In [285…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits.mplot3d import Axes3D
```

In [ ]:

1. KNN Imputation and Classification

In [ ]:

a.

In [92]:

```
df1 = pd.read_csv('DodgerLoopGame\DodgerLoopGame_TRAIN.txt',sep="\s+",header=None)
df2 = pd.read_csv('DodgerLoopGame\DodgerLoopGame_TEST.txt',sep="\s+",header=None)
df2.head()
```

Out[92]:

	0	1	2	3	4	5	6	7	8	9	...	279	280	281	282	283	284	285	286	287	288
0	1.0	2.0	8.0	7.0	6.0	2.0	8.0	5.0	10.0	3.0	...	11.0	7.0	12.0	4.0	9.0	13.0	3.0	5.0	5.0	3.0
1	1.0	8.0	5.0	10.0	11.0	9.0	10.0	7.0	18.0	11.0	...	13.0	7.0	6.0	7.0	1.0	2.0	6.0	7.0	8.0	6.0
2	1.0	1.0	9.0	5.0	2.0	5.0	5.0	5.0	13.0	6.0	...	10.0	5.0	14.0	9.0	13.0	10.0	9.0	5.0	11.0	6.0
3	1.0	2.0	6.0	6.0	3.0	8.0	6.0	3.0	4.0	3.0	...	8.0	13.0	9.0	11.0	10.0	4.0	6.0	6.0	3.0	10.0
4	1.0	8.0	6.0	5.0	11.0	7.0	4.0	10.0	8.0	8.0	...	19.0	13.0	15.0	17.0	10.0	8.0	12.0	16.0	15.0	13.0

5 rows × 289 columns

In [96]:

```
NaTrain = df1.isna().sum().sum()
NaTest = df2.isna().sum().sum()
print('Numberof NaN values in Train Datasets are',NaTrain)
print('Numberof NaN values in Test Datasets are',NaTest)
```

```
Numberof NaN values in Train Datasets are 65
Numberof NaN values in Test Datasets are 272
```

In [ ]:

In [ ]:

b. KNN Imputer

In [98]:

```
from sklearn.impute import KNNImputer
from scipy.spatial.distance import cdist

impute = KNNImputer(n_neighbors=3)

# Train Dataset
df1_imput = impute.fit_transform(df1)
df1_imputed = pd.DataFrame(df1_imput, columns=df1.columns)

# Test Dataset
df2_imput = impute.fit_transform(df2)
df2_imputed = pd.DataFrame(df2_imput, columns=df2.columns)

print("\nTrain Dataset Number of NaN values after Imputation:", df1_imputed.isna().sum().sum())
print("\nTest Dataset Number of NaN values after Imputation:", df2_imputed.isna().sum().sum())

#mse = mean_squared_error(df2_imput, df2_imputed)
#print(mse)
```

```
Train Dataset Number of NaN values after Imputation: 0

Test Dataset Number of NaN values after Imputation: 0
```

In [146…

```
from sklearn.impute import KNNImputer
from scipy.spatial.distance import cdist

def knn_imputer_grid_search(X_train, X_test, k_values):
    best_k = 1
    best_mean_distance = float('inf')

    for k in k_values:
```

```python
        imputer = KNNImputer(n_neighbors=k)
        train_imputed = imputer.fit_transform(X_train)
        test_imputed = imputer.transform(X_test)
        # Checking shapes to see if it aligns or not ..
      #  print("Shape of imputed values after flattening:", np.isnan(imputed_values).sum())

       # distances = cdist(original_values_without_nan.reshape(-1, 1), imputed_values.reshape(-1, 1), metric='eu
        distances = cdist(train_imputed, test_imputed, metric='euclidean') # calculating the  pairwise Euclidea
        mean_distance = np.mean(distances) # Getting the mean distances

        print(f"K={k}, Mean Distance={mean_distance:.4f}")

        # Update the best K if current mean distance is lower
        if mean_distance < best_mean_distance:
            best_mean_distance = mean_distance
            best_k = k

    print(f"Optimal number of neighbors (K): {best_k}")    # Using the best K to impute both train and test data
    print(f"Best Mean Distance: {best_mean_distance:.4f}")

    # Final imputation with the best K
    final_imputer = KNNImputer(n_neighbors=best_k)
    # Using the best k for the input data
    train_imputed = pd.DataFrame(final_imputer.fit_transform(X_train), columns=X_train.columns)
    # test_imputed = pd.DataFrame(final_imputer.transform(X_test), columns=X_test.columns  # To perform imputat

    return best_k

k_values = [1, 3, 5, 7, 9]
optimal_k = knn_imputer_grid_search(df1, df2,k_values)
print(f"Optimal K: {optimal_k}")
```

```
K=1, Mean Distance=179.2819
K=3, Mean Distance=178.9790
K=5, Mean Distance=178.8261
K=7, Mean Distance=178.8077
K=9, Mean Distance=178.7810
Optimal number of neighbors (K): 9
Best Mean Distance: 178.7810
Optimal K: 9
```

In [176…

```python
# # def knn_imputer_grid_search(X_train, k_values):
# #     best_k = 1
# #     best_mean_distance = float('inf')

# #     missing_mask = ~X_train.isna()

# #     for k in k_values:

# #         imputer = KNNImputer(n_neighbors=k)


# #         train_imputed = imputer.fit_transform(X_train)

# #         # Getting the original and imputed values where the original values are missing
# #         original_values = X_train[missing_mask].values.ravel()
# #         imputed_values = train_imputed[missing_mask].ravel()

# #         original_values_without_nan = original_values[~np.isnan(original_values)]
# #         print("Shape of original values after flattening:", original_values_without_nan.shape)
# #         print("Shape of imputed values after flattening:", imputed_values.shape)


# #         # Checking shapes to see if it aligns or not ..
# #       #  print("Shape of original values after dropping 65:", np.isnan(original_values_dropped).sum())
# #       #  print("Shape of imputed values after flattening:", np.isnan(imputed_values).sum())

# #         # Compute pairwise Euclidean distance between original and imputed values
# #         distances = cdist(original_values_without_nan.reshape(-1, 1), imputed_values.reshape(-1, 1), metric=
# #         #distances = cdist(X_train.values.reshape(-1, 1), imputed_values.reshape(-1, 1), metric='euclidean'

# #         # Compute the mean distance
# #         mean_distance = np.mean(distances)

# #         print(f"K={k}, Mean Distance={mean_distance:.4f}")

# #         # Update the best K if current mean distance is lower
# #         if mean_distance < best_mean_distance:
# #             best_mean_distance = mean_distance
# #             best_k = k

# #     return best_k
```

```
# # k_values = [1, 3, 5, 7, 9]
# # optimal_k = knn_imputer_grid_search(df1, k_values)
# # print(f"Optimal K: {optimal_k}")



# OUTPUT

# Shape of original values after flattening: (5715,)
# Shape of imputed values after flattening: (5715,)
# K=1, Mean Distance=14.9319
# Shape of original values after flattening: (5715,)
# Shape of imputed values after flattening: (5715,)
# K=3, Mean Distance=14.9319
# Shape of original values after flattening: (5715,)
# Shape of imputed values after flattening: (5715,)
# K=5, Mean Distance=14.9319
# Shape of original values after flattening: (5715,)
# Shape of imputed values after flattening: (5715,)
# K=7, Mean Distance=14.9319
# Shape of original values after flattening: (5715,)
# Shape of imputed values after flattening: (5715,)
# K=9, Mean Distance=14.9319
# Optimal number of neighbors (K): 1
# Best Mean Distance: 14.9319
# Optimal K: 1
```

In [ ]:

c. KNN Classifier

In [ ]:

In [273…

```
def train_val_test_split(X,y):
    # Calculate the split indices
    split_train_idx = int(len(data) * 0.7)  # 70% for training
    split_temp_idx = int(len(data) * 0.85)  # 85% for training + validation (so 15% remains for testing)

    # Split the data into training, validation, and test sets
    train_data = data[:split_train_idx]      # First 70% for training
    validation_data = data[split_train_idx:split_temp_idx]  # Next 15% for validation
    test_data = data[split_temp_idx:]     # Last 15% for testing


    train_data = data[:split_train_idx]      # First 70% for training
    validation_data = data[split_train_idx:split_temp_idx]  # Next 15% for validation
    test_data = data[split_temp_idx:]

    return train_data, validation_data, test_data


def train_test_split(data, test_size=0.2):

    split_idx = int(len(data) * (1 - test_size))

    # Split the data into train and test sets
    train_data = data[:split_idx]
    test_data = data[split_idx:]

    return train_data, test_data
```

In [ ]:

In [ ]:

In [178…

```
from collections import Counter
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

def EuclideanDistance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))


def y_prediction(k_neighbors, y_train):
    y_pred = [y_train[i] for i in k_neighbors]
    return Counter(y_pred).most_common(1)[0][0] # Return the most common label (majority voting)

def predict_knn_class(x_train, y_train, k, z): # KNN for classification
    distances = []

    for i in range(len(x_train)): # We need to Calculate the Euclidean distance from the test point (z) to all p
```

```python
        distance = EuclideanDistance(z, x_train[i])
        distances.append((i, distance))  # Storing the index and corresponding distance

    distances.sort(key=lambda x: x[1])
    k_neighbors = [distances[i][0] for i in range(k)]  # taking indices of the k nearest neighbors

    return y_prediction(k_neighbors, y_train)  #  majority voting prediction

def grid_search_knn(X, y, k_values): # Function to perform grid search for finding optimal K value

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    best_k = None
    best_accuracy = 0

    for k in k_values:  # Trying different values of K and checking
        y_train_pred = []
        for i in range(len(X_train)):
            y_train_pred.append(predict_knn_class(X_train, y_train, k, X_train[i]))  # Get predictions for trai

        accuracy_train = accuracy_score(y_train, y_train_pred)

        y_test_pred = []  # Predict for the test set
        for i in range(len(X_test)):
            y_test_pred.append(predict_knn_class(X_train, y_train, k, X_test[i]))  # Getting test set predictio

        accuracy_test = accuracy_score(y_test, y_test_pred)


        if accuracy_test > best_accuracy:
            best_accuracy = accuracy_test
            best_k = k

        print(f"K={k}, Train Accuracy: {accuracy_train:.4f}, Test Accuracy: {accuracy_test:.4f}")

    print(f"Optimal K: {best_k}")
    print(f"Best Test Accuracy with K={best_k}: {best_accuracy:.4f}")

    return best_k, best_accuracy

def main_knn_example():

    # Creating dataset (500 rsamples, 5 features)  ## There were no target columns in the dodgr loop dataset
    np.random.seed(42)
    X = np.random.rand(500, 5)  #
    y = np.random.randint(0, 2, 500)  # 0 or 1

    k_values = [1, 3, 5, 7, 9]  # Perform grid search over different K values
    grid_search_knn(X, y, k_values)

main_knn_example()
```

```
K=1, Train Accuracy: 1.0000, Test Accuracy: 0.5100
K=3, Train Accuracy: 0.7750, Test Accuracy: 0.4500
K=5, Train Accuracy: 0.6875, Test Accuracy: 0.5100
K=7, Train Accuracy: 0.6175, Test Accuracy: 0.4800
K=9, Train Accuracy: 0.5800, Test Accuracy: 0.5400
Optimal K: 9
Best Test Accuracy with K=9: 0.5400
```

In [171…

```python
df1.head(3)
```

Out[171…

	0	1	2	3	4	5	6	7	8	9	...	279	280	281	282	283	284	285	286	287	288
0	1.0	7.0	3.0	6.0	11.0	8.0	6.0	6.0	10.0	4.0	...	12.0	5.0	9.0	4.0	4.0	6.0	9.0	5.0	16.0	8.0
1	1.0	9.0	10.0	5.0	7.0	10.0	9.0	5.0	6.0	8.0	...	8.0	5.0	4.0	8.0	6.0	11.0	5.0	8.0	9.0	6.0
2	1.0	12.0	18.0	11.0	11.0	19.0	17.0	4.0	6.0	8.0	...	10.0	9.0	11.0	8.0	4.0	7.0	3.0	6.0	3.0	6.0

3 rows × 289 columns

In [169…

```python
## df1.iloc[1] #better for numpy
```

Out[169…
```
0       1.0
1       9.0
2      10.0
3       5.0
4       7.0
       ... 
284    11.0
285     5.0
286     8.0
287     9.0
288     6.0
Name: 1, Length: 289, dtype: float64
```

In [ ]:

2. Decision Trees

In [286…
```
df3 = pd.read_csv('iris\iris.data',header=None)
df3[4].value_counts()
```

Out[286…
```
4
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: count, dtype: int64
```

In [192…
```
# df4 = pd.get_dummies(df3, columns=[4], drop_first=True)
# df4.head()
```

Out[192…

	0	1	2	3	4_Iris-versicolor	4_Iris-virginica
0	5.1	3.5	1.4	0.2	False	False
1	4.9	3.0	1.4	0.2	False	False
2	4.7	3.2	1.3	0.2	False	False
3	4.6	3.1	1.5	0.2	False	False
4	5.0	3.6	1.4	0.2	False	False

In [287…
```
df3[4] = df3[4].map({'Iris-setosa':'0', 'Iris-versicolor':'1', 'Iris-virginica':'2'})
df3[4] = df3[4].astype(float)
df3.head()
```

Out[287…

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	0.0
1	4.9	3.0	1.4	0.2	0.0
2	4.7	3.2	1.3	0.2	0.0
3	4.6	3.1	1.5	0.2	0.0
4	5.0	3.6	1.4	0.2	0.0

In [288…
```
x = df3.drop(columns=4)
y = df3[4]
```

In [ ]:

a) Class for Tree Node

In [289…
```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, *, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # Predicted value for leaf nodes

    def is_leaf_node(self):
        return self.value is not None
```

b) Functions to build the Tree using RSS as the criterion

In [290…
```
def rss(y):
    if len(y) == 0:  # Handle empty data
        return 0
    mean_y = np.mean(y)
```

```python
    return np.sum((y - mean_y) ** 2)


class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100, n_feats=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_feats = n_feats
        self.root = None

    def fit(self, X, y):
        self.n_feats = X.shape[1]
        self.root = self._grow_tree(X, y)

    # def predict(self, X):
    #     return np.array([self._traverse_tree(x, self.root) for x in X])


    def _grow_tree(self, X, y, depth=0):
        n_samples, n_features = X.shape

        # Stopping criteria
        if (depth >= self.max_depth or n_samples < self.min_samples_split):
            leaf_value = np.mean(y)
            return Node(value=leaf_value)

        feat_idxs = np.random.choice(n_features, self.n_feats, replace=False)

        # Greedy search
        best_feat, best_thresh = self._best_criteria(X, y, feat_idxs)

        # Growing the children
        left_idxs, right_idxs = self._split(X[:, best_feat], best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth + 1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth + 1)
        return Node(best_feat, best_thresh, left, right)


    def _best_criteria(self, X, y, feat_idxs):
        best_gain = -1
        split_idx, split_thresh = None, None
        for feat_idx in feat_idxs:
            X_column = X[:, feat_idx]
            thresholds = np.unique(X_column)
            for threshold in thresholds:
                gain = self._calculate_rss(y, X_column, threshold)

                if gain > best_gain:
                    best_gain = gain
                    split_idx = feat_idx
                    split_thresh = threshold

        return split_idx, split_thresh


    def _calculate_rss(self, y, X_column, split_thresh):

        left_idxs, right_idxs = self._split(X_column, split_thresh)

        # If no split (empty subset), return infinity to avoid it
        if len(left_idxs) == 0 or len(right_idxs) == 0:
            return float("inf")

        rss_left = rss(y[left_idxs])
        rss_right = rss(y[right_idxs])

        return rss_left + rss_right

    def _split(self, X_column, split_thresh):
        left_idxs = np.argwhere(X_column <= split_thresh).flatten()
        right_idxs = np.argwhere(X_column > split_thresh).flatten()

        if len(left_idxs) == 0 or len(right_idxs) == 0:
            print(f"Warning: One of the splits is empty (threshold: {split_thresh}).")
        return left_idxs, right_idxs


    # def _traverse_tree(self, x, node):
    #     if node.is_leaf_node():
    #         if node.value is None:
    #             print("error: leaf node value is None")
    #             raise ValueError("leaf node has no value.")
    #         return node.value
```

```
    #     if node.feature is None or node.threshold is None:
    #         print("Error: Node feature or threshold is None")
    #         raise ValueError("Node feature or threshold is None during traversal.")

    #     print(f"Traversing node: feature {node.feature}, threshold {node.threshold}, value {node.value}")
    #     if x[node.feature] <= node.threshold:
    #         return self._traverse_tree(x, node.left)
    #     return self._traverse_tree(x, node.right)



    def mean_squared_error(self, y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)

# x = np.array(x)
# y = np.array(y).flatten()
# X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
# clf = DecisionTree(max_depth=10)
# clf.fit(X_train, y_train)

#  # Make predictions on the test set
# y_pred = clf.predict(X_test)

# mse = clf.mean_squared_error(y_test, y_pred)
# print("Mean Squared Error:", mse)
```

c) Transversing the tree and making predictions

In [ ]:
```
def _traverse_tree(self, x, node):
    if node.is_leaf_node():
        return node.value

    if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)


def predict(self, X):
    return np.array([self._traverse_tree(x, self.root) for x in X])
```

In [ ]: