5CS037 - Concepts and Technologies of AI. Getting Started with Machine Learning Models. Implementation of `k-Nearest Neighbor` from Scratch.

1 Instructions

This worksheet contains programming exercises on building k-NN machine learning models based on the material discussed from the slides.This is a graded exercise and submission are mandatory.
Please answer the questions below using python in the Jupyter Notebook and follow the guidelines below:

- This worksheet must be completed individually.
- All the solutions must be written in Jupyter Notebook.
- Our Recommendation - `Google Colaboratory`.
- Dataset used for this session can be downloaded from shared drive.
- Complete the task only using core python library such as `Numpy, pandsa, matplotlib etc.`

Figure 1: Getting Started with Machine Learning.

———————— Here comes the Machine Learning!! ————————-

2 Building k-NN from Scratch.

1. k-NN Algorithm:

Algorithm 1 K-Nearest Neighbors (KNN) Algorithm

```
1: Input: Training dataset $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, test data $x_{\text{test}}$, number of neighbors $k$
2: Output: Predicted label for $x_{\text{test}}$
3: Step 1: For each training sample $x_i$, compute the distance $d(x_{\text{test}}, x_i)$ using Euclidean distance
4: Step 2: Sort the distances in ascending order and select the $k$ nearest neighbors
5: Step 3: Find the majority class among the $k$ nearest neighbors
6: Step 4: Return the majority class as the predicted label for $x_{\text{test}}$
```

2.1 Implementing k-NN Algorithm from scratch Step - step Guide:

Step - 1: Understanding Data:

Example Dataset: 'titanic.csv'.

1. Load Dataset: The following code loads a Titanic dataset, keeps only relevant columns, checks for missing data, and handles it by either filling with the mean (for columns with over 10% missing) or dropping rows (for columns with 10% or less missing).

Loading and Cleaning Data.

```python
import pandas as pd
import numpy as np
# Load the Titanic dataset
data = pd.read_csv("titanic.csv")
# Drop all categorical columns except 'Survived'
categorical_columns = data.select_dtypes(include=['object']).columns
data = data.drop(columns=[col for col in categorical_columns if col != 'Survived'])
# Check for missing values
missing_info = data.isnull().sum() / len(data) * 100
# Handle missing values
for column in data.columns:
    if missing_info[column] > 10: # If more than 10% missing
        data[column].fillna(data[column].mean(), inplace=True)
    else: # If less than 10% missing
        data.dropna(subset=[column], inplace=True)
# Display cleaned data
print("Data after processing:\n", data.head())
print("\nMissing values after processing:\n", data.isnull().sum())
```

- **Importing Libraries:** We imported `pandas` as `pd` for data manipulation and `numpy` as `np` (though `numpy` is not used in this snippet).
- **Loading the Titanic Dataset:** We loaded a dataset about Titanic passengers from a given URL using `pd.read_csv()` and stored it in a variable called `data`.
- **Dropping Categorical Columns:** We selected columns with categorical data (text-based columns) and dropped all of them except the `'Survived'` column. This means only the `'Survived'` column is kept if it is of type `object` (e.g., text), and all others are removed from the `data` DataFrame.

- **Checking for Missing Values:** We calculated the percentage of missing values in each column using

$$\texttt{missing\_info} = \frac{\texttt{data.isnull().sum()}}{\texttt{len(data)}} \times 100$$

  to identify columns that have missing data and how much of it is missing as a percentage.

- **Handling Missing Values:** We iterated over each column in the `data` DataFrame:
  - If a column had more than 10% missing values, we filled those missing values with the mean of that column using:

    `data[column].fillna(data[column].mean(), inplace=True)`

  - If a column had 10% or fewer missing values, we dropped any rows with missing values in that column using:

    `data.dropna(subset=[column], inplace=True)`

- **Displaying the Results:** We printed the first few rows of the cleaned data to see what it looks like after processing:

  `print("Data after processing:n", data.head())`

  We also printed the count of missing values in each column after processing to confirm that the missing values were handled appropriately:

  `print("nMissing values after processing:n", data.isnull().sum())`

2. Creating a Feature Matrix and Label Vector and Splitting `Train - Test Split:`This code separates the Titanic dataset into features (X) and target (y) and implements a custom train-test splitting function. It shuffles the data, splits it into training (70%) and testing (30%) sets, and verifies the output by displaying the shapes of the resulting datasets. This ensures that the model will have separate data for training and evaluation.

Feature Matrix and Label Vector with Train - Test Split:

```python
import numpy as np
# Separate features (X) and target (y)
X = data.drop(columns=['Survived']).values # Convert features to NumPy array
y = data['Survived'].values # Convert target to NumPy array
# Define a function for train-test split from scratch
def train_test_split_scratch(X, y, test_size=0.3, random_seed=42):
    """
    Splits dataset into train and test sets.
    Arguments:
    X : np.ndarray
        Feature matrix.
    y : np.ndarray
        Target array.
    test_size : float
        Proportion of the dataset to include in the test split (0 < test_size < 1).
    random_seed : int
        Seed for reproducibility.
    Returns:
    X_train, X_test, y_train, y_test : np.ndarray
        Training and testing splits of features and target.
    """
    np.random.seed(random_seed)
    indices = np.arange(X.shape[0])
    np.random.shuffle(indices) # Shuffle the indices
    test_split_size = int(len(X) * test_size)
    test_indices = indices[:test_split_size]
    train_indices = indices[test_split_size:]
    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]
    return X_train, X_test, y_train, y_test
# Perform the train-test split
X_train, X_test, y_train, y_test = train_test_split_scratch(X, y, test_size=0.3)
# Output shapes to verify
print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

- **Importing Required Library:** The `numpy` library was imported as `np` for numerical computations.
- **Separating Features and Target:**
  - `X` is created by dropping the `'Survived'` column from the `data` DataFrame and converting the remaining features into a NumPy array.
  - `y` is created by extracting the `'Survived'` column from the `data` DataFrame and converting it into a NumPy array. This column represents the target variable (whether a passenger survived or not).
- **Defining a Train-Test Split Function:** A custom function `train_test_split_scratch` is defined to split the dataset into training and testing subsets:
  - **Arguments:**
    - `X`: The feature matrix.
    - `y`: The target array.
    - `test_size`: The proportion of the dataset allocated to the test set (default is 0.3, meaning 30% of the data).
    - `random_seed`: A seed for reproducibility of random operations (default is 42).
  - **Steps in the Function:**
    - Set the random seed using `np.random.seed()` for reproducibility.
    - Generate an array of indices (`np.arange(X.shape[0])`) corresponding to the rows of `X`.
    - Shuffle these indices randomly using `np.random.shuffle()`.
    - Determine the number of samples in the test set as a proportion of the total dataset size:

$$\texttt{test\_split\_size} = \texttt{int(len(X) * test\_size)}$$

    - Split the indices into `test_indices` (first part) and `train_indices` (remaining part).
    - Use the indices to split `X` and `y` into `X_train`, `X_test`, `y_train`, and `y_test`.
- **Performing the Train-Test Split:** The custom function `train_test_split_scratch` is called with the feature matrix `X`, target array `y`, and a test size of 30%. The resulting training and testing datasets are stored in `X_train`, `X_test`, `y_train`, and `y_test`.
- **Verifying the Results:** The shapes of the resulting datasets (`X_train`, `X_test`, `y_train`, `y_test`) are printed to verify the split.

Step - 2 - Computing Euclidean Distance Metrics:

1. The function ensures that the inputs are compatible and calculates the Euclidean distance using the formula. It is versatile for n-dimensional spaces and raises errors for incompatible inputs.

Implementation of Euclidean Distance:

```python
def euclidean_distance(point1, point2):
    """
    Calculate the Euclidean distance between two points in n-dimensional space.
    Arguments:
    point1 : np.ndarray
        The first point as a numpy array.
    point2 : np.ndarray
        The second point as a numpy array.
    Returns:
    float
        The Euclidean distance between the two points.
    Raises:
    ValueError: If the input points do not have the same dimensionality.
    """
    # Check if the points are of the same dimension
    if point1.shape != point2.shape:
        raise ValueError("Points must have the same dimensions to calculate Euclidean distance.")
    # Calculate the Euclidean distance
    distance = np.sqrt(np.sum((point1 - point2) ** 2))
    return distance
```

- **Function Purpose:** The `euclidean_distance` function computes the straight-line distance between two points represented as NumPy arrays in any n-dimensional space.
- **Arguments:**
  - `point1`: A NumPy array representing the coordinates of the first point.
  - `point2`: A NumPy array representing the coordinates of the second point.
- **Return Value:** The function returns the Euclidean distance as a floating-point number.
- **Error Handling:**
  - If the dimensions of `point1` and `point2` do not match, the function raises a `ValueError` with the message:

    "Points must have the same dimensions to calculate Euclidean distance."
- **Calculation:** The Euclidean distance is calculated using the formula:

$$\text{distance} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

  where x_i and y_i are the corresponding coordinates of `point1` and `point2`.
- **Steps in the Function:**
  - Verify that `point1` and `point2` have the same dimensions:

    ```
    if point1.shape != point2.shape:
    ```

  If not, raise a `ValueError`.
  - Compute the squared differences between corresponding elements, sum them, and take the square root:

    `distance = np.sqrt(np.sum((point1 - point2) ** 2))`

Test Case for Euclidean Distance Computing Function:

```python
# Test case for the function
try:
    # Define two points
    point1 = np.array([3, 4])
    point2 = np.array([0, 0])
    # Calculate the distance
    result = euclidean_distance(point1, point2)
    # Check if the result matches the expected value (e.g., sqrt(3^2 + 4^2) = 5)
    expected_result = 5.0
    assert np.isclose(result, expected_result), f"Expected {expected_result}, but got {result}"
    print("Test passed successfully!")
except ValueError as ve:
    print(f"ValueError: {ve}")
except AssertionError as ae:
    print(f"AssertionError: {ae}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

This code is a test case to validate the functionality of the `euclidean_distance function`. The code helps:

- Computes distances accurately.
- Handles edge cases (e.g., mismatched dimensions) appropriately.
- Handles Errors:
  (a) ValueError: If the points have mismatched dimensions.
  (b) AssertionError: If the computed result does not match the expected value.
  (c) Other Exceptions: Catches unexpected errors.

Step - 3 - Implementation of core k-NN algorithm:

1. Predict the class label for a Single Query Point - knn_predict_single: This function implements the core logic of the K-Nearest Neighbors (KNN) algorithm for a single query point. It calculates the Euclidean distances from the query to all points in the training dataset, identifies the k nearest neighbors, and predicts the class label based on a majority vote among the neighbors. This function is useful for focused predictions on individual data points and acts as a modular building block for the generalized KNN algorithm.

Implementation of Core k-NN Algorithm:

```python
# Function for KNN prediction for a single query
def knn_predict_single(query, X_train, y_train, k=3):
    """
    Predict the class label for a single query using the K-nearest neighbors algorithm.
    Arguments:
    query : np.ndarray
        The query point for which the prediction is to be made.
    X_train : np.ndarray
        The training feature matrix.
    y_train : np.ndarray
        The training labels.
    k : int, optional
        The number of nearest neighbors to consider (default is 3).
    Returns:
    int
        The predicted class label for the query.
    """
    distances = [euclidean_distance(query, x) for x in X_train]
    sorted_indices = np.argsort(distances)
    nearest_indices = sorted_indices[:k]
    nearest_labels = y_train[nearest_indices]
    prediction = np.bincount(nearest_labels).argmax()
    return prediction
```

2. Predict Class Labels for All Test Samples - knn_predict: This function extends the KNN algorithm to handle multiple test samples simultaneously. By repeatedly calling knn_predict_single for each query in the test dataset, it predicts class labels for the entire test set. The function provides a streamlined interface to efficiently classify test points in bulk, leveraging the modularity of knn_predict_single.

Implementing kNN for whole Test Data set:

```python
# Function to test KNN for all test samples
def knn_predict(X_test, X_train, y_train, k=3):
    """
    Predict the class labels for all test samples using the K-nearest neighbors algorithm.
    Arguments:
    X_test : np.ndarray
        The test feature matrix.
    X_train : np.ndarray
        The training feature matrix.
    y_train : np.ndarray
        The training labels.
    k : int, optional
        The number of nearest neighbors to consider (default is 3).
    Returns:
```

```
    np.ndarray
        An array of predicted class labels for the test samples.
    """
    predictions = [knn_predict_single(x, X_train, y_train, k) for x in X_test]
    return np.array(predictions)
```

3. `Test Case for both above function:` This test case verifies the functionality of the `knn_predict` function using a small subset of the test data. It predicts class labels for the subset and compares them to the actual labels. The test ensures that the predictions have the correct shape and raises an error if they do not. If successful, it confirms the function's correctness and prints both the predictions and the actual labels.

   Test Function for knn_predict:

```
        # Test case for KNN on the Titanic dataset
# Assume X_train, X_test, y_train, and y_test have been prepared using the code above
try:
    # Define the test set for the test case
    X_test_sample = X_test[:5] # Taking a small subset for testing
    y_test_sample = y_test[:5] # Corresponding labels for the subset
    # Make predictions
    predictions = knn_predict(X_test_sample, X_train, y_train, k=3)
    # Print test results
    print("Predictions:", predictions)
    print("Actual labels:", y_test_sample)
    # Check if predictions match expected format
    assert predictions.shape == y_test_sample.shape, "The shape of predictions does not match the
        shape of the actual labels."
    print("Test case passed successfully!")
except AssertionError as ae:
    print(f"AssertionError: {ae}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Step - 4 - Computing Accuracy:

The compute_accuracy function calculates the accuracy of predictions by comparing the true labels `(y_true)` with the predicted labels `(y_pred)`. It computes the percentage of correct predictions out of the total predictions and returns the accuracy as a float value between 0 and 100.

Feature Matrix and Label Vector with Train - Test Split:

```
# Function to compute accuracy of predictions
def compute_accuracy(y_true, y_pred):
    """
    Compute the accuracy of predictions.
    Arguments:
    y_true : np.ndarray ; The true labels.
    y_pred : np.ndarray; The predicted labels.
    Returns:
    float : The accuracy as a percentage (0 to 100).
    """
    correct_predictions = np.sum(y_true == y_pred)
    total_predictions = len(y_true)
    accuracy = (correct_predictions / total_predictions) * 100
    return accuracy
```

The following code evaluates the KNN model's performance on the entire test set. It predicts the class labels for all test samples using the `knn_predict function` and calculates the model's accuracy with the `compute_accuracy function`. The accuracy is then displayed as a percentage. Any errors during prediction or computation are caught and reported.

Feature Matrix and Label Vector with Train - Test Split:

```
    # Perform prediction on the entire test set
try:
   # Make predictions on the entire test set
   predictions = knn_predict(X_test, X_train, y_train, k=3)
   # Compute the accuracy
   accuracy = compute_accuracy(y_test, predictions)
   # Print the accuracy
   print(f"Accuracy of the KNN model on the test set: {accuracy:.2f}%")
except Exception as e:
   print(f"An unexpected error occurred during prediction or accuracy computation: {e}")
```

Step - 5 - Experiment with different values of k:

The function `experiment_knn_k_values function` evaluates the KNN algorithm's performance for various values of `k` (number of neighbors). It predicts test labels for each k using the `knn_predict function`, computes the corresponding accuracy with `compute_accuracy`, and stores the results in a dictionary. Finally, it plots k values against their respective accuracies, providing a visual analysis of how the choice of k affects the model's performance. The function also prints the accuracy for each k.

Experimenting with Various k values:

```
# Function to test KNN on different values of k and plot the accuracies
import matplotlib.pyplot as plt
def experiment_knn_k_values(X_train, y_train, X_test, y_test, k_values):
   """
   Run KNN predictions for different values of k and plot the accuracies.

   Arguments:
   X_train : np.ndarray
      The training feature matrix.
   y_train : np.ndarray
      The training labels.
   X_test : np.ndarray
      The test feature matrix.
   y_test : np.ndarray
      The test labels.
   k_values : list of int
      A list of k values to experiment with.
   Returns:
   dict
      A dictionary with k values as keys and their corresponding accuracies as values.
   """
   accuracies = {}
   for k in k_values:
      # Make predictions using the current value of k
      predictions = knn_predict(X_test, X_train, y_train, k=k)
      # Compute the accuracy
      accuracy = compute_accuracy(y_test, predictions)
      accuracies[k] = accuracy
      print(f"Accuracy for k={k}: {accuracy:.2f}%")
```

```
    # Plot the accuracies
    plt.figure(figsize=(10, 5))
    plt.plot(k_values, list(accuracies.values()), marker='o')
    plt.xlabel('k (Number of Neighbors)')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy of KNN with Different Values of k')
    plt.grid(True)
    plt.show()
    return accuracies
```

The following code sets up an experiment to test the KNN model's accuracy across a range of k values from 1 to 20 (modifiable range). It calls the `experiment_knn_k_values function` to run predictions and accuracy calculations for each k and plots the results to show how accuracy varies with different numbers of neighbors. If an error occurs during the experiment, it is caught and reported. The output indicates the completion of the experiment and suggests checking the plot for trends in accuracy.

Test code for the Experiment:

```
# Define the range of k values to experiment with
k_values = range(1, 21) # You can adjust this range as needed
# Run the experiment
try:
    accuracies = experiment_knn_k_values(X_train, y_train, X_test, y_test, k_values)
    print("Experiment completed. Check the plot for the accuracy trend.")
except Exception as e:
    print(f"An unexpected error occurred during the experiment: {e}")
```

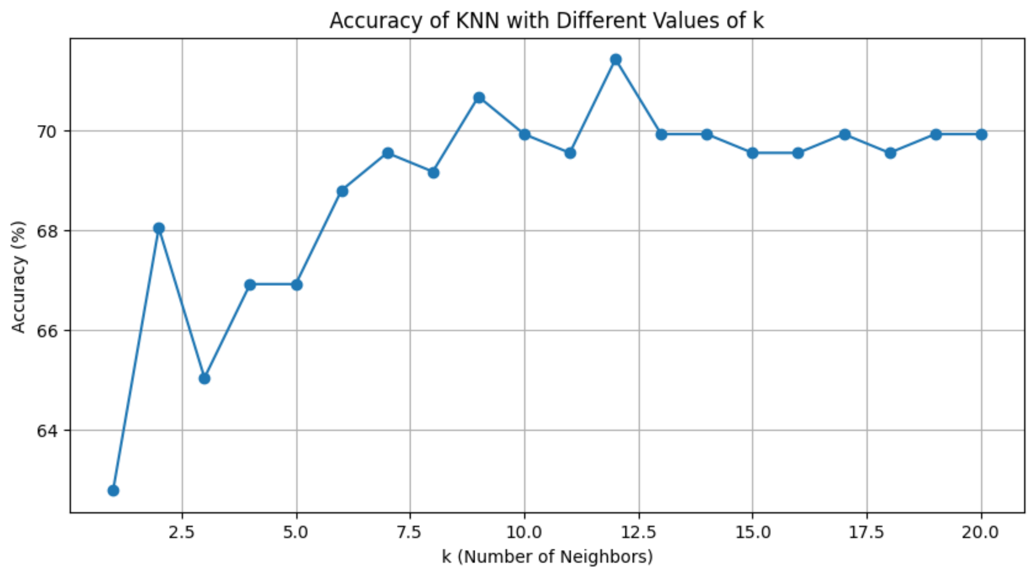

Figure 2: Experiment with different values of `k`.

3 To - Do Exercise:

For the provided dataset:

- `diabetes.csv`

Complete the following Problems.

Submission Instructions:

- Submit a single notebook containing:
  1. Clean and well-documented code.
  2. Outputs and visualizations.
  3. Detailed explanations and analysis for all steps.
- Ensure all cells are executed before submission.

Problem - 1: Perform a classification task with `knn` from scratch.

1. `Load the Dataset:`
   - Read the dataset into a pandas DataFrame.
   - Display the first few rows and perform exploratory data analysis (EDA) to understand the dataset (e.g., check data types, missing values, summary statistics).
2. `Handle Missing Data:`
   - Handle any missing values appropriately, either by dropping or imputing them based on the data.
3. `Feature Engineering:`
   - Separate the feature matrix `(X)` and target variable `(y)`.
   - Perform a train - test split from scratch using a $70\% - 30\%$ ratio.
4. `Implement KNN:`
   - Build the KNN algorithm from scratch (no libraries like sickit-learn for KNN).
   - Compute distances using Euclidean distance.
   - Write functions for:
     - Predicting the class for a single query.
     - Predicting classes for all test samples.
   - Evaluate the performance using accuracy.

Problem - 2 - Experimentation:

1. Repeat the Classification Task:
   - Scale the Feature matrix `X`.
   - Use the scaled data for training and testing the `kNN` Classifier.
   - Record the results.
2. Comparative Analysis: Compare the Results -
   - Compare the accuracy and performance of the `kNN` model on the original dataset from problem 1 versus the scaled dataset.
   - Discuss:
     - How scaling impacted the KNN performance.
     - The reason for any observed changes in accuracy.

Problem - 3 - Experimentation with `k`:

1. Vary the number of neighbors - `k`:
   - Run the KNN model on both the original and scaled datasets for a range of:

$$\texttt{k} = \mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots \mathbf{15}$$

   - For each `k`, record:
     - Accuracy.
     - Time taken to make predictions.
2. Visualize the Results:
   - Plot the following graphs:
     - `k vs. Accuracy` for original and scaled datasets.
     - `k vs. Time Taken` for original and scaled datasets.
3. Analyze and Discuss:
   - Discuss how the choice of `k` affects the accuracy and computational cost.
   - Identify the optimal `k` based on your analysis.

Problem - 4 - Additional Questions {Optional - But Highly Recommended}:

- Discuss the challenges of using KNN for large datasets and high-dimensional data.
- Suggest strategies to improve the efficiency of KNN (e.g., approximate nearest neighbors, dimensionality reduction).

——————— The - End ———————-