kun2.fit(X_train, y_train)
print(f'Training accuracy: {kun2.score(X_train, y_train):0.4f}')
1 - y_train.sum() / y_train.shape[0]
X_val = scaler.transform(X_val)
X_val = imputer.transform(X_val)
y_pred = kun2.predict(X_val)
print(f'Balanced accuracy (train): {metrics.balanced_accuracy_score(y_val, y_pred):0.4f}')
print('SVM with poly kernel Classification Report\n when C = 0.01')
print(classification_report(y_val, y_pred))
```

- **Output**

```
Balanced accuracy (train): 0.7061
```

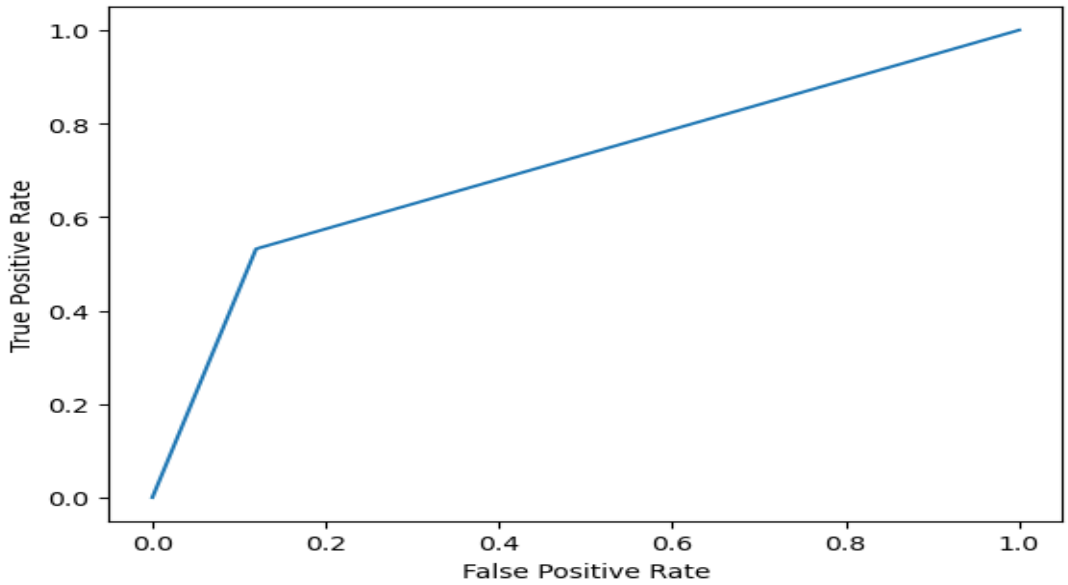
```
SVM with poly kernel Classification Report
 when C = 0.01
              precision    recall  f1-score   support

         0.0       0.95      0.88      0.92      1003
         1.0       0.29      0.53      0.38        94

    accuracy                           0.85      1097
   macro avg       0.62      0.71      0.65      1097
weighted avg       0.90      0.85      0.87      1097
```

- **ROC Graph Curve**

- **Code**

```
fpr, tpr, _ = metrics.roc_curve(y_val,  y_pred)
auc = metrics.roc_auc_score(y_val, y_pred)
print("AUC for our data is : ",auc)
plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

- **ROC Curve**



c. Kernel = Poly and C = 1.0

For a combination of these values since the value of C is higher compared to previous testing model and hence the test accuracy is reduced.

- **Code**

```
kun3 = SVC(C= 1.0, kernel='poly', gamma= 5)
kun3.fit(X_train, y_train)
```

```
print(f'Training accuracy: {kun3.score(X_train, y_train):0.4f}')
1 - y_train.sum() / y_train.shape[0]
X_val = scaler.transform(X_val)
X_val = imputer.transform(X_val)
y_pred = kun.predict(X_val)
print(f'Balanced accuracy (train): {metrics.balanced_accuracy_score(y_val,
y_pred):0.4f}')
print('SVM with poly kernel Classification Report\n')
print(classification_report(y_val, y_pred))
```

- **Output**

```
SVM with poly kernel Classification Report

              precision    recall  f1-score   support

         0.0       0.95      0.82      0.88      1003
         1.0       0.21      0.52      0.30        94

    accuracy                           0.79      1097
   macro avg       0.58      0.67      0.59      1097
weighted avg       0.88      0.79      0.83      1097
```

PART 2

A. Dataset description

In this section, the features found on Mars' surface are identified from pictures using deep convolution neural networks. This section has two characteristics, namely fans and blotches. First, the code must be trained for five epochs (passes through the dataset). Initial ResNet50 model and SGD-based optimizer (Stochastic Gradient Descent). SGD is a broad optimization algorithm that can solve a variety of issues at their best. The size of the steps, which is regulated by learning rate hyperparameters, is a crucial GD parameter. The algorithm will need to go through many iterations to converge, which will take a lot of time, if the learning rate is too little. If it is too high, the model may even outperform the ideal value.

Code:

```
#Importing libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch
import torch.nn.functional as F
```

```
import torch.nn as nn
import torch.optim as optim
import torchvision.transforms as transforms
import torchvision.models as models
import sklearn.metrics as metrics
import tqdm
from torch.utils.data import DataLoader
from torchvision.datasets.folder import pil_loader
from pathlib import Path
from PIL import Image
from torchvision.models import resnet50, ResNet50_Weights
#Stating if CPU or GPU is used:
device = 'cpu'
data = str()
#Dataset
class PlanetFourDataset(object):
  def __init__(self, split='train', transform=None, loader=pil_loader):
    super().__init__()
    self.split = split
    self.base_dir = 'C:/Users/Kunah/OneDrive/Documents/Dublin City University/Assignment/DA && ML/planetfour 2/planetfour'
    self.image_dir = self.base_dir + '/' + split
    self.labels_file = self.base_dir + '/' + (split + '.csv')
    self.labels_df = pd.read_csv(self.labels_file)
    self.transform = transform
    self.loader = loader
  def __getitem__(self, index):
    row = self.labels_df.iloc[index]
    filename = self.image_dir + '/' + (row.tile_id + '.jpg')
    fans = int(row.fans)
    blotches = int(row.blotches)
    image = self.loader(str(filename))
    if self.transform is not None:
      image = self.transform(image)
```

```
        return image, torch.tensor([fans, blotches], dtype=torch.float32)
    def __len__(self):
        return len(self.labels_df)
#Data Augmentation
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
valid_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
])
#Data Loaders
train_set = PlanetFourDataset('train', transform=train_transform)
valid_set = PlanetFourDataset('valid', transform=train_transform)
train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
valid_loader = DataLoader(valid_set, batch_size=64, shuffle=False)
#Load a pretrained model
model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V2)
model.fc = nn.Linear(2048, 2)
model.to(device);
#Loss
criterion = nn.BCEWithLogitsLoss()
#Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

#Training and validation functions
avg_train_losses = []
avg_valid_losses = []
valid_accuracies = []
trainloss_list = []
validloss_list = []
```

```
def make_list(x,y):
    trainloss_list.append(x)
    validloss_list.append(y)
def train_for_epoch(optimizer):
    model.train()
    train_losses = []
    for batch, target in tqdm.tqdm(train_loader):

        # data to GPU
        batch = batch.to(device)
        target = target.to(device)

        # reset optimizer
        optimizer.zero_grad()

        # forward pass
        predictions = model(batch)

        #breakpoint()
        # calculate loss
        loss = criterion(predictions, target)

        # backward pass
        loss.backward()
        # parameter update
        optimizer.step()

        # track loss
        train_losses.append(float(loss.item()))
    train_losses = np.array(train_losses)
    return train_losses
def validate():
    model.eval()
```

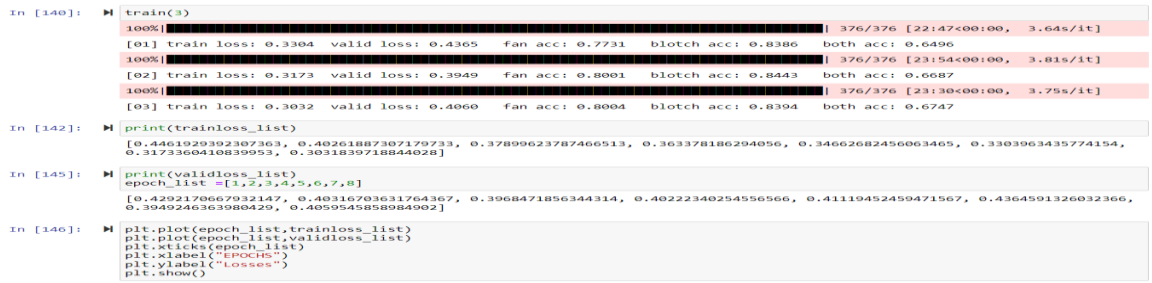
```
valid_losses = []
y_true, y_prob = [], []
with torch.no_grad():
  for batch, target in valid_loader

    # move data to the device
    batch = batch.to(device)
    target = target.to(device)

    # make predictions
    predictions = model(batch)

    # calculate loss
    loss = criterion(predictions, target)

    # logits -> probabilities
    torch.sigmoid_(predictions)

    # track losses and predictions
    valid_losses.append(float(loss.item()))
    y_true.extend(target.cpu().numpy())
    y_prob.extend(predictions.cpu().numpy(
y_true = np.array(y_true)
y_prob = np.array(y_prob)
y_pred = y_prob > 0.5
valid_losses = np.array(valid_losses)

# calculate validation accuracy from y_true and y_pred
fan_accuracy = metrics.accuracy_score(y_true[:,0], y_pred[:,0])
blotch_accuracy = metrics.accuracy_score(y_true[:,1], y_pred[:,1])
exact_accuracy = np.all(y_true == y_pred, axis=1).mean()
# calculate the mean validation loss
valid_loss = valid_losses.mean()
```

```
    return valid_loss, fan_accuracy, blotch_accuracy, exact_accuracy


def train(epochs, first_epoch=1):
    for epoch in range(first_epoch, epochs+first_epoch):

        # train
        train_loss = train_for_epoch(optimizer)

        # validation
        valid_loss, fan_accuracy, blotch_accuracy, both_accuracy = validate()
        print(f'[{epoch:02d}] train loss: {train_loss.mean():0.04f}  '
            f'valid loss: {valid_loss:0.04f}  ',
            f'fan acc: {fan_accuracy:0.04f}  ',
            f'blotch acc: {blotch_accuracy:0.04f}  ',
            f'both acc: {both_accuracy:0.04f}'
        )
        make_list(train_loss.mean(),valid_loss)
        # update learning curves
        avg_train_losses.append(train_loss.mean())
        avg_valid_losses.append(valid_loss)
        valid_accuracies.append((fan_accuracy, blotch_accuracy, both_accuracy))
print('trainloss_list = ', trainloss_list)
print('validloss_list = ', validloss_list)
        # save checkpoint
        #torch.save(model, f'checkpoints/baseline_{epoch:03d}.pkl')
def constant_clf_accuracy():
    y_true, y_pred = [], []
    with torch.no_grad():
        for _, target in valid_loader:
            y_true.extend(target.cpu().numpy())
            y_pred.extend(np.ones((target.shape[0], 2), dtype=np.float32))
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)
```

```
    # calculate validation accuracy from y_true and y_pred
    f = metrics.accuracy_score(y_true[:,0], y_pred[:,0])
    b = metrics.accuracy_score(y_true[:,1], y_pred[:,1])
    t = np.all(y_true == y_pred, axis=1).mean()
    print(f'fan: {f}  blotch: {b}  both: {t}')
constant_clf_accuracy()
train(5)
print(trainloss_list)
print(validloss_list)
epoch_list =[1,2,3,4,5]
plt.plot(epoch_list,trainloss_list)
plt.plot(epoch_list,validloss_list)
plt.xticks(epoch_list)
plt.xlabel("EPOCHS")
plt.ylabel("Losses")
plt.show()
```

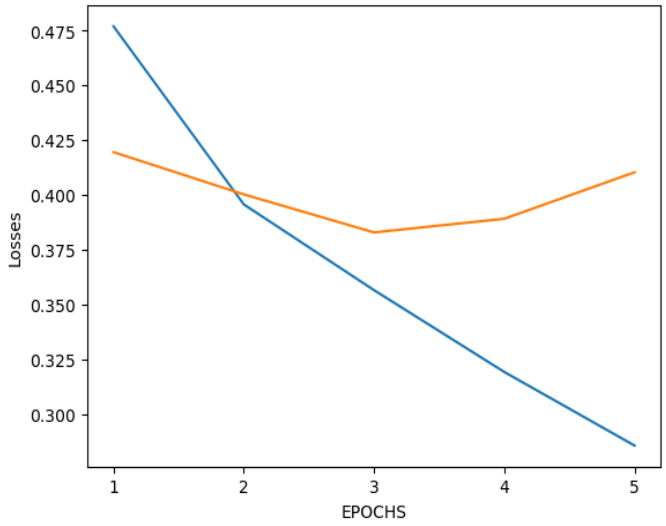
- **Output: For first 5 epochs using Adam:**

- **Output for 3 more epochs using Adam:**



Output Learning Graph using SGD optimizer:

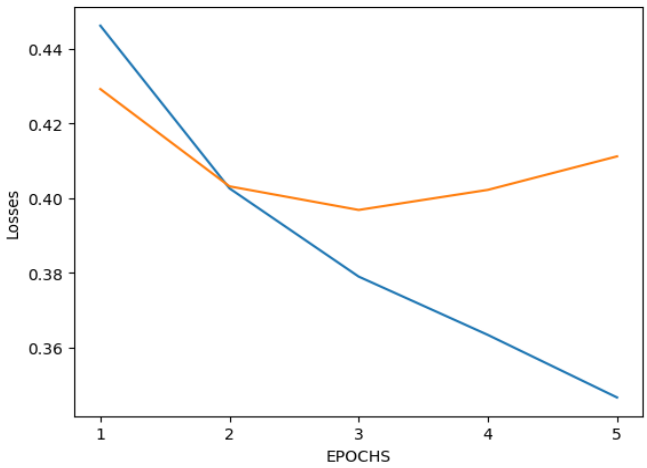
In the below graph, it is clear that the train loss decreases with each training epoch and since it's the first time the decrease in loss would be slower than expected. Also, since the number of trainings performed is less, in graph it is observed that at certain points the value suddenly drops.



Output Learning Graph using Adam optimizer:

- **<u>For 1st 5 Epochs</u>**

The blue line indicates train loss whereas the orange line indicates valid loss plots, which is gradually decreasing since compared between the first 5 values of epoch. This is due to the more a model is trained, gradual decrease in the loss since in each epoch the model goes through the same set of data multiple times so the chances of error to occur and to miss out an image to determine fans and blotch accuracy should have a steady increase. The higher the number of epochs the better and higher the accuracy would be.



In the second test, optimizer used is Adam and the major difference between SGD and Adam optim is SGD takes more time to complete and calculate the train epochs. Adam optimizer is used as a replacement for the optimizer for gradient descent. The below plot is a combination of 8 epochs and there's a steady decrease in the number of train loss. For

valid loss for 8 epochs we observe a sudden spike in the value this is due to some images might have poor predictions that keeps getting worse. Ideally, both the loss value is expected to decrease. Also, poor predictions are penalized more strongly than the good predictions are rewarded. Finally, it can be concluded that to achieve higher accuracy and less loss for any given model, it needs to be trained as many times as possible.

- **For 8 Epochs**