**Program 1: Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions.**

The Python code uses the scikit-learn library to perform a k-nearest neighbors (KNN) classification on the Iris dataset.

1. Import necessary libraries:
   * **load\_iris** from **sklearn.datasets** to load the Iris dataset.
   * **train\_test\_split** from **sklearn.model\_selection** to split the dataset into training and testing sets.
   * **KNeighborsClassifier** from **sklearn.neighbors** to create a KNN classifier.
   * **accuracy\_score** from **sklearn.metrics** to calculate the accuracy of the classifier.
2. Load the Iris dataset and separate the features (X) and target labels (y).
3. Split the dataset into a training set (X\_train, y\_train) and a testing set (X\_test, y\_test) using **train\_test\_split**. The testing set contains 30% of the data, and **random\_state** is set to 42 for reproducibility.
4. Define the number of neighbors **k** for the KNN classifier, and create the KNN classifier **knn** with **n\_neighbors** set to **k**.
5. Fit the KNN classifier with the training data (X\_train, y\_train) using **knn.fit**.
6. Use the trained KNN classifier to make predictions on the testing set (**X\_test**), and store the predictions in **y\_pred**.
7. Calculate the accuracy of the KNN model by comparing the predicted labels (**y\_pred**) with the true labels (**y\_test**) using **accuracy\_score**. The accuracy is expressed as a percentage and stored in the **accuracy** variable.
8. Initialize counters for correct and wrong predictions as **correct\_predictions** and **wrong\_predictions**, respectively.
9. Loop through the elements of the testing set (the length of **y\_test**).
10. Inside the loop, compare the predicted label (**y\_pred[i]**) with the true label (**y\_test[i]**). If they are the same, it's considered a correct prediction, and the information is printed to the console. If they are different, it's considered a wrong prediction, and that information is also printed.
11. Increment the appropriate counter (**correct\_predictions** or **wrong\_predictions**) based on whether the prediction was correct or wrong.
12. After the loop, print the overall accuracy, the total number of correct predictions, and the total number of wrong predictions.

This code performs KNN classification on the Iris dataset, makes predictions, and then calculates and displays the accuracy along with a breakdown of correct and wrong predictions for the test set. The **k** value, which represents the number of neighbors, can be adjusted to fine-tune the model's performance.

**Program 2:** **Develop a program to apply K-means algorithm to cluster a set of data stored in .CSV file. Use the same data set for clustering using EM algorithm. Compare the results of these two algorithms and comment on the quality of clustering.**

The Python code performs clustering on a dataset using two different clustering algorithms, K-means and Gaussian Mixture Models (GMM). It also evaluates the quality of the clustering using silhouette score and Davies-Bouldin Index.

1.Import necessary libraries:

* + **pandas** to work with data in a tabular format.
  + **numpy** for numerical operations.
  + **matplotlib.pyplot** for data visualization.
  + **KMeans** and **GaussianMixture** from **sklearn.cluster** for clustering algorithms.
  + **StandardScaler** from **sklearn.preprocessing** to standardize the data.
  + **silhouette\_score** and **davies\_bouldin\_score** from **sklearn.metrics** for evaluating clustering quality.

1. Read a dataset from a CSV file named 'data.csv' using **pd.read\_csv** and store it in the **data** variable.
2. Standardize the dataset using **StandardScaler** to make sure that all features have a mean of 0 and standard deviation of 1. The scaled data is stored in **data\_scaled**.
3. Create a NumPy array from the scaled data, which is stored in **data\_array**.
4. Perform K-means clustering on the standardized data with 3 clusters using the **KMeans** class, and assign the cluster labels to **kmeans\_clusters** by calling **kmeans.fit\_predict(data\_array)**.
5. Perform Gaussian Mixture Model (GMM) clustering on the standardized data with 2 components (which are equivalent to clusters) using the **GaussianMixture** class, and assign the cluster labels to **gmm\_clusters** by calling **gmm.fit(data\_array).predict(data\_array)**.
6. Create a 1x2 subplot for data visualization using **plt.figure** and **plt.subplot**. In the first subplot, scatter the data points in the standardized data, color-coded by K-means cluster labels. In the second subplot, scatter the data points color-coded by GMM cluster labels.
7. Display the plots using **plt.show()**.
8. Calculate the silhouette score and Davies-Bouldin Index for both K-means and GMM clustering results using **silhouette\_score** and **davies\_bouldin\_score**.
9. Print the silhouette score and Davies-Bouldin Index for both clustering algorithms.

The code clusters the data into groups using K-means and GMM, visualizes the results in two subplots, and then quantitatively evaluates the quality of the clustering using silhouette score and Davies-Bouldin Index. These metrics help assess the compactness and separation of clusters, with higher silhouette scores and lower Davies-Bouldin Index values indicating better clustering results.

Program 3: **Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs**

The Python code demonstrates Locally Weighted Regression (LOESS), a non-parametric regression technique that fits a smooth curve to a set of data points by giving different weights to nearby data points based on their proximity to the point of interest.

1. Import necessary libraries:
   * **numpy** for numerical operations.
   * **matplotlib.pyplot** for data visualization.
2. Set a random seed for reproducibility using **np.random.seed(0)**.
3. Generate synthetic data points:
   * Create a vector **X** containing 80 values evenly spaced between 0 and 5.
   * Compute the corresponding **y** values by taking the sine of **X** and then adding some random noise to simulate real-world data.
4. Define a function **loess(x, X, y, tau)** for LOESS:
   * **x** is the point at which you want to estimate the value.
   * **X** and **y** are the data points and their corresponding responses.
   * **tau** is the bandwidth or smoothing parameter, controlling the degree of local versus global influence.
5. Calculate the LOESS estimation at each **x\_pred** value (from 0 to 5) by applying the **loess** function in a loop. This process involves the following steps:
   * For each **x\_pred** value, calculate the weights for the data points based on their proximity to **x\_pred**. Weights are computed using a Gaussian kernel.
   * Multiply each **X** value by its corresponding weight and sum these weighted **X** values to get **weighted\_X**.
   * Multiply each **y** value by its corresponding weight and sum these weighted **y** values to get **weighted\_y**.
   * Calculate a weighted average, **theta**, of **X** using the formula **theta = sum(weighted\_X) / sum(weighted\_X^2)**.
   * Estimate the value of **y** at **x\_pred** using **y\_hat = theta \* x\_pred**.
6. Plot the data points as red 'x' markers and the LOESS curve as a blue line using **plt.scatter** and **plt.plot**. Add labels, a legend, and a title to the plot to make it informative.
7. Display the plot using **plt.show()**.

The code essentially implements LOESS regression by fitting a smooth curve to the data while giving more weight to nearby data points. The **tau** parameter controls the bandwidth or the degree of smoothing. The result is a locally adaptive regression that captures the underlying pattern in the data, and you can use this to make predictions for new values of **x**.

**Program 4: Build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets**

The Python code implements a simple feedforward neural network with one hidden layer to perform the XOR logic gate operation using the sigmoid activation function.

1. Import the necessary library, **numpy**, which is used for numerical operations.
2. Define two functions:
   * **sigmoid(x)**: This is the sigmoid activation function, which takes an input **x** and returns the sigmoid activation value.
   * **sigmoid\_derivative(x)**: This function computes the derivative of the sigmoid function, which is used during backpropagation.
3. Define the input data **X** and the corresponding target outputs **y** for the XOR logic gate. There are four input combinations (0, 0), (0, 1), (1, 0), and (1, 1), and the corresponding outputs are (0, 1, 1, 0).
4. Specify the network architecture parameters:
   * **i\_s** (input size) is set to 2, as there are two input features (0 and 1).
   * **hs** (hidden size) is set to 4, representing the number of neurons in the hidden layer.
   * **os** (output size) is set to 1, representing the number of output neurons.
5. Set the learning rate (**lr**) to 0.1 and the number of training epochs (**epochs**) to 10,000.
6. Initialize the neural network weights and biases:
   * **hw** (hidden layer weights) is initialized with random values in the range [0, 1).
   * **hb** (hidden layer biases) is initialized with zeros.
   * **ow** (output layer weights) is initialized with random values in the range [0, 1).
   * **ob** (output layer biases) is initialized with zeros.
7. Implement the training loop for the specified number of epochs. In each epoch, the following steps are performed:

a. Forward Pass:

* + Calculate the hidden layer output (**hlo**) by applying the sigmoid activation function to the dot product of the input (**X**) and hidden layer weights (**hw**) and adding the hidden layer biases (**hb**).
  + Calculate the final output (**olo**) by applying the sigmoid activation function to the dot product of the hidden layer output and the output layer weights (**ow**) and adding the output layer biases (**ob**).

b. Backpropagation:

* + Calculate the error (**d\_output**) by finding the element-wise difference between the true target outputs (**y**) and the predicted outputs (**olo**). This is multiplied by the derivative of the sigmoid function applied to **olo**.
  + Calculate the error for the hidden layer (**d\_hidden**) by finding the dot product of **d\_output** and the transpose of the output layer weights (**ow**). This is also multiplied by the derivative of the sigmoid function applied to **hlo**.

c. Update Weights and Biases:

* + Update the output layer weights (**ow**) by taking the dot product of the transpose of **hlo** and **d\_output**, and then multiplying by the learning rate (**lr**).
  + Update the output layer biases (**ob**) by summing the **d\_output** values along the rows and multiplying by the learning rate.
  + Update the hidden layer weights (**hw**) and biases (**hb**) in a similar manner.

1. After training, the neural network is tested by performing another forward pass to get the predicted outputs.
2. Print the predicted output values, which should approximate the XOR logic gate operation.

This code demonstrates a basic neural network that can learn and predict the XOR operation, which is a classic example of a problem that requires a non-linear decision boundary. The network uses the sigmoid activation function and backpropagation for training.

**Program 5: Demonstrate Genetic Algorithm by taking a suitable data for any simple application**

**The target string is "HELLO, WORLD!" and we want to evolve a population of random strings to eventually produce this target string.**

The Python code demonstrates a basic genetic algorithm for evolving a population of random strings to match a target string. The target in this example is "HELLO, WORLD!" and the algorithm aims to generate a population of strings that closely resemble the target string.

1. Import the necessary libraries: The code does not require any additional libraries beyond the Python standard library.
2. Define the target string as "HELLO, WORLD!"
3. Set parameters for the genetic algorithm:
   * **ps** (population size) is set to 200, representing the number of individuals in each generation.
   * **mr** (mutation rate) is set to 0.01, determining the probability of each character in a string being mutated.
4. Define three main functions:

a. **generate\_random\_string(length)**: This function generates a random string of a given **length** by selecting characters from the **string.printable** set, which includes all printable characters, such as letters, digits, and symbols.

b. **calculate\_fitness(string)**: This function calculates the fitness of a given string by counting the number of characters in the string that match the corresponding characters in the target string.

c. **select(population)**: This function selects an individual from the population based on their fitness. It calculates the total fitness of the population, selects a random value **r** between 0 and the total fitness, and iterates through the population, accumulating the fitness of individuals until **current\_sum** is greater than or equal to **r**. The selected individual is returned.

1. Initialize the population with random strings of the same length as the target string. Each individual in the population is created by calling **generate\_random\_string** within a loop.
2. Start the main loop for the genetic algorithm, which continues until a string matching the target is found.
3. In each generation:

a. Sort the population in descending order of fitness (i.e., the individuals that most closely match the target will appear at the beginning of the list).

b. Retrieve the best individual (with the highest fitness) from the sorted population.

c. Print the best individual for the current generation.

d. If the best individual matches the target string, exit the loop.

e. Otherwise, create a new generation by selecting two parents from the current population, splitting their strings at a random position, and recombining them to create a child. The child is then mutated by randomly replacing characters with new characters with a probability of **mr**.

1. The loop continues to the next generation, and the process repeats until a string matching the target is found.

This code demonstrates a simple genetic algorithm to evolve a population of strings to match a target string, with fitness determined by the number of characters that match the target. The algorithm uses selection, recombination, and mutation to evolve the population over generations, gradually improving the fitness of individuals until the target string is reached.

**Program 6: Demonstrate Q learning algorithm with suitable assumption for a problem statement**

**Assumed problem statement : a 2D grid world where an agent needs to find the shortest path to a goal while avoiding obstacles.**

The code implements Q-learning, a reinforcement learning algorithm, to find the optimal path in a grid world from the start state to the goal state while avoiding obstacles.

1. Define the environment, represented as a 2D grid world (**env**). In this grid:
   * **0** represents an empty cell.
   * **1** represents an obstacle.
   * **2** represents the goal.
2. Define the Q-table (**Q**) to store state-action values. The Q-table has one row for each state and one column for each possible action (up, down, left, right).
3. Set hyperparameters:
   * **learning\_rate** is the step size for updating Q-values.
   * **discount\_factor** is the discount factor for future rewards.
   * **exploration\_prob** is the probability of exploration (choosing a random action) rather than exploitation (choosing the action with the highest Q-value).
   * **num\_episodes** is the number of episodes (iterations) for training.
4. Define a function **state\_to\_index(state)** to convert a 2D state into a 1D index for the Q-table.
5. Perform Q-learning in a loop for the specified number of episodes:
   * Initialize the current state to the top-left corner (start state) and set the episode completion flag (**done**) to **False**.
   * While the episode is not completed:
     + Choose an action using an epsilon-greedy strategy:
       - With probability **exploration\_prob**, choose a random action.
       - Otherwise, choose the action with the highest Q-value for the current state.
     + Perform the selected action to transition to a new state.
     + Calculate the reward based on the new state:
       - If the new state is an obstacle (**1**), a penalty of **-1** is received.
       - If the new state is the goal (**2**), a reward of **10** is received, and the episode is marked as completed.
       - For other states, there is no immediate reward (**0**).
     + Update the Q-value for the current state-action pair using the Q-learning update rule.
     + Update the current state to the new state.
6. After training, the Q-table contains state-action values that have been learned from the environment.
7. Find the optimal path by starting from the start state and choosing the action with the highest Q-value at each step until the goal state is reached. This is done in a loop and the resulting path is stored in the **optimal\_path** list.
8. Print the optimal path, the Q-table, and the optimal path again to visualize the result.

The code demonstrates how Q-learning can be used to train an agent to find an optimal path in a grid world by learning state-action values and using them to make decisions that lead to the goal while avoiding obstacles.