CS484 Introduction to Machine Learning

Problem 4 (30 Points): Cross-Validation on Classification Models

```
#libraries Requireds
import numpy as np
import pandas as pd
import sklearn
#importing datasets and data preprocessing libraries
from sklearn.datasets import load_iris, load_breast_cancer, fetch_20newsgroups_vectorized, fetch_20newsgroups
from sklearn.model_selection import train_test_split
from scipy.sparse import vstack
#imporing the classification models
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
#importing performance metrics libraries
from sklearn.metrics import accuracy_score
#in order to prevent printing warnings in the output, i am importing warnings and exception librarie
import warnings
from sklearn.exceptions import ConvergenceWarning # Suppressing convergence warnings
# For Newsgroup dataset, due to large scale data, I am using SVD as a dimensional
# reduction technique, so i can prevent the issue of memory error
from sklearn.decomposition import TruncatedSVD
```

Analysing the 3 datasets which I will be using for classication training.

```
# Note: As required from the question, I will firstly load the complete dataset then later
# use train test split library to split it into 60:40 ratio
#1
print("Analyzing Iris dataset \n -------------------------")
iris_dataset = load_iris()
print(f"Displaying the first 5 rows of the feature matrix: {iris_dataset.data[:5, :]}")
print(f"Showing the shape of the feature matrix: {iris_dataset.data.shape}")
print(f"Displaying the first 5 values of the target vector: {iris_dataset.target[:5]}")
print(f"Listing the class names in the Iris dataset: {iris_dataset.target_names}")
print(f"Showing the shape of the target vector: {iris_dataset.target.shape}")


#2
print("Analyzing breast cancer dataset \n -------------------------")
breast_cancer_dataset = load_breast_cancer()
print(f"Displaying the first 5 rows of the feature matrix: {breast_cancer_dataset.data[:5, :]}")
print(f"Showing the shape of the feature matrix: {breast_cancer_dataset.data.shape}")
print(f"Displaying the first 5 values of the target vector: {breast_cancer_dataset.target[:5]}")
print(f"Listing the class names in the Breast Cancer dataset: {breast_cancer_dataset.target_names}")
print(f"Showing the shape of the target vector: {breast_cancer_dataset.target.shape}")

#3
print("Analyzing 20 News group dataset \n -------------------------")
news_group_dataset = sklearn.datasets.fetch_20newsgroups_vectorized(subset='all')
print(f"Displaying the first 5 rows of the feature matrix: {news_group_dataset.data[:5, :]}")
print(f"Showing the shape of the feature matrix: {news_group_dataset.data.shape}")
print(f"Displaying the first 5 values of the target vector: {news_group_dataset.target[:5]}")
print(f"Listing the class names in the 20 News Group dataset: {news_group_dataset.target_names}")
print(f"Showing the shape of the target vector: {news_group_dataset.target.shape}")
```

```
Analyzing Iris dataset
 -------------------------
Displaying the first 5 rows of the feature matrix: [[5.1 3.5 1.4 0.2]
 [4.9 3.  1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.  3.6 1.4 0.2]]
Showing the shape of the feature matrix: (150, 4)
Displaying the first 5 values of the target vector: [0 0 0 0 0]
Listing the class names in the Iris dataset: ['setosa' 'versicolor' 'virginica']
Showing the shape of the target vector: (150,)
Analyzing breast cancer dataset
 -------------------------
Displaying the first 5 rows of the feature matrix: [[1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
  1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
  6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
  1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
  4.601e-01 1.189e-01]
 [2.057e+01 1.777e+01 1.329e+02 1.326e+03 8.474e-02 7.864e-02 8.690e-02
  7.017e-02 1.812e-01 5.667e-02 5.435e-01 7.339e-01 3.398e+00 7.408e+01
  5.225e-03 1.308e-02 1.860e-02 1.340e-02 1.389e-02 3.532e-03 2.499e+01
  2.341e+01 1.588e+02 1.956e+03 1.238e-01 1.866e-01 2.416e-01 1.860e-01
  2.750e-01 8.902e-02]
 [1.969e+01 2.125e+01 1.300e+02 1.203e+03 1.096e-01 1.599e-01 1.974e-01
  1.279e-01 2.069e-01 5.999e-02 7.456e-01 7.869e-01 4.585e+00 9.403e+01
  6.150e-03 4.006e-02 3.832e-02 2.058e-02 2.250e-02 4.571e-03 2.357e+01
  2.553e+01 1.525e+02 1.709e+03 1.444e-01 4.245e-01 4.504e-01 2.430e-01
  3.613e-01 8.758e-02]
 [1.142e+01 2.038e+01 7.758e+01 3.861e+02 1.425e-01 2.839e-01 2.414e-01
  1.052e-01 2.597e-01 9.744e-02 4.956e-01 1.156e+00 3.445e+00 2.723e+01
  9.110e-03 7.458e-02 5.661e-02 1.867e-02 5.963e-02 9.208e-03 1.491e+01
  2.650e+01 9.887e+01 5.677e+02 2.098e-01 8.663e-01 6.869e-01 2.575e-01
  6.638e-01 1.730e-01]
 [2.029e+01 1.434e+01 1.351e+02 1.297e+03 1.003e-01 1.328e-01 1.980e-01
  1.043e-01 1.809e-01 5.883e-02 7.572e-01 7.813e-01 5.438e+00 9.444e+01
  1.149e-02 2.461e-02 5.688e-02 1.885e-02 1.756e-02 5.115e-03 2.254e+01
  1.667e+01 1.522e+02 1.575e+03 1.374e-01 2.050e-01 4.000e-01 1.625e-01
  2.364e-01 7.678e-02]]
Showing the shape of the feature matrix: (569, 30)
Displaying the first 5 values of the target vector: [0 0 0 0 0]
Listing the class names in the Breast Cancer dataset: ['malignant' 'benign']
Showing the shape of the target vector: (569,)

  (4, 111322)   0.06468462273531508
  (4, 112421)   0.06468462273531508
  (4, 114440)   0.19405386820594528
  (4, 114455)   0.06468462273531508
  (4, 114520)   0.06468462273531508
  (4, 114579)   0.06468462273531508
  (4, 114625)   0.06468462273531508
  (4, 114646)   0.12936924547063017
  (4, 114692)   0.06468462273531508
  (4, 114731)   0.06468462273531508
  (4, 115475)   0.12936924547063017
  (4, 116636)   0.06468462273531508
  (4, 116665)   0.06468462273531508
  (4, 116722)   0.06468462273531508
  (4, 118280)   0.06468462273531508
  (4, 119714)   0.06468462273531508
  (4, 123292)   0.06468462273531508
  (4, 123796)   0.06468462273531508
  (4, 124026)   0.06468462273531508
  (4, 124046)   0.19405386820594528
  (4, 124616)   0.12936924547063017
  (4, 124946)   0.06468462273531508
  (4, 125074)   0.06468462273531508
  (4, 128402)   0.32342311367657545
  (4, 128420)   0.06468462273531508
Showing the shape of the feature matrix: (18846, 130107)
Displaying the first 5 values of the target vector: [17  7 10 10  7]
Listing the class names in the 20 News Group dataset: ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.
mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles', 'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronic
s', 'sci.med', 'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast', 'talk.politics.misc', 'talk.religion.misc']
Showing the shape of the target vector: (18846,)
```

So, the above results give a clear understanding of the dataset which we are going to train, Iris dataset and breast cancer dataset seems to have less instances and easy to train. But the news group data has sparse data as it indicates the data matrix is text embeddings of 18846 samples with 130107 attributed in total. This is going to be tricky to handle especially when we perform cross fold validation for hyperparameter tuning which leads to very long time in training and apply cross fold validation. So, I will write two functions for cross fold validation, one for iris and cancer dataset

and one for news group dataset. This is mainly because in in cross fold validation, I concatenate the rest of dataset [excluding the portion given to validation] and use it in train set, but in the case of news group data, as it is sparse data, instead of concatenating, I will vstack them, so the arrays inside the array will add up.

So, cross validation 1 --> iris, cancer data

cross validation 2 --> News group data

My approach for defining cross validation function:

I will start with writing a function to split the data into train and test by 60:40 ration, then define a function for cross validation with 10 folds.

For cross validation:

1. No. of folds = no. of iterations.

2. length of each fold = Total samples / no. of folds.

3. So, for 10 folds, we have 10 equal length data of fold size.

4. The goal is to use all the hyperparameters separately and for each parameter cross validation has to be performed. For this I will create nested loops with hyper-parameters being the parent loop then, in each iteration, the validation set will be assigned with all 10 equal length data of fold size, this will make sure, all samples are used in the validation.

5. Then the rest will be concatenated and used for training.

6. Will obtain the accuracies for each fold and average it to find average accuracy of model classification for each hyper parameter.

```python
# Splitting the dataset into 60% training and 40% testing sets
def split_data(features, targets):
    return train_test_split(features, targets, test_size=0.4, random_state=42)
    # Using 60% of the data for performing cross-validation analysis

def cross_validation_1(model, X_train, y_train, hyperparameter_name, hyperparameter_values):

    # Handling sparse matrices using the shape property
    num_samples = X_train.shape[0]  # Counting the number of samples
    fold_size = num_samples // 10  # Determining the size of each fold
    mean_accuracies = []  # Initializing a list to store mean accuracies for each hyperparameter value

    for param_value in hyperparameter_values:
        fold_accuracies = []  # Initializing a list to store accuracies for the current hyperparameter value
        for fold in range(10):
            start_index, end_index = fold * fold_size, (fold + 1) * fold_size
            # Extracting the validation set for the current fold
            X_val, y_val = X_train[start_index:end_index], y_train[start_index:end_index]
            # Combining the remaining data to form the training set
            X_train_fold = np.concatenate((X_train[:start_index], X_train[end_index:]), axis=0)
            y_train_fold = np.concatenate((y_train[:start_index], y_train[end_index:]), axis=0)

            # Setting the model's hyperparameter and training
            setattr(model, hyperparameter_name, param_value)
            model.fit(X_train_fold, y_train_fold)

            # Validating the model and calculating accuracy
            y_pred = model.predict(X_val)
            fold_accuracies.append(accuracy_score(y_val, y_pred))

        # Calculating mean accuracy for the current hyperparameter value
        mean_accuracies.append(np.mean(fold_accuracies))

    return mean_accuracies
```

Its given in the question that hidden layer sizes = {1/10,1/5,1/2,1,2,5,10} × data features
(If hidden layer sizes < 1, ignore it).
So i will write a function to find the parameters in hidden layer from given values and multiply with input features.

```python
def hidden_neurons(input_feature, hidden_layer_sizes):
    return [max(1, int(h * input_feature)) for h in hidden_layer_sizes]
```

I am going to train firstly on Iris and cancer, later using modified cross validation, i will train on news group data in next stage

```
# given hyper parameters in the question:
C_values = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
hidden_layer_sizes = [1/10, 1/5, 1/2, 1, 2, 5, 10]
datasets = {
    "Iris data": (iris_dataset.data, iris_dataset.target),
    "Breast Cancer data": (breast_cancer_dataset.data, breast_cancer_dataset.target)
}

# Iterating over the 2 datasets, performing cross-validation for hyperparameter tuning, and
# reporting the optimal hyperparameters for each dataset.
# while training, i received this convergence warning, and made it hard to view the results,
# so i am going to ignore those convergence warning
warnings.filterwarnings("ignore", category=ConvergenceWarning)

for dataset_name, (X, y) in datasets.items():
    print(f"Processing Dataset: {dataset_name} {'-' *40}\n")

    # Splitting the data into train and test sets
    X_train, X_test, y_train, y_test = split_data(X, y)
    print(f"Training size: {X_train.shape}, Testing size: {X_test.shape} \n")
    # for our cross validation, we only use train set.
    # Training and evaluating models
    # Setting iteration to 1000 for all models to ensure sufficient convergence
    # Model 1
    print(f"Training L2-LR model {'-' *40}")
    lr_model = LogisticRegression(penalty='l2', max_iter=1000)
    lr_mean_accuracies = cross_validation_1(lr_model, X_train, y_train, 'C', C_values)
    best_C_lr = C_values[np.argmax(lr_mean_accuracies)]
    print(f"Mean accuracies for Logistic Regression: {lr_mean_accuracies}")
    print(f"Optimal C for Logistic Regression: {best_C_lr}")

    # Model 2

    print(f"Training SVM with Linear Kernel {'-' *40}")
    svm_linear = SVC(kernel='linear', max_iter=1000)
    svm_linear_accuracies = cross_validation_1(svm_linear, X_train, y_train, 'C', C_values)
    best_svm_linear = C_values[np.argmax(svm_linear_accuracies)]
    print(f"Mean accuracies for SVM (Linear Kernel): {svm_linear_accuracies}")
    print(f"Optimal C for SVM (Linear Kernel): {best_svm_linear}")


  # Model 3
  print(f"Training SVM with RBF Kernel {'-' *40}")
  svm_rbf = SVC(kernel='rbf', max_iter=1000)
  svm_rbf_accuracies = cross_validation_1(svm_rbf, X_train, y_train, 'C', C_values)
  best_svm_rbf = C_values[np.argmax(svm_rbf_accuracies)]
  print(f"Mean accuracies for SVM (RBF Kernel): {svm_rbf_accuracies}")
  print(f"Optimal C for SVM (RBF Kernel): {best_svm_rbf}")

  #Model 4
  print(f"Training MLP model {'-' *40}")
  feature_count = X_train.shape[1]  # Extracting the number of features from the data
  mlp_neurons = hidden_neurons(feature_count, hidden_layer_sizes)
  print(f"Neurons being evaluated for {dataset_name}: {mlp_neurons}")
  mlp_model = MLPClassifier(max_iter=1000)
  mlp_mean_accuracies = cross_validation_1(mlp_model, X_train, y_train, 'hidden_layer_sizes', mlp_neurons)
  best_mlp_neurons = mlp_neurons[np.argmax(mlp_mean_accuracies)]
  print(f"Mean accuracies for MLP: {mlp_mean_accuracies}")
  print(f"Optimal neuron count for MLP: {best_mlp_neurons}")

  print(f" {'=' *80}\n")
```

```
Processing Dataset: Iris data ----------------------------------------

Training size: (90, 4), Testing size: (60, 4)

Training L2-LR model ----------------------------------------
Mean accuracies for Logistic Regression: [0.5222222222222223, 0.8222222222222223, 0.9222222222222223, 0.9444444444444444, 0.9666666666666668, 0.9666666
66666666, 0.9666666666666666]
Optimal C for Logistic Regression: 10
Training SVM with Linear Kernel ----------------------------------------
Mean accuracies for SVM (Linear Kernel): [0.41111111111111115, 0.8555555555555557, 0.9555555555555555, 0.9666666666666668, 0.9222222222222222, 0.9666666
666666666, 0.9666666666666666]
Optimal C for SVM (Linear Kernel): 1
Training SVM with RBF Kernel ----------------------------------------
Mean accuracies for SVM (RBF Kernel): [0.4222222222222222, 0.4222222222222222, 0.8666666666666668, 0.9555555555555555, 0.9666666666666666, 0.92222222222
22222, 0.9555555555555555]
Optimal C for SVM (RBF Kernel): 10
Training MLP model ----------------------------------------
Neurons being evaluated for Iris data: [1, 1, 2, 4, 8, 20, 40]
Mean accuracies for MLP: [0.45555555555555555, 0.4666666666666666, 0.6555555555555557, 0.7555555555555556, 0.8444444444444444, 0.9666666666666666, 0.955
5555555555555]
Optimal neuron count for MLP: 20
 ================================================================================


 Processing Dataset: Breast Cancer data ----------------------------------------

 Training size: (341, 30), Testing size: (228, 30)

 Training L2-LR model ----------------------------------------
 Mean accuracies for Logistic Regression: [0.9205882352941177, 0.9352941176470588, 0.9294117647058824, 0.9441176470588235, 0.95, 0.95, 0.952941176470588
 2]
 Optimal C for Logistic Regression: 1000
 Training SVM with Linear Kernel ----------------------------------------
 Mean accuracies for SVM (Linear Kernel): [0.9205882352941177, 0.8941176470588236, 0.7147058823529412, 0.6588235294117647, 0.6588235294117647, 0.65882352
 94117647, 0.6588235294117647]
 Optimal C for SVM (Linear Kernel): 0.001
 Training SVM with RBF Kernel ----------------------------------------
 Mean accuracies for SVM (RBF Kernel): [0.611764705882353, 0.611764705882353, 0.8647058823529411, 0.8823529411764705, 0.8911764705882353, 0.9, 0.92647058
 82352942]
 Optimal C for SVM (RBF Kernel): 1000
 Training MLP model ----------------------------------------
 Neurons being evaluated for Breast Cancer data: [3, 6, 15, 30, 60, 150, 300]
 Mean accuracies for MLP: [0.6029411764705882, 0.4558823529411646, 0.6882352941176471, 0.7823529411764707, 0.8823529411764707, 0.9117647058823529, 0.911
 7647058823529]
 Optimal neuron count for MLP: 150
  ================================================================================
```

Now I will define cross validation 2, with a minute change in the indexing for X_train_fold, as the data in news group is sparse data, I will use vstack for adding the data [other than validation set]

```python
def cross_validation_2(model, X_train, y_train, hyperparameter_name, hyperparameter_values):

    # Handling sparse matrices using the shape property
    num_samples = X_train.shape[0]  # Counting the number of samples
    fold_size = num_samples // 10  # Determining the size of each fold
    mean_accuracies = []  # Initializing a list to store mean accuracies for each hyperparameter value

    for param_value in hyperparameter_values:
        fold_accuracies = []  # Initializing a list to store accuracies for the current hyperparameter value
        for fold in range(10):
            start_index, end_index = fold * fold_size, (fold + 1) * fold_size
            # Extracting the validation set for the current fold
            X_val, y_val = X_train[start_index:end_index], y_train[start_index:end_index]
            # Combining the remaining data to form the training set
            X_train_fold = X_train_fold = vstack([X_train[:start_index], X_train[end_index:]]) # changes made only here
            y_train_fold = np.concatenate((y_train[:start_index], y_train[end_index:]), axis=0)

            # Setting the model's hyperparameter and training
            setattr(model, hyperparameter_name, param_value)
            model.fit(X_train_fold, y_train_fold)

            # Validating the model and calculating accuracy
            y_pred = model.predict(X_val)
            fold_accuracies.append(accuracy_score(y_val, y_pred))

        # Calculating mean accuracy for the current hyperparameter value
        mean_accuracies.append(np.mean(fold_accuracies))

    return mean_accuracies
```

```
Due to large dimension of data in news group, i am reducing the iterations to 25, as it was taking a lot of time when i set the iteration to higher
number.
```

```python
X = news_group_dataset.data
y = news_group_dataset.target
dataset_name = "Newsgroup data"
print(f"Processing Dataset: {dataset_name} {'-' *40}\n")

# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = split_data(X, y)
print(f"Training size: {X_train.shape}, Testing size: {X_test.shape} \n")
# for our cross validation, we only use train set.
# Training and evaluating models
# Setting iteration to 1000 for all models to ensure sufficient convergence
# Model 1
print(f"Training L2-LR model {'-' *40}")
lr_model = LogisticRegression(penalty='l2', max_iter=25)
lr_mean_accuracies = cross_validation_2(lr_model, X_train, y_train, 'C', C_values)
best_C_lr = C_values[np.argmax(lr_mean_accuracies)]
print(f"Mean accuracies for Logistic Regression: {lr_mean_accuracies}")
print(f"Optimal C for Logistic Regression: {best_C_lr}")
print(f" {'=' *80}\n")
# Model 2

print(f"Training SVM with Linear Kernel {'-' *40}")
svm_linear = SVC(kernel='linear', max_iter=25)
svm_linear_accuracies = cross_validation_2(svm_linear, X_train, y_train, 'C', C_values)
best_svm_linear = C_values[np.argmax(svm_linear_accuracies)]
print(f"Mean accuracies for SVM (Linear Kernel): {svm_linear_accuracies}")
print(f"Optimal C for SVM (Linear Kernel): {best_svm_linear}")
print(f" {'=' *80}\n")
# Model 3
print(f"Training SVM with RBF Kernel {'-' *40}")
svm_rbf = SVC(kernel='rbf', max_iter=25)
svm_rbf_accuracies = cross_validation_2(svm_rbf, X_train, y_train, 'C', C_values)
best_svm_rbf = C_values[np.argmax(svm_rbf_accuracies)]
print(f"Mean accuracies for SVM (RBF Kernel): {svm_rbf_accuracies}")
print(f"Optimal C for SVM (RBF Kernel): {best_svm_rbf}")

print(f" {'=' *80}\n")
```

```
Processing Dataset: Newsgroup data ----------------------------------------

Training size: (11307, 130107), Testing size: (7539, 130107)

Training L2-LR model ----------------------------------------
Mean accuracies for Logistic Regression: [0.10176991150442478, 0.2813274336283186, 0.5336283185840708, 0.7275221238938052, 0.7479646017699115, 0.7405309
734513275, 0.7406194690265486]
Optimal C for Logistic Regression: 10
 ================================================================================

Training SVM with Linear Kernel ----------------------------------------
Mean accuracies for SVM (Linear Kernel): [0.16389380530973452, 0.16398230088495577, 0.16168141592920354, 0.29230088495575224, 0.6204424778761062, 0.6232
743362831858, 0.6215044247787611]
Optimal C for SVM (Linear Kernel): 100
 ================================================================================

Training SVM with RBF Kernel ----------------------------------------
Mean accuracies for SVM (RBF Kernel): [0.23911504424778762, 0.23858407079646016, 0.2497345132743362, 0.5000884955752213, 0.5863716814159292, 0.584070796
460177, 0.5836283185840707]
Optimal C for SVM (RBF Kernel): 10
 ================================================================================
```

While training on news group data, I realized due to large dimension of its data, in the case of MLP, i am facing memory error issue like this "Unable to allocate 12.6 GiB for an array with shape (130107, 13010) and data type float64". So in order to handle this issue, I am using SVD as a dimensional reduction technique to reduce the dimension from 130107 to 200, this will prevent the issue of memory error.

```
X = news_group_dataset.data
y = news_group_dataset.target
X_train, X_test, y_train, y_test = split_data(X, y)
print(f'shape of data matrix {X_train.shape}')

svd = TruncatedSVD(n_components=200)
X_train = svd.fit_transform(X_train)
print(f'The shape of Dataset post SVD embedding is: {X_train.shape}')
```

```
shape of data matrix (11307, 130107)
The shape of Dataset post SVD embedding is: (11307, 200)
```

```
#Model 4
print(f"Training MLP model {'-' *40}")
feature_count = X_train.shape[1]  # Extracting the number of features from the data
mlp_neurons = hidden_neurons(feature_count, hidden_layer_sizes)
print(f"Neurons being evaluated for {dataset_name}: {mlp_neurons}")
mlp_model = MLPClassifier(max_iter=25)
mlp_mean_accuracies = cross_validation_2(mlp_model, X_train, y_train, 'hidden_layer_sizes', mlp_neurons)
best_mlp_neurons = mlp_neurons[np.argmax(mlp_mean_accuracies)]
print(f"Mean accuracies for MLP: {mlp_mean_accuracies}")
print(f"Optimal neuron count for MLP: {best_mlp_neurons}")
```

```
Training MLP model ----------------------------------------
Neurons being evaluated for Newsgroup data: [20, 40, 100, 200, 400, 1000, 2000]
Mean accuracies for MLP: [0.6727433628318583, 0.7088495575221239, 0.7317699115044248, 0.7385840707964603, 0.7408849557522124, 0.7447787610619468, 0.7453
982300884956]
Optimal neuron count for MLP: 2000
```

	Iris Data			Breast Cancer Data			20 News Group Data		
C	**L2-LR**	**SVM[LK]**	**SVM[RBF]**	**L2-LR**	**SVM[LK]**	**SVM[RBF]**	**L2-LR**	**SVM[LK]**	**SVM[RBF]**
0.001	0.522	0.411	0.422	0.92	**0.92**	0.611	0.101	0.163	0.239
0.01	0.822	0.855	0.422	0.935	0.894	0.611	0.281	0.163	0.238
0.1	0.922	0.955	0.866	0.929	0.714	0.864	0.533	0.161	0.249
1	0.944	**0.966**	0.955	0.944	0.658	0.882	0.727	0.292	0.5
10	**0.966**	0.922	**0.966**	0.95	0.658	0.891	**0.747**	0.620	**0.586**
100	0.966	0.966	0.922	0.95	0.658	0.9	0.74	**0.623**	0.584
1000	0.966	0.966	0.955	**0.952**	0.658	**0.926**	0.74	0.621	0.583

Iris Data							
Hidden Neurons	1	1	2	4	8	20	40
Mean Accuracy	0.455	0.466	0.655	0.755	0.844	**0.966**	0.955
Breast Cancer Data							
Hidden Neurons	3	6	15	30	60	150	300
Mean Accuracy	0.602	0.455	0.688	0.782	0.882	**0.911**	0.911
20 News Group Data							
Hidden Neurons	20	40	100	200	400	1000	2000
Mean Accuracy	0.672	0.708	0.731	0.739	0.740	0.744	**0.745**

In the case of 20 news group data, the neurons are calculated based on reduced dimension of 200.

I have highlighted the optimal parameters for each model in their corresponding dataset.

Coming to best and worst classifiers:

1. Iris Data:
   Best: All 4 models have given the same highest accuracy of 0.966.
2. Breast cancer data:
   Best model is from L2-LR with highest accuracy of 0.952 while worst model will be MLP with its highest accuracy of 0.911
3. News group data:
   Best model is from L2-LR with highest accuracy of 0.747 while worst model will be SVM[RBF] with its highest accuracy of 0.586

Problem 5 (10 Points): Reconstructing Europe via MDS

```
#Libraries Required:
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Loading the distance matrix for 24 European cities from the provided file
data_path = "D:/cities.csv"
# Specifying no header in the CSV file since it doesn't include column names
distance_matrix_df = pd.read_csv(data_path, delimiter=';', index_col=0, header=None)
# Displaying the first five rows of the distance matrix
print("Previewing the initial rows of the distance matrix:")
print(distance_matrix_df.head())
```

```
Previewing the initial rows of the distance matrix:
                1        2        3        4        5        6        7   \
0
Barcelona     0.00  1528.13  1497.61  1062.89  1968.42  1498.79  1757.54
Belgrade   1528.13     0.00   999.25  1372.59   447.34   316.41  1327.24
Berlin     1497.61   999.25     0.00   651.62  1293.40   689.06   354.03
Brussels   1062.89  1372.59   651.62     0.00  1769.69  1131.52   766.67
Bucharest  1968.42   447.34  1293.40  1769.69     0.00   639.77  1571.54

                8        9        10  ...       16       17       18       19  \
0                                     ...
Barcelona  1469.29  1471.78  2230.42  ...  1054.55   831.59  1353.90   856.69
Belgrade   2145.39  1229.93   809.48  ...   773.33  1445.70   738.10   721.55
Berlin     1315.16   254.51  1735.01  ...   501.97   876.96   280.34  1181.67
Brussels    773.20   489.76  2178.85  ...   601.87   261.29   721.08  1171.34
Bucharest  2534.72  1544.17   445.62  ...  1186.37  1869.95  1076.82  1137.38

                20       21       22       23       24  25
0
Barcelona  2813.02  1745.55  2276.51  1347.43  1862.33 NaN
Belgrade   1797.75   329.46  1620.96   489.28   826.66 NaN
Berlin     1319.62  1318.67   810.38   523.61   516.06 NaN
Brussels   1903.66  1697.83  1280.88   914.81  1159.85 NaN
Bucharest  1740.39   296.68  1742.25   855.32   946.12 NaN

[5 rows x 25 columns]
```

We can observe that the last column contains NaN values, likely due to an extra delimiter (;) at the end of each row in the CSV file. Removing the last column by slicing with indexing.

```
distance_matrix_df = distance_matrix_df.iloc[:, :24]
#i will preview it once again now
print(distance_matrix_df.head())
```

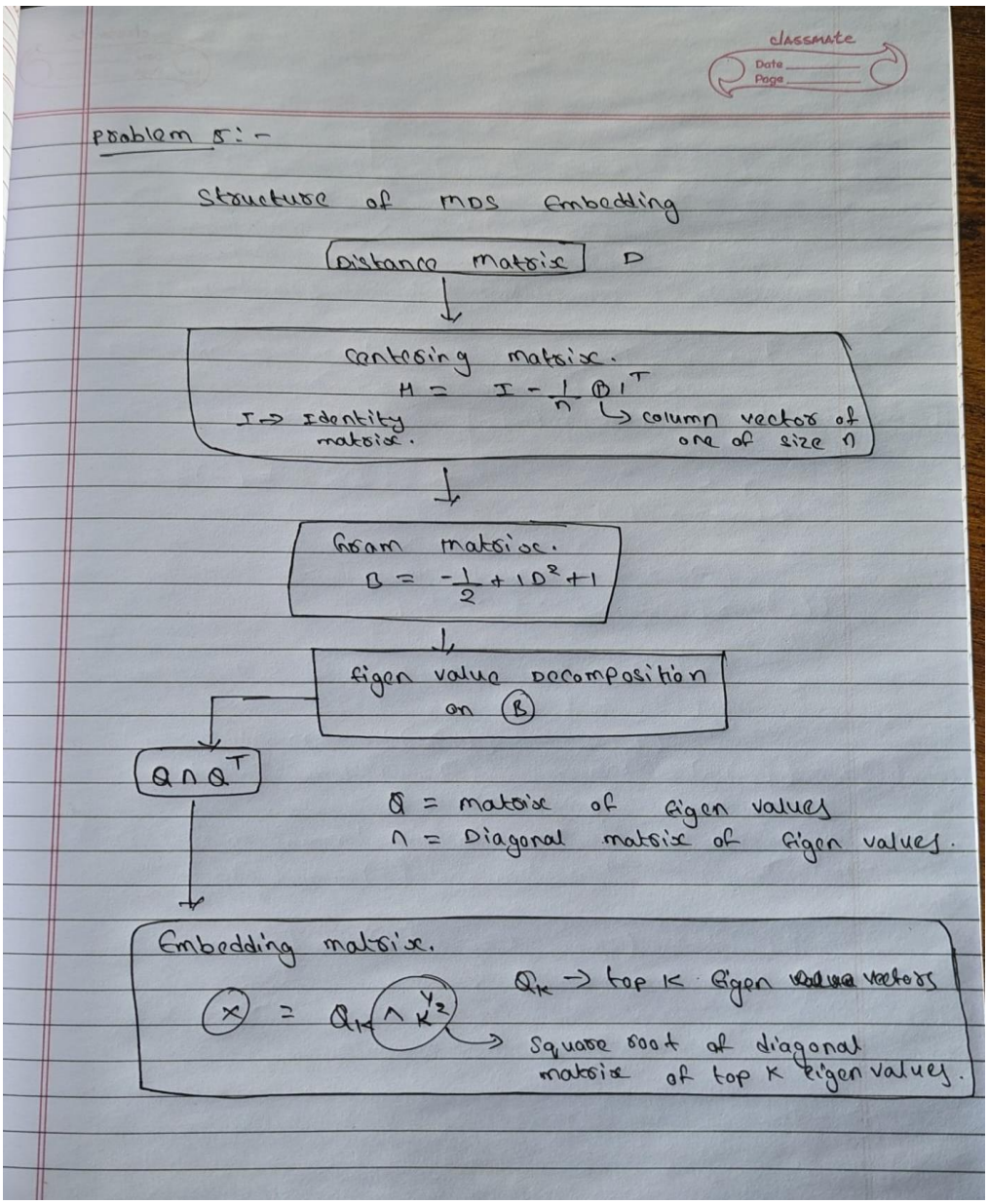
```
                1        2        3        4        5        6        7   \
0
Barcelona     0.00  1528.13  1497.61  1062.89  1968.42  1498.79  1757.54
Belgrade   1528.13     0.00   999.25  1372.59   447.34   316.41  1327.24
Berlin     1497.61   999.25     0.00   651.62  1293.40   689.06   354.03
Brussels   1062.89  1372.59   651.62     0.00  1769.69  1131.52   766.67
Bucharest  1968.42   447.34  1293.40  1769.69     0.00   639.77  1571.54

                8        9        10  ...       15       16       17       18  \
0                                     ...
Barcelona  1469.29  1471.78  2230.42  ...  3006.93  1054.55   831.59  1353.90
Belgrade   2145.39  1229.93   809.48  ...  1710.99   773.33  1445.70   738.10
Berlin     1315.16   254.51  1735.01  ...  1607.99   501.97   876.96   280.34
Brussels    773.20   489.76  2178.85  ...  2253.26   601.87   261.29   721.08
Bucharest  2534.72  1544.17   445.62  ...  1497.56  1186.37  1869.95  1076.82

                19       20       21       22       23       24
0
Barcelona   856.69  2813.02  1745.55  2276.51  1347.43  1862.33
Belgrade    721.55  1797.75   329.46  1620.96   489.28   826.66
Berlin     1181.67  1319.62  1318.67   810.38   523.61   516.06
Brussels   1171.34  1903.66  1697.83  1280.88   914.81  1159.85
Bucharest  1137.38  1740.39   296.68  1742.25   855.32   946.12

[5 rows x 24 columns]
```

Now the task is to perform MDS embedding on the distance matrix, reduce its dimension into 2d plane, then using the top 2 eigen values [2 because we want to transform our data from 24 dimensions 2] and its corresponding eigen vectors, use them as coordinates and plot them on a 2d plane and observe for any rotation or mirror effect and nullify them by reverse transformation and compare it with original map of Europe.

Given a Distance matrix, the structure to find MDS embeddings of k dimensions is:

Problem 5:-

Structure of MDS Embedding



```python
# converting the given data frame into numpy array to apply matrix operations for MDS embedding
distance_matrix = distance_matrix_df.to_numpy()
n = 24 # Given that there are 24 cities.
# In reference to the attached image, I will first find H, then D_squared, B and eigen value Decomposit

# H matrix:
H = np.eye(n) - (1 / n) * np.ones((n, n))
print(f'the first 5 rows of centering matrix H is \n: {H[:5, :5]}\n')
# D square:
D_squared = distance_matrix ** 2
# B = -0.5 * H * D_squared * H
# Note: For matrix multiplication we use @ for product.
B = -0.5 * H @ D_squared @ H
print(f'The first 5 rows B matrix is: \n {B[:5, :5]}')

# Eigen value decomposition on B
eigenvalues, eigenvectors = np.linalg.eigh(B)
```

```
the first 5 rows of centering matrix H is
: [[ 0.95833333 -0.04166667 -0.04166667 -0.04166667 -0.04166667]
 [-0.04166667  0.95833333 -0.04166667 -0.04166667 -0.04166667]
 [-0.04166667 -0.04166667  0.95833333 -0.04166667 -0.04166667]
 [-0.04166667 -0.04166667 -0.04166667  0.95833333 -0.04166667]
 [-0.04166667 -0.04166667 -0.04166667 -0.04166667  0.95833333]]

The first 5 rows B matrix is:
 [[1765403.62876493  -38198.84285799 -185924.4625309   587833.62295868
  -503486.79533924]
 [ -38198.84285799  493379.9824191  -199768.71090382 -425312.27821424
   697783.49188785]
 [-185924.4625309  -199768.71090382  105583.15827326  110486.65156285
  -232500.16238507]
 [ 587833.62295868 -425312.27821424  110486.65156285  539998.76925243
  -744751.92494549]
 [-503486.79533924  697783.49188785 -232500.16238507 -744751.92494549
  1102300.0769566 ]]
```

Eigen values explain the variance in the data, so the greater the magnitude of eigen values the more they carry the information of original data. # So, i will sort eigen values in Descending order.

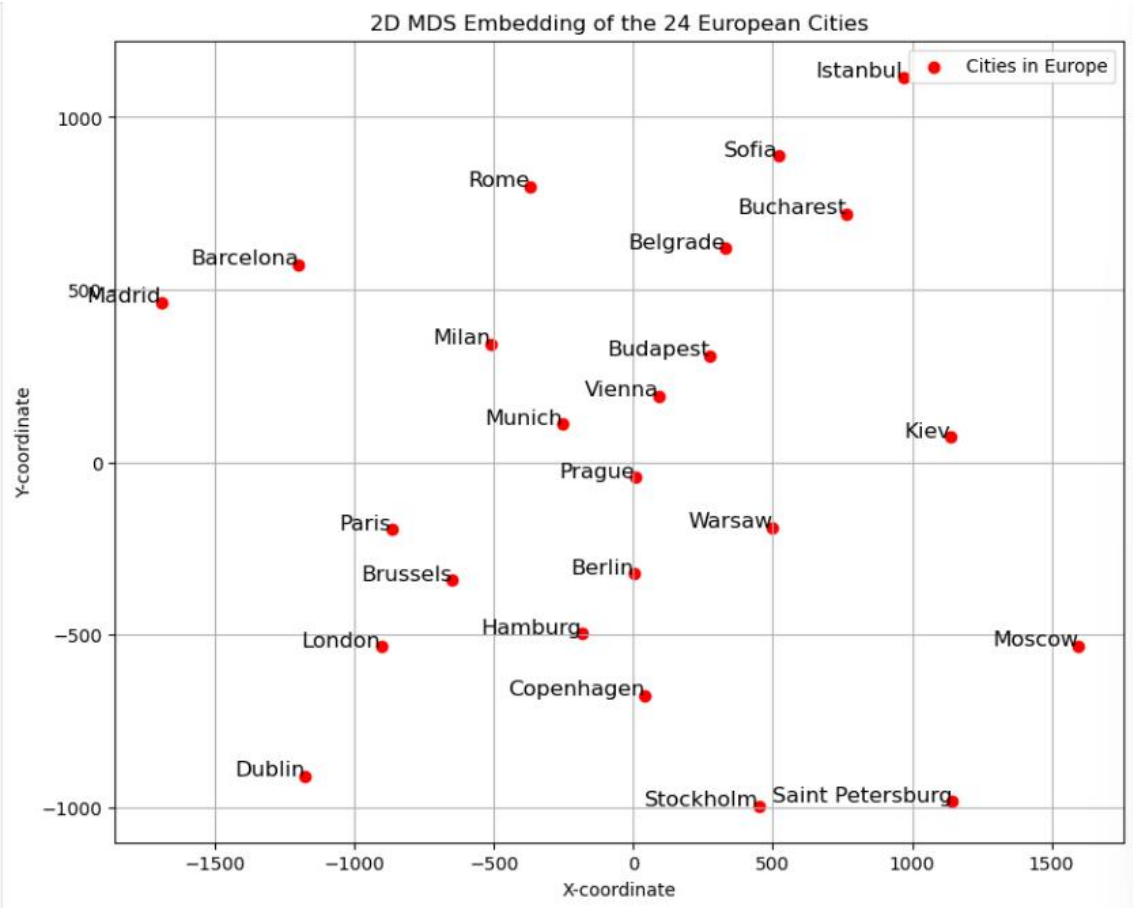
```
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvalues = eigenvalues[sorted_indices]
print(f"The eigen values in descending order are: {sorted_eigenvalues}")
# Now i will sort the eigen vectors based on the magintude of its corresponding eigen value
sorted_eigenvectors = eigenvectors[:, sorted_indices]

# Reducing the dimensionality to 2D using the top 2 eigenvalues and their corresponding eigenvectors
top_2_eigenvalues = sorted_eigenvalues[:2]
print(f"The top 2 eigenvalues are: {top_2_eigenvalues}")
top_2_eigenvectors = sorted_eigenvectors[:, :2]
print(f"The top 2 eigenvectors corresponding to the eigenvalues are: {top_2_eigenvectors}")
```

```
The eigen values in descending order are: [ 1.57872177e+07  8.73373090e+06  3.07511197e+04  2.67933915
e+02
  5.70274942e+01  1.87076579e+01  1.79028050e+01  1.19559854e+01
  8.68503209e+00  7.20711204e+00  3.85383093e+00  4.86023761e-01
 -9.49718183e-11 -2.08401635e+00 -3.11467185e+00 -5.03729407e+00
 -9.29167346e+00 -1.16149994e+01 -1.30884807e+01 -1.62803611e+01
 -2.30756445e+01 -7.14754096e+01 -1.65297107e+04 -5.71303569e+04]
The top 2 eigenvalues are: [15787217.7325849   8733730.90463019]
The top 2 eigenvectors corresponding to the eigenvalues are: [[-3.02311560e-01  1.93794481e-01]
 [ 8.26032718e-02  2.10131043e-01]
 [ 1.06841695e-04 -1.09212310e-01]
 [-1.63841513e-01 -1.15417937e-01]
 [ 1.92576868e-01  2.43622682e-01]
 [ 6.92284937e-02  1.04645660e-01]
 [ 1.00805168e-02 -2.28288795e-01]
 [-2.96449848e-01 -3.07405885e-01]
 [-4.68473327e-02 -1.67901128e-01]
 [ 2.43693196e-01  3.77587791e-01]
 [ 2.86596025e-01  2.46896261e-02]
 [-2.27915375e-01 -1.80390423e-01]
 [-4.26450649e-01  1.56663796e-01]
 [-1.29152311e-01  1.16210393e-01]
 [ 4.01717402e-01 -1.80018863e-01]
 [-6.36396383e-02  3.75032868e-02]
 [-2.18168297e-01 -6.53451826e-02]
 [ 1.33356497e-03 -1.43388641e-02]
 [-9.38597803e-02  2.70379173e-01]
 [ 2.87636646e-01 -3.32444724e-01]
 [ 1.30707344e-01  3.00770359e-01]
 [ 1.13624960e-01 -3.36860419e-01]
 [ 2.31430196e-02  6.52676641e-02]
 [ 1.25588155e-01 -6.36414250e-02]]
```

Using the first two eigenvalues to represent the highest variance (or information) in the data,

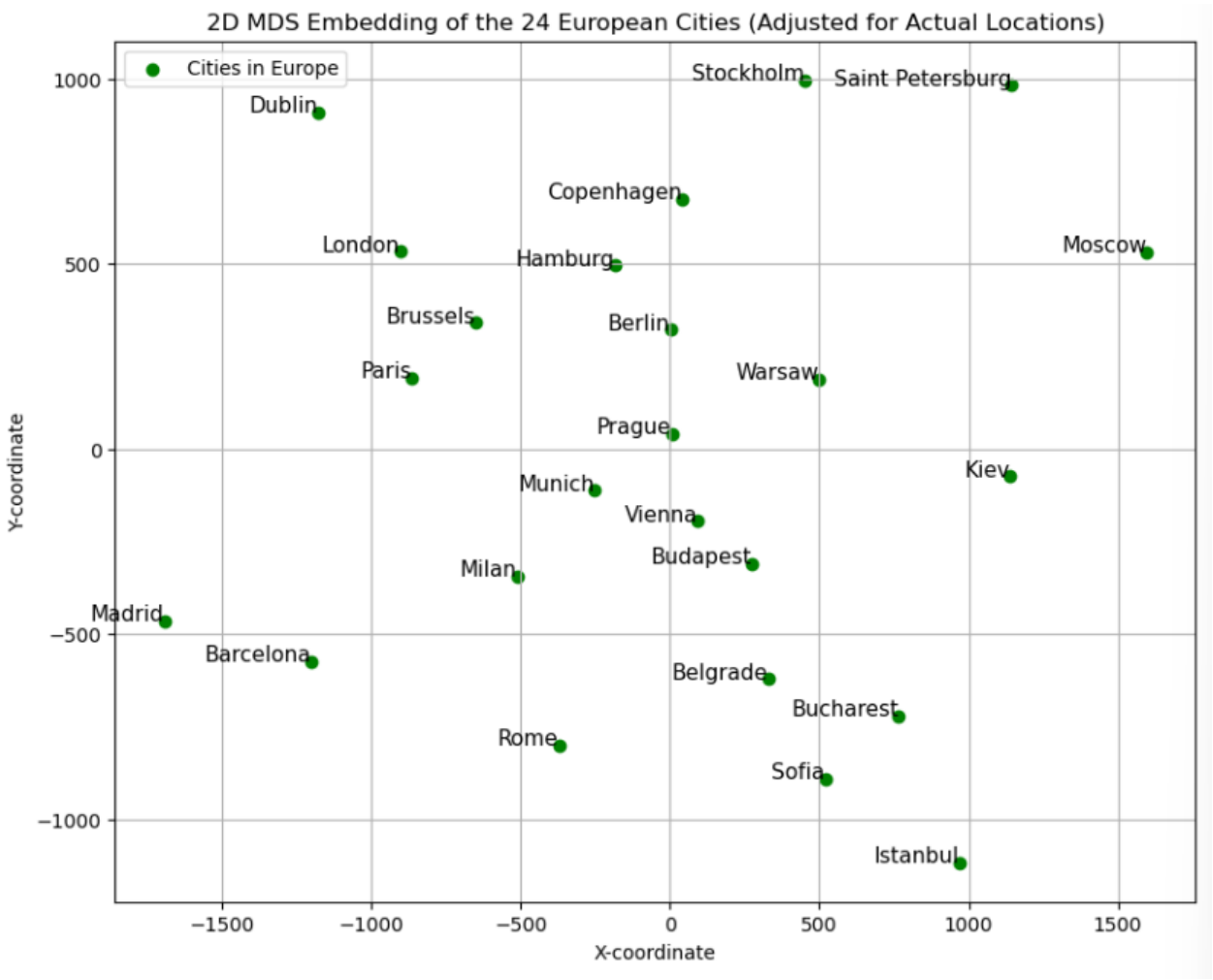
The corresponding eigenvectors indicate the directions of these dimensions in the transformed space. Creating a 2D embedding using MDS:

```
mds_coordinates = top_2_eigenvectors * np.sqrt(top_2_eigenvalues)
# Visualizing the MDS embedding
plt.figure(figsize=(10, 8))
plt.scatter(mds_coordinates[:, 0], mds_coordinates[:, 1], color='red', label="Cities in Europe")
# Annotating each city on the plot
for idx, city_name in enumerate(distance_matrix_df.index):
    plt.annotate(city_name, (mds_coordinates[idx, 0], mds_coordinates[idx, 1]), fontsize=12, ha='right'
plt.title("2D MDS Embedding of the 24 European Cities")
plt.xlabel("X-coordinate")
plt.ylabel("Y-coordinate")
plt.legend()
plt.grid()
plt.show()
```



When compared to the actual map of Europe I realized the above plot is reflection over X-axis[Horizontal Plane]. So all change the sign of y-coordinates so that the cities above X-axis which are having positive y value will now change to negative y-value which brings them back to actual position and vice versa

```python
# Changing the sign of y-coordinates to negate the effect of reflection over x-axis
mds_coordinates[:, 1] = -mds_coordinates[:, 1]

# Plotting the adjusted MDS embedding
plt.figure(figsize=(10, 8))
plt.scatter(mds_coordinates[:, 0], mds_coordinates[:, 1], color='green', label="Cities in Europe")

# Annotating each city on the adjusted plot
for idx, city_name in enumerate(distance_matrix_df.index):
    plt.annotate(city_name, (mds_coordinates[idx, 0], mds_coordinates[idx, 1]), fontsize=11, ha='right'

plt.title("2D MDS Embedding of the 24 European Cities (Adjusted for Actual Locations)")
plt.xlabel("X-coordinate")
plt.ylabel("Y-coordinate")
plt.legend()
plt.grid()
plt.show()
```



The actual map: Source [map]

When comparing the positions of cities in actual map and in the 2D embeddings map, we see they almost have same positions. This shows how effective MDS is in reducing the dimensions, at same time keeping most of the information explained by the data.

Problem 6 (20 Points): The (Random) Walking Dead

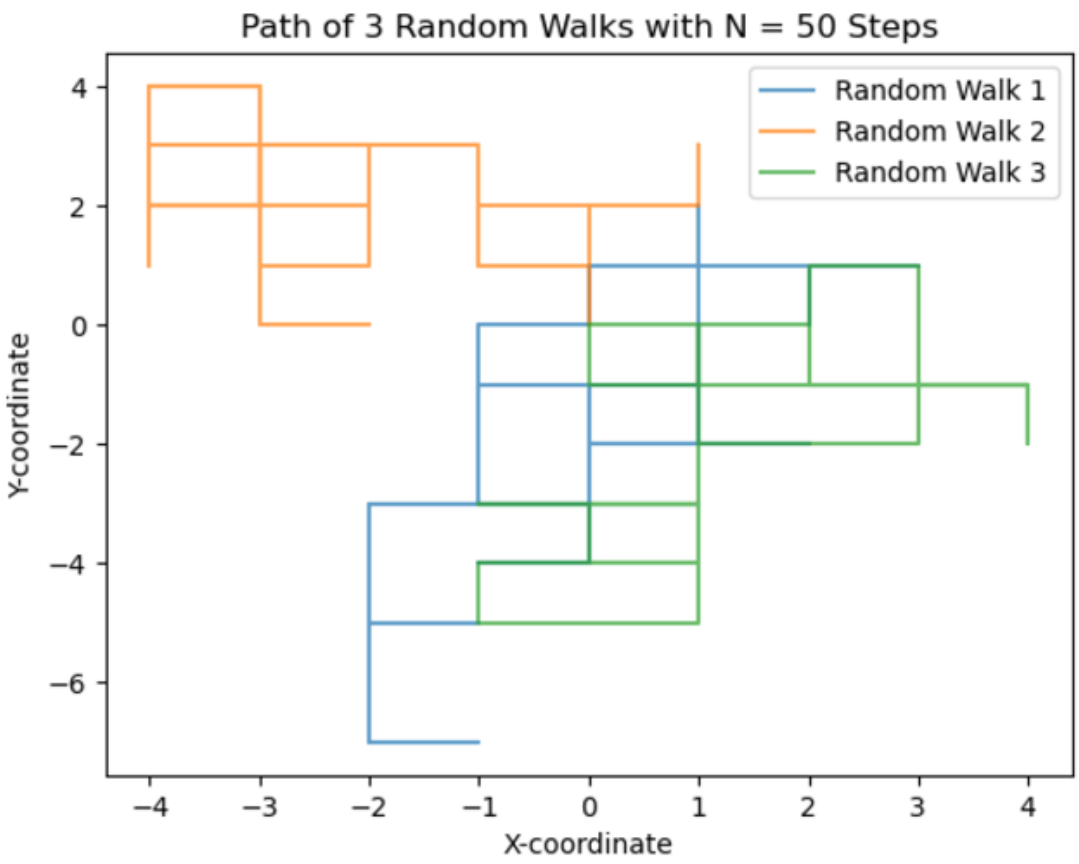
```
# Libraries required
import math
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

Part 1:

To plot the trace of 3 independent realizations of such random walk.

My approach:- Start with given initial coordinates --> random choice in 4 options [up, down, right, left] with step size of 1 --> update the coordinates based on the random selection. --> Repeat from random choice step until you reach the step count (N) = 50. Do this for 3 different times and plot each random walk.

```
def random_walk(steps_count):
    #initial position
    x, y = 0, 0 # Random walk starts from origin.
    # to plot the trace of the random walk, I define two lists for x,y coordinates
    path_x = [x]
    path_y = [y]

    for i in range(steps_count):
        direction = np.random.choice(['up', 'down', 'left', 'right'])
        if direction == 'up':
            y += 1
        elif direction == 'down':
            y -= 1
        elif direction == 'right':
            x += 1
        else:
            x -= 1
        path_x.append(x) # adding the x coordinates of random walk
        path_y.append(y) # adding the y coordinates of random walk
    return path_x, path_y
# given steps count [N] = 50
N = 50
# in order to observe the overlapping paths, i plot the lines with transparency [alpha=0.7]
# so that in case of overlapping trace, the line will be more darker with merge of colors
for i in range(3):
    x, y = random_walk(N)
    plt.plot(x, y, label=f'Random Walk {i+1}', alpha=0.7)
plt.title('Path of 3 Random Walks with N = 50 Steps')
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.legend()
plt.show()
```



Part 2

To find the expected value of d = sqrt(x^2+y^2) where x, y are the final coordinates of random walk after 50 steps and calculate the final distance for 10,000 random walks. THen measure the average of 10,000 final distances.

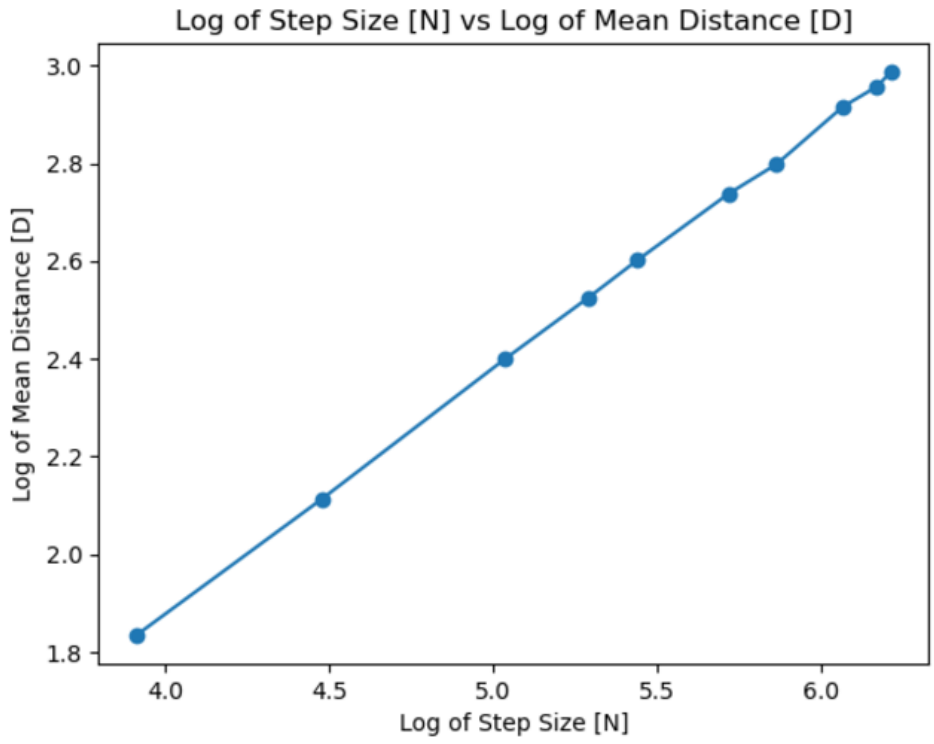
```
def avg_final_dist(steps, num_simulations):
    final_dist = [] # initiating a list to store the final distances of each random walk
    for i in range(num_simulations):
        path_x, path_y = random_walk(steps)
        final_x, final_y = path_x[-1], path_y[-1]
        D = math.sqrt(final_x ** 2 + final_y ** 2)
        final_dist.append(D)
    return np.mean(final_dist)

Avg_dist = avg_final_dist(50, 10000)
print(f'The mean distance after 50 steps over 10,000 random walks is: {Avg_dist}')
```

```
The mean distance after 50 steps over 10,000 random walks is: 6.2567795754137405
```

Part 3

To show d scales as sqrt(N). where N varies from 10 to 500. From the hint, I will plot the log N vs Log D and fit a regression model to find the slope of the question: log(D) = Beta1 * log(N) + Beta0, where Beta1 = slope. when d = N ^1/2 --> log(d) = 1/2 *log(N) where 1/2 is the slope. So we will find the slope and check if it is equal to 0.5 or not

```
# Defining different values for step counts ranging from 10 to 500
step_sizes = [50, 88, 154, 198, 230, 304, 352, 431, 478, 499] # N varying from 10 to 500

mean_distances = []  # Empty list to store the mean distances from multiple random walks

# Looping through each step size and compute the mean distance for 10,000 random walks
for steps in step_sizes:
    mean_dist = avg_final_dist(steps, 10000)
    mean_distances.append(mean_dist)

# Plotting the logarithmic relationship between step size and mean distance
plt.plot(np.log(step_sizes), np.log(mean_distances), marker='o')
plt.title('Log of Step Size [N] vs Log of Mean Distance [D]')
plt.xlabel('Log of Step Size [N]')
plt.ylabel('Log of Mean Distance [D]')
plt.show()
```



when d = N ^1/2 --> log(d) = 1/2 *log(N) where 1/2 is the slope. So we will find the slope and check if it is equal to 0.5 or not

In order find the slope of the above plot by fitting a linear regression and equating the beta 1 to slope.

```
log_steps = np.log(step_sizes)
log_distance = np.log(mean_distances)
regression_model = LinearRegression()
regression_model.fit(log_steps, log_distance)
```

```
ValueError: Expected 2D array, got 1D array instead:
array=[3.91202301 4.47733681 5.0369526  5.28826703 5.43807931 5.7170277
 5.86363118 6.06610809 6.16961073 6.2126061 ].
Reshape your data either using array.reshape(-1, 1) if your data has a single feature or array.reshape
(1, -1) if it contains a single sample.
```

As the linear regression is expecting a 2D array, i will reshape log_steps using log_steps.reshape(-1,1)

```
log_steps = log_steps.reshape(-1, 1)
regression_model = LinearRegression()
regression_model.fit(log_steps, log_distance)
Beta1 = regression_model.coef_[0]
print(f'The slope of log N vs Log D is {Beta1}')
```

```
The slope of log N vs Log D is 0.49969013743343804
```

Slope = 0.4996 approximately equal to 0.5 Therefore we proved D scales as sqrt(N)