APPLICATION OF ML IN INDUSTRIES

Department of Informatics
School of Computer Science
University of Petroleum & Energy Studies,
Dehradun-248007, Uttarakhand, India

LAB FILE

Submitted by:	**Submitted to:**
Name: Desh Iyer	Dr Mohammad Ahsan
SAP ID: 500081889	
Roll No: R214220386	
Batch: AIML, B5(H)	
Course: B-Tech CSE AIML (Hons.)	
Semester: VI	

INDEX

8.	Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering.	27/03/23	
9.	Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.	03/04/23	
10.	Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs.	10/04/23	

Experiment 1 - Find S Algorithm

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Implement and demonstrate the FIND S algorithm for finding the most specific hypothesis based on a given set of training data samples. Read the training data from a `.csv` file.

1.3 Theory

The FIND-S algorithm is a concept learning algorithm used to induce the most specific hypothesis from a set of training data samples. The goal of the algorithm is to generate a hypothesis that fits all positive training examples and no negative training examples. The algorithm works by initializing the hypothesis with the most specific possible hypothesis, which in the case of a boolean function is a conjunction of n negated literals, where n is the number of attributes in the data. The algorithm then iteratively refines the hypothesis by making it more specific based on the training data.

Here are the terms used in the description of the algorithm.

- A hypothesis h is a conjunction of n literals, where each literal can take one of two values: true or false.
- A training example is a pair (x, y), where x is an n-dimensional vector of attribute values, and y is either true or false.
- A positive example is a training example where y is true.
- A negative example is a training example where y is false.

1.4 Advantages

- The algorithm is simple and easy to understand.
- The algorithm is guaranteed to converge to the correct hypothesis, provided that the hypothesis space is rich enough to represent the target concept.

1.5 Limitations

- The algorithm can only handle boolean functions, and does not work with continuous or categorical data.

- The algorithm is sensitive to noise in the training data, and can produce an overly specific hypothesis if there are errors in the training data.
- The algorithm only produces a single hypothesis, and does not provide any measure of the quality or confidence of the hypothesis.

1.6 Pseudocode

The FIND-S algorithm starts by initializing the hypothesis h with the most specific hypothesis possible:

```
h ← < ¬, ¬, ..., ¬ >
```

Then, for each positive training example x in the training data, it updates the hypothesis by setting the i-th literal to true if the i-th attribute of x is true, and leaves it unchanged otherwise. For each negative training example, the algorithm does not update the hypothesis.

```
for each training example (x, y) do
  if y = true then
    for i = 1 to n do
      if h[i] = ¬ and x[i] = true then
        h[i] ← true
    end for
  end if
end for
```

2 Import Libraries

[ ]:
```python
import pandas as pd
import numpy as np
```

3 Implement Algorithm

Declare a function to calculate the final specific hypothesis given a vector of concepts (tuples) and a vector of targets.

[ ]:
```python
def train(concepts, targets, specificHypothesis):
    for i, val in enumerate(targets):
        if val == 'Yes':
            specificHypothesis = concepts[i]
            break

    for i, val in enumerate(concepts):
        if targets[i] == 'Yes':
            for i in range (len(specificHypothesis)):
                if val[i] != specificHypothesis[i]:
                    specificHypothesis[i] = '?'

    return specificHypothesis
```

4 Import Dataset

```
[ ]: data = pd.read_csv('../data/find-s.csv',␣
      ↪names=['Sky','Temperature','Humidity','Wind','Water','Forecast','Enjoy Sport?
      ↪'])
```

Here's what the data looks like in a data frame.

```
[ ]: data
```

```
[ ]:      Sky Temperature Humidity    Wind Water Forecast Enjoy Sport?
     0  Sunny        Warm   Normal  Strong  Warm     Same          Yes
     1  Sunny        Warm     High  Strong  Warm     Same          Yes
     2  Rainy        Cold     High  Strong  Warm   Change           No
     3  Sunny        Warm     High  Strong  Cool   Change          Yes
```

Retrieving the data points as the vector of concepts (tuples) of the data set.

```
[ ]: dataPoints = np.array(data)[:, :-1]
     phiLength = dataPoints.shape[0] + 1
```

Retrieving the target vector.

```
[ ]: dataTarget = np.array(data)[:, -1]
     dataTarget
```

```
[ ]: array(['Yes', 'Yes', 'No', 'Yes'], dtype=object)
```

5 Calculate Hypotheses

Defining the specific hypothesis to be all zeros initially.

```
[ ]: specificHypothesis = np.zeros(phiLength)
     specificHypothesis
```

```
[ ]: array([0., 0., 0., 0., 0.])
```

Calling the function defined above and obtaining our final hypothesis.

```
[ ]: print(f'The final hypothesis is: {train(dataPoints, dataTarget,␣
      ↪specificHypothesis)}')
```

```
The final hypothesis is: ['Sunny' 'Warm' '?' 'Strong' '?' '?']
```

Experiment 2 - Candidate-Elimination Algorithm

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

For a given set of training data examples stored in a `.csv` file, implement and demonstrate the candidate elimination algorithm to output a description of the set of all hypotheses consistent with the training examples.

1.3 Theory

The Candidate-Elimination algorithm is a concept learning algorithm used to find the set of all hypotheses that are consistent with a given set of training data samples. The algorithm maintains two sets of hypotheses: the set of most specific hypotheses S and the set of most general hypotheses G. Initially, S contains the most specific hypothesis possible, and G contains the most general hypothesis possible. The algorithm iteratively updates S and G based on the training data.

Here are the terms used in the description of the algorithm.

- A hypothesis h is a conjunction of n literals, where each literal can take one of two values: true or false.
- A training example is a pair (x, y), where x is an n-dimensional vector of attribute values, and y is either true or false.
- A positive example is a training example where y is true.
- A negative example is a training example where y is false.

1.4 Advantages

- The algorithm can handle noisy data and errors in the training data.
- The algorithm outputs a set of hypotheses that are consistent with the training data, which can be used for further analysis or decision making.

1.5 Limitations

- The algorithm can become computationally expensive when the number of attributes or the size of the hypothesis space is large.
- The algorithm does not provide any measure of the quality or confidence of the hypotheses.
- The algorithm assumes that the target concept is represented by a single consistent hypothesis.

1.6 Pseudocode

The Candidate-Elimination algorithm starts by initializing S with the most specific hypothesis possible, and G with the most general hypothesis possible:

```
S ← { <  ,  , ...,  > }
G ← { < ?, ?, ..., ? > }
```

Then, for each training example (x, y) in the training data, the algorithm updates S and G based on whether the example is positive or negative. For positive examples, the algorithm updates S to include only the hypotheses that are consistent with the example, and updates G to remove any hypotheses that are inconsistent with the example. For negative examples, the algorithm updates G to include only the hypotheses that are consistent with the example, and updates S to remove any hypotheses that are inconsistent with the example.

```
for each training example (x, y) do
    if y = true then
        S ← { h belongs to S : h(x) = y } # Keep only the hypotheses that are consistent with
        for g belongs to G do
        if g(x) != y and g is still in G then
            G ← G - { g } # Remove any hypotheses that are inconsistent with x.
            # add to G all minimal generalizations of h
            # that are consistent with all positive training examples seen so far
        end if
        end for
    else # y = false
        G ← { h  G : h(x) != y } # Keep only the hypotheses that are consistent with x.
        for s belongs to S do
        if s(x) = y and s is still in S then
            S ← S - { s } # Remove any hypotheses that are inconsistent with x.
            # add to S all minimal specializations of h
            # that are consistent with all negative training examples seen so far
        end if
        end for
    end if
end for
```

2 Import Libraries

[ ]:
```python
import numpy as np
import pandas as pd
```

3 Extract Data

[ ]:
```python
data = pd.read_csv('../data/candidate-elimination.csv')

concepts = np.array(data.iloc[:, 0:-1])
target = np.array(data.iloc[:, -1])
```

```
print("\nInstances are:\n",concepts)
print("\nTarget values are:\n",target)
```

```
Instances are:
 [['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
 ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
 ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']]

Target values are:
 ['Yes' 'No' 'Yes']
```

4 Define Function to Learn Dataset

```
[ ]: def learn(concepts, target):
         specific_h = concepts[0]
         general_h = [["?" for i in range(len(specific_h))] for i in␣
      ↪range(len(specific_h))]

         print("\nInitializing hypotheses")
         print("\nSpecific Boundary: ", specific_h)
         print("\nGeneric Boundary:\n",general_h)

         for i, h in enumerate(concepts):
             print("\nInstance", i + 1 , "is ", h)
             if target[i] == "yes":
                 print("Instance is Positive")
                 for x in range(len(specific_h)):
                     if h[x] != specific_h[x]:
                         specific_h[x] ='?'
                         general_h[x][x] ='?'

             if target[i] == "no":
                 print("Instance is Negative ")
                 for x in range(len(specific_h)):
                     if h[x] != specific_h[x]:
                         general_h[x][x] = specific_h[x]
                     else:
                         general_h[x][x] = '?'

             print("Specific boundary after", i + 1, "iteration is ", specific_h)
             print("Generic boundary after", i + 1, "iteration is ", general_h)

         indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?
      ↪', '?', '?']]
```

```
    for i in indices:
        general_h.remove(['?', '?', '?', '?', '?', '?'])
    return specific_h, general_h
```

5 Generate Hypotheses

[ ]:
```
s_final, g_final = learn(concepts, target)

print("\nFinal Specific Hypothesis: ", s_final, sep="\n")
print("Final General Hypothesis: ", g_final, sep="\n")
```

```
Initializing hypotheses

Specific Boundary:  ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']

Generic Boundary:
 [['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?',
'?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?',
'?'], ['?', '?', '?', '?', '?', '?']]

Instance 1 is  ['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
Specific boundary after 1 iteration is  ['Sunny' 'Warm' 'High' 'Strong' 'Warm'
'Same']
Generic boundary after 1 iteration is  [['?', '?', '?', '?', '?', '?'], ['?',
'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 2 is  ['Rainy' 'Cold' 'High' 'Strong' 'Warm' 'Change']
Specific boundary after 2 iteration is  ['Sunny' 'Warm' 'High' 'Strong' 'Warm'
'Same']
Generic boundary after 2 iteration is  [['?', '?', '?', '?', '?', '?'], ['?',
'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Instance 3 is  ['Sunny' 'Warm' 'High' 'Strong' 'Cool' 'Change']
Specific boundary after 3 iteration is  ['Sunny' 'Warm' 'High' 'Strong' 'Warm'
'Same']
Generic boundary after 3 iteration is  [['?', '?', '?', '?', '?', '?'], ['?',
'?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?',
'?', '?'], ['?', '?', '?', '?', '?', '?'], ['?', '?', '?', '?', '?', '?']]

Final Specific Hypothesis:
['Sunny' 'Warm' 'High' 'Strong' 'Warm' 'Same']
Final General Hypothesis:
[]
```

Experiment 3 - Decision Tree Using ID3

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample.

1.3 Theory

The ID3 (Iterative Dichotomiser 3) algorithm is a decision tree based algorithm used for classification in machine learning. It works by building a decision tree incrementally, using a heuristic called information gain to select the best attribute to split the data at each node. The algorithm starts with the entire training set and recursively splits it into smaller subsets based on the selected attributes until all the data in a subset belong to the same class. The decision tree can then be used for classification by traversing it from the root node to the leaf node that corresponds to the predicted class.

Here are the terms used in the description of the algorithm.

- A decision tree is a tree-like model used for decision-making in machine learning.
- A node in the decision tree represents a subset of the data.
- An attribute is a characteristic of the data that can be used to split it into subsets.
- A leaf node represents a class label.
- Information gain is a heuristic used to select the best attribute to split the data.

1.4 Advantages

- The algorithm is simple and easy to understand.
- The resulting decision tree is easy to interpret and can be used to explain the classification results.
- The algorithm can handle both categorical and numerical data.

1.5 Limitations

- The algorithm tends to overfit the training data, which can lead to poor generalization performance.

- The algorithm does not handle missing values well.
- The algorithm is sensitive to noisy data and outliers.

1.6 Pseudocode

In this pseudocode, `data` represents the training data, `attributes` represents the set of attributes in the data, and `target_attribute` represents the class label. The function returns a decision tree that can be used for classification.

```
function ID3(data, attributes, target_attribute)
    if all examples in data are of the same class:
        return a leaf node with the class label
    else if attributes is empty:
        return a leaf node with the majority class label
    else:
        best_attribute ← the attribute with the highest information gain
        tree ← a new decision tree with root node best_attribute
        for each value vi of best_attribute do
        subset ← the subset of examples in data with value vi for best_attribute
        if subset is empty:
            subtree ← a leaf node with the majority class label
        else:
            subtree ← ID3(subset, attributes - {best_attribute}, target_attribute)
        add subtree to tree as a child node with label vi
        return tree
```

2 Import Libraries

In the ID3 algorithm, decision trees are calculated using the concept of entropy and information gain.

[ ]:
```python
import pandas as pd
import numpy as np

# eps for making value a bit greater than 0 later on
eps = np.finfo(float).eps

from numpy import log2 as log
```

3 Creating Dataset

[ ]:
```python
dataset = {'Taste':
 ↪['Salty','Spicy','Spicy','Spicy','Spicy','Sweet','Salty','Sweet','Spicy','Salty'],
        'Temperature':
 ↪['Hot','Hot','Hot','Cold','Hot','Cold','Cold','Hot','Cold','Hot'],
        'Texture':
 ↪['Soft','Soft','Hard','Hard','Hard','Soft','Soft','Soft','Soft','Hard'],
        'Eat':['No','No','Yes','No','Yes','Yes','No','Yes','Yes','Yes']}
```

[ ]:
```
df = pd.DataFrame(dataset,columns=['Taste','Temperature','Texture','Eat'])
df
```

[ ]:
```
   Taste Temperature Texture  Eat
0  Salty         Hot    Soft   No
1  Spicy         Hot    Soft   No
2  Spicy         Hot    Hard  Yes
3  Spicy        Cold    Hard   No
4  Spicy         Hot    Hard  Yes
5  Sweet        Cold    Soft  Yes
6  Salty        Cold    Soft   No
7  Sweet         Hot    Soft  Yes
8  Spicy        Cold    Soft  Yes
9  Salty         Hot    Hard  Yes
```

4 Function to Calculate Entropy of a Label

[ ]:
```
def find_entropy(df):
    '''
    Function to calculate the entropy of a label
    '''
    Class = df.keys()[-1]
    entropy = 0
    values = df[Class].unique()
    for value in values:
        fraction = df[Class].value_counts()[value] / len(df[Class])
        entropy += -fraction * np.log2(fraction)
    return entropy
```

5 Function to Calculate Entropy of all Features

[ ]:
```
def find_entropy_attribute(df,attribute):
    '''
    Function to calculate the entropy of all features.
    '''
    Class = df.keys()[-1]
    target_variables = df[Class].unique()
    variables = df[attribute].unique()
    entropy2 = 0
    for variable in variables:
        entropy = 0
        for target_variable in target_variables:
                num =␣
↪len(df[attribute][df[attribute]==variable][df[Class]==target_variable])
                den = len(df[attribute][df[attribute]==variable])
                fraction = num/(den + eps)
                entropy += -fraction * log(fraction + eps)
        fraction2 = den / len(df)
        entropy2 += -fraction2 * entropy
    return abs(entropy2)
```

6 Function to Find the Feature with the Highest Information Gain

```
[ ]: def find_winner(df):
         '''
         Function to find the feature with the highest information gain.
         '''
         IG = []
         for key in df.keys()[:-1]:
         # Entropy_att.append(find_entropy_attribute(df,key))
             IG.append(find_entropy(df) - find_entropy_attribute(df,key))
         return df.keys()[:-1][np.argmax(IG)]
```

7 Function to Get a Sub-table of Met Conditions

```
[ ]: def get_subTable(df, node, value):
         '''
         Function to get a subTable of met conditions.

         node: Column name
         value: Unique value of the column
         '''
         return df[df[node] == value].reset_index(drop=True)
```

8 Function to Build the ID3 Decision Tree

```
[ ]: def buildTree(df,tree=None):
         '''
         Function to build the ID3 Decision Tree.
         '''
         Class = df.keys()[-1]
         #Here we build our decision tree

         #Get attribute with maximum information gain
         node = find_winner(df)
```

```
    #Get distinct value of that attribute e.g Salary is node and Low,Med and␣
↪High are values
    attValue = np.unique(df[node])

    #Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}

   #We make loop to construct a tree by calling this function recursively.
    #In this we check if the subset is pure and stops if it is pure.

    for value in attValue:

        subTable = get_subTable(df,node,value)
        clValue,counts = np.unique(subTable['Eat'],return_counts=True)

        if len(counts)==1:#Checking purity of subset
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subTable) #Calling the function␣
↪recursively

    return tree
```

9 Building the Decision Tree

```
[ ]: tree = buildTree(df)
```

The tree splits are as follows,

```
[ ]: import pprint
     pprint.pprint(tree)
```

```
{'Taste': {'Salty': {'Texture': {'Hard': 'Yes', 'Soft': 'No'}},
           'Spicy': {'Temperature': {'Cold': {'Texture': {'Hard': 'No',
                                                          'Soft': 'Yes'}},
                                     'Hot': {'Texture': {'Hard': 'Yes',
                                                         'Soft': 'No'}}}},
           'Sweet': 'Yes'}}
```

10 Function to Predict for Any Input

Now, for prediction we go through each node of the tree to find the output.

```
[ ]: def predict(inst,tree):
         '''
         Function to predict for any input variable.
         '''
         # Recursively we go through the tree that we built earlier

         for nodes in tree.keys():

             value = inst[nodes]
             tree = tree[nodes][value]
             prediction = 0

             if type(tree) is dict:
                 prediction = predict(inst, tree)
             else:
                 prediction = tree
                 break;

         return prediction
```

11 Predicting on Test Data

```
[ ]: data = {'Taste':'Salty','Temperature':'Cold','Texture':'Hard'}
```

```
[ ]: inst = pd.Series(data)
```

```
[ ]: prediction = predict(inst,tree)
     prediction
```

```
[ ]: 'Yes'
```

Experiment 4 - Artificial Neural Network

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Build an Artificial Neural Network (ANN) by implementing the Backpropagation algorithm and test the same using appropriate data sets.

1.3 Theory

Artificial Neural Networks (ANNs) are a set of algorithms that are designed to recognize patterns. They are inspired by the way the human brain works and are used to solve complex problems that cannot be easily solved using traditional methods. The Backpropagation algorithm is a supervised learning method that is commonly used to train ANNs. It is used to update the weights of the network so that it can better predict the output for a given input.

The Backpropagation algorithm consists of two phases: the forward phase and the backward phase. In the forward phase, the input is propagated through the network to produce an output. In the backward phase, the error between the predicted output and the actual output is calculated and used to update the weights of the network. This process is repeated until the error is minimized.

1.4 Mathematics of ANNs

Forward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$
$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}))$$
$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$
$$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$$

Backward propagation

$$dZ^{[2]} = A^{[2]} - Y$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m}\Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T}dZ^{[2]}.*g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = \frac{1}{m}dZ^{[1]}A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m}\Sigma dZ^{[1]}$$

Parameter updates

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

Vars and shapes

Forward prop

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]}A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]}A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

1.5 Steps

Here are the steps to implement the Backpropagation algorithm for an ANN:

1. Initialize the weights of the network randomly.
2. Feed the input through the network and calculate the output.
3. Calculate the error between the predicted output and the actual output.
4. Update the weights of the network using the error calculated in step 3.
5. Repeat steps 2-4 until the error is minimized.

1.6 Advantages

- ANNs can be used to solve complex problems that cannot be easily solved using traditional methods.
- Backpropagation is a powerful algorithm that can learn complex relationships between inputs and outputs.
- ANNs are highly parallelizable, which makes them suitable for use in large-scale applications.

1.7 Limitations

- ANNs can be difficult to train, and require a large amount of data and computational resources.
- ANNs can suffer from overfitting, where the network performs well on the training data but poorly on new data.
- ANNs can be difficult to interpret, and it can be hard to understand how they arrive at their predictions.

1.8 Pseudocode

Here's the pseudocode for the Backpropagation algorithm:

```
1. Initialize the weights randomly.
2. Repeat until the error is minimized:
    1. Feed the input through the network and calculate the output.
    2. Calculate the error between the predicted output and the actual output.
    3. Calculate the gradient of the error with respect to the weights.
    4. Update the weights using the gradient.
3. Return the trained network.
```

2 Check if GPU is Active

[ ]:
```
!nvidia-smi
```

```
Sun Apr 30 22:59:19 2023
+-------------------------------------------------------------------------------
--------+
| NVIDIA-SMI 530.30.02              Driver Version: 530.30.02    CUDA Version:
12.1     |
|-----------------------------------------+----------------------+--------------
--------+
| GPU  Name                  Persistence-M| Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp  Perf            Pwr:Usage/Cap|         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
MIG M. |
|=========================================+======================+==============
========|
|   0  NVIDIA GeForce GTX 1650 Ti      On | 00000000:01:00.0 Off |
N/A |
| N/A   49C    P0              18W /  50W|    574MiB /  4096MiB |     33%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+

+-------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
|===============================================================================
========|
|    0   N/A  N/A      2389      G   /usr/lib/xorg/Xorg
248MiB |
|    0   N/A  N/A      3016      G   /usr/bin/gnome-shell
32MiB |
|    0   N/A  N/A      4210      G   …5354413,16873341058258144221,131072
50MiB |
|    0   N/A  N/A      5857      G   …sion,SpareRendererForSitePerProcess
182MiB |
|    0   N/A  N/A     47450      G   …,WinRetrieveSuggestionsOnlyOnDemand
57MiB |
+-------------------------------------------------------------------------------
--------+
```

3 Importing Libraries and Data

[ ]:
```python
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Using a relative path instead of an absolute
path = r'./data/ann/train.csv'

data = pd.read_csv(path)
```

Loaded the data into a pandas dataframe.

[ ]:
```python
data.head()
```

```
[ ]:    label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  \
     0      1       0       0       0       0       0       0       0       0
     1      0       0       0       0       0       0       0       0       0
     2      1       0       0       0       0       0       0       0       0
     3      4       0       0       0       0       0       0       0       0
     4      0       0       0       0       0       0       0       0       0

        pixel8  …  pixel774  pixel775  pixel776  pixel777  pixel778  pixel779  \
     0       0  …         0         0         0         0         0         0
     1       0  …         0         0         0         0         0         0
     2       0  …         0         0         0         0         0         0
     3       0  …         0         0         0         0         0         0
     4       0  …         0         0         0         0         0         0

        pixel780  pixel781  pixel782  pixel783
     0         0         0         0         0
     1         0         0         0         0
     2         0         0         0         0
     3         0         0         0         0
     4         0         0         0         0

     [5 rows x 785 columns]
```

Converting the data from a dataframe to a numpy array.

```
[ ]: data = np.array(data)
```

Making sure that the model isn't overfitted, i.e., the model makes fairly accurate predictions for the training data but isn't generalised for the data it's supposed to have a high accuracy for. Setting aside a portion of the training data to perform cross-validation on to avoid overfitting.

Shuffling the data before we split the data into dev and training data. Note, `np.random.shuffle()` permutes the sequence in place.

```
[ ]: np.random.shuffle(data)
```

```
[ ]: data
```

```
[ ]: array([[9, 0, 0, …, 0, 0, 0],
            [6, 0, 0, …, 0, 0, 0],
            [5, 0, 0, …, 0, 0, 0],
            …,
            [4, 0, 0, …, 0, 0, 0],
            [1, 0, 0, …, 0, 0, 0],
            [3, 0, 0, …, 0, 0, 0]])
```

```
[ ]: # Storing the dimensions
     m, n = data.shape
```

```
# m - Number of images; n - label + pixels for each image
m, n
```

```
[ ]: (42000, 785)
```

Splitting the data into dev and training. We're using dev to cross validate and we're setting aside only 1000 images to do so.

```
[ ]: # Transposing the data using only 1000 images
data_dev = data[:1000].T

# Storing the labels in YDev
Y_dev = data_dev[0]

# Storing the pixels
X_dev = data_dev[1:]
X_dev = X_dev / 255
```

```
[ ]: # Storing the rest of the images
data_train = data[1000:m].T

# Extract labels
Y_train = data_train[0]

# Get the rest
X_train = data_train[1:]
X_train = X_train / 255
```

Printing details of all arrays implemented so far.

```
[ ]: print(
    f'Printing dimensions of all existing arrays:\n(i) X - pixels\nX_dev:␣
 ↪{X_dev.shape}\nX_train: {X_train.shape}\n\n(ii) Y - labels\nY_dev: {Y_dev.
 ↪shape}\nY_train: {Y_train.shape}')
```

```
Printing dimensions of all existing arrays:
(i) X - pixels
X_dev: (784, 1000)
X_train: (784, 41000)

(ii) Y - labels
Y_dev: (1000,)
Y_train: (41000,)
```

4 Defining Initial Parameters

Defining a function to initialise the neural network by creating random weights. We use `rand()` to obtain a random value between 0 and 1 and then we subtract from those values to make sure the range in which our random values lie is `[-0.5, 0.5]`.

[ ]:
```
def init_params():
    # There's 10 connections for each of the 784 nodes
    W1 = np.random.rand(10, n - 1) - 0.5

    # There's 10 biases
    b1 = np.random.rand(10, 1) - 0.5

    # Similarly,
    # There's 10 connections to 10 output nodes
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5

    return W1, b1, W2, b2
```

Function implementing the ReLU (rectified linear unit) activation function.

[ ]:
```
def ReLU(Z):
    # Taking the maximum element-wise using numpy
    return np.maximum(0, Z)
```

Function implementing the softmax activation function

[ ]:
```
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))

    # Returning the probability
    return A
```

5 Forward Propagation

Defining a function to implement forward propagation through the neural net.

[ ]:
```
def forward_propagation(W1, b1, W2, b2, X):
    # Deactivated first layer
    Z1 = W1.dot(X) + b1

    # Activating Z1
    A1 = ReLU(Z1)

    # Creating the next layer's deactivated input
    Z2 = W2.dot(A1) + b2
    # Since the next layer is the output layer, we apply softmax
    A2 = softmax(Z2)

    return Z1, A1, Z2, A2
```

Function to implement one-hot encoding of Y. This is to represent the target classes as an array instead of a label.

```
[ ]: def one_hot_encode(Y):
         # Encoding
         one_hot_encoded_df = pd.get_dummies(Y)

         # Taking the transpose so the columns represent images
         one_hot_encoded_array = np.array(one_hot_encoded_df).T

         return one_hot_encoded_array
```

Test to illustrate the working of `one_hot_encode(Y)`.

```
[ ]: test = Y_train[:20]
     test
```

```
[ ]: array([1, 3, 9, 3, 9, 9, 3, 9, 5, 1, 3, 0, 6, 8, 4, 5, 2, 4, 2, 5])
```

```
[ ]: df = pd.get_dummies(test).T
     df

     ls = np.array(df)
     ls
```

```
[ ]: array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
            [0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
           dtype=uint8)
```

Comparing it to our function.

```
[ ]: one_hot_encode(test)
```

```
[ ]: array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0],
            [0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
            [0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
           dtype=uint8)
```

Alternative one-hot encoding function.

```
[ ]: def one_hot(Y):
         one_hot_Y = np.zeros((Y.size, Y.max() + 1))

         one_hot_Y[np.arange(Y.size), Y] = 1

         one_hot_Y = one_hot_Y.T

         return one_hot_Y
```

Function to implement the derivative of ReLU.

```
[ ]: def derivative_ReLU(Z):
         # Returning 1 if the value is greater than 0. This is because the slope of␣
      ↪the `linear` thing is 1.
         return Z > 0
```

6 Back Propagation

Function to back propagate through the neural network to calculate the differences in the weights and biases.

```
[ ]: def back_propagation(Z1, A1, Z2, A2, W2, X, Y):
         m = Y.size
         # one_hot_encoded_Y = one_hot_encode(Y)

         # dZ2 = A2 - one_hot_encoded_Y

         one_hot_Y = one_hot(Y)
         dZ2 = A2 - one_hot_Y

         dW2 = (1 / m) * dZ2.dot(A1.T)
         db2 = (1 / m) * np.sum(dZ2)

         dZ1 = W2.T.dot(dZ2) * derivative_ReLU(Z1)
         dW1 = (1 / m) * dZ1.dot(X.T)
         db1 = (1 / m) * np.sum(dZ1)
```

```
    return dW1, db1, dW2, db2
```

7 Updating Parameters

Function to update the parameters using learning rate `alpha`.

```
[ ]: def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
         W1 = W1 - alpha * dW1
         b1 = b1 - alpha * db1

         W2 = W2 - alpha * dW2
         b2 = b2 - alpha * db2

         return W1, b1, W2, b2
```

8 Defining Gradient Descent

```
[ ]: def get_predictions(A):
         # Returns the indices of the max values
         return np.argmax(A, 0)
```

Testing the `get_predictions()` function.

```
[ ]: test, get_predictions(test)
```

```
[ ]: (array([1, 3, 9, 3, 9, 9, 3, 9, 5, 1, 3, 0, 6, 8, 4, 5, 2, 4, 2, 5]), 2)
```

```
[ ]: def get_accuracy(predictions, Y):
         # print(predictions, Y)

         return np.sum(predictions == Y) / Y.size
```

Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

```
[ ]: def gradient_descent(X, Y, epochs, alpha):
         # Defining weights and biases
         W1, b1, W2, b2 = init_params()

         for i in range(epochs):
             # Step 1
```

```
        Z1, A1, Z2, A2 = forward_propagation(W1, b1, W2, b2, X)

        # Step 2
        dW1, db1, dW2, db2 = back_propagation(Z1, A1, Z2, A2, W2, X, Y)

        # Step 3
        W1, b1, W2, b2 = update_params(
            W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)

        # Every 50th iteration
        if i % 50 == 0:
            # A2 is the output from the forward propagation
            predictions = get_predictions(A2)

            print("Epoch: {} | Accuracy: {:.2}".format(i,␣
↪get_accuracy(predictions, Y)))

    return W1, b1, W2, b2
```

9 Running the Neural Network

Running it on `X_train` and `Y_train`. We run it for 500 epochs with a learning rate of 1. As we can clearly tell, the output displays for each 10 iterations, a rapidly increasing accuracy settling at a nice 91.7%.

[ ]:
```
EPOCHS = 500
LEARNING_RATE = 1
```

[ ]:
```
W1, b1, W2, b2 = gradient_descent(X_train, Y_train, EPOCHS, LEARNING_RATE)
```

```
Epoch: 0 | Accuracy: 0.057
Epoch: 50 | Accuracy: 0.45
Epoch: 100 | Accuracy: 0.74
Epoch: 150 | Accuracy: 0.78
Epoch: 200 | Accuracy: 0.85
Epoch: 250 | Accuracy: 0.87
Epoch: 300 | Accuracy: 0.84
Epoch: 350 | Accuracy: 0.89
Epoch: 400 | Accuracy: 0.9
Epoch: 450 | Accuracy: 0.86
```

Running it on `X_dev` and `Y_dev` to cross-validate the model to overcome overfitting.

[ ]:
```
W1, b1, W2, b2 = gradient_descent(X_dev, Y_dev, EPOCHS, LEARNING_RATE)
```

```
Epoch: 0 | Accuracy: 0.11
Epoch: 50 | Accuracy: 0.6
Epoch: 100 | Accuracy: 0.72
```

```
Epoch: 150 | Accuracy: 0.94
Epoch: 200 | Accuracy: 0.97
Epoch: 250 | Accuracy: 0.98
Epoch: 300 | Accuracy: 0.98
Epoch: 350 | Accuracy: 0.98
Epoch: 400 | Accuracy: 0.99
Epoch: 450 | Accuracy: 0.99
```

10 Conclusion

Implementing the Backpropagation algorithm is a powerful technique for building an Artificial Neural Network. With the right data set and parameters, ANNs can be used to solve complex problems and provide insights that would be difficult or impossible to obtain using traditional methods.

Our NN will have a simple two-layer architecture. Input layer $a^{[0]}$ will have 784 units corresponding to the 784 pixels in each 28x28 input image. A hidden layer $a^{[1]}$ will have 10 units with ReLU activation, and finally our output layer $a^{[2]}$ will have 10 units corresponding to the ten digit classes with softmax activation.

Experiment 5 - Naive Bayes Classifier

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Write a program to implement the Naïve Bayesian classifier for a sample training data set stored as a `.csv` file. Compute the accuracy of the classifier, considering few test data sets.

1.3 Theory

The Naive Bayes Classifier is a probabilistic algorithm used for classification tasks. It is based on Bayes' theorem, which states that the probability of a hypothesis H given some observed evidence E is proportional to the probability of the evidence given the hypothesis times the prior probability of the hypothesis.

The Naive Bayes Classifier assumes that the features are independent of each other given the class label. That is, it assumes that the probability of observing a particular combination of feature values is the product of the probabilities of observing each individual feature value given the class label.

1.4 Steps

Here are the steps to implement the Naive Bayes Classifier:

1. Load the training data from the CSV file.
2. Preprocess the data, if necessary. This may involve steps such as removing missing values, converting categorical variables to numerical ones, and scaling numerical variables.
3. Split the data into training and testing sets.
4. Calculate the prior probabilities of each class label in the training set.
5. For each feature, calculate the conditional probabilities of each possible value given each class label in the training set.
6. For each test instance, calculate the posterior probability of each class label given the feature values using Bayes' theorem.
7. Assign the test instance to the class with the highest posterior probability.
8. Evaluate the accuracy of the classifier on the testing set by comparing the predicted class labels to the true class labels.

1.5 Pseudocode

Here's the pseudocode for the Naive Bayes Classifier:

```
# Load the data
data = load_csv('data.csv')

# Preprocess the data, if necessary
data = preprocess_data(data)

# Split the data into training and testing sets
train_set, test_set = split_data(data)

# Calculate the prior probabilities of each class label
priors = calculate_priors(train_set)

# Calculate the conditional probabilities of each feature given each class label
cond_probs = calculate_conditional_probs(train_set)

# Classify the test set
predictions = []
for instance in test_set:
    posterior_probs = calculate_posterior_probs(instance, priors, cond_probs)
    predicted_class = get_max_class(posterior_probs)
    predictions.append(predicted_class)

# Evaluate the accuracy of the classifier
accuracy = calculate_accuracy(test_set, predictions)
print('Accuracy:', accuracy)
```

2 Import Libraries

[ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

import pickle

import pandas as pd
```

3 Load Data

[ ]:
```
data = pd.read_csv('naive-bayes-classification-data.csv')
```

4 Train-test Split

```
[ ]: X_train, X_test, y_train, y_test = train_test_split(
         data[['glucose', 'bloodpressure']], data['diabetes'], test_size=0.3,␣
      ↪random_state=42)

     print(f'X_Train shape: {X_train.shape}\ny_train shape: {y_train.shape}')
```

```
X_Train shape: (696, 2)
y_train shape: (696,)
```

5 Define Gaussian Naive-Bayes Classifier

```
[ ]: classifier = GaussianNB()
     y_predicted = classifier.fit(X_train, y_train).predict(X_test)
```

6 Print Accuracy

```
[ ]: print("Number of mislabeled points out of a total %d points : %d" %
           (X_test.shape[0], (y_test != y_predicted).sum()))
     print("The resultant accuracy of the Gaussian Naive Bayes classifier is: %f" %
           accuracy_score(y_test, y_predicted))
```

```
Number of mislabeled points out of a total 299 points : 20
The resultant accuracy of the Gaussian Naive Bayes classifier is: 0.933110
```

7 Save Model using Pickle

```
[ ]: with open('model.pickle', 'wb') as f:
         pickle.dump(classifier, f)
```

Experiment 6 - Naive Bayes Classifier for Document Classification

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Assuming a set of documents that need to be classified, use the naive Bayesian Classifier model to perform this task. Built-in Java classes/API can be used to write the program. Calculate the accuracy, precision, and recall for your data set.

1.3 Naive Bayes Classifier for Document Classification

The Naive Bayes Classifier is a probabilistic algorithm used for classification. It is based on Bayes' theorem and the assumption of independence between features. The formula for Bayes' theorem is:

P(A|B) = P(B|A) * P(A) / P(B)

where A and B are events, P(A|B) is the probability of A given B, P(B|A) is the probability of B given A, P(A) is the prior probability of A, and P(B) is the prior probability of B.

In the context of classification, A is the class variable (e.g. spam or not spam), and B is a set of features or attributes (e.g. words in an email). The Naive Bayes Classifier calculates the probability of each class given the features using Bayes' theorem and the assumption of independence between features:

P(class|features) = P(features|class) * P(class) / P(features)

where class is the class variable, features is a set of features, P(class|features) is the probability of the class given the features, P(features|class) is the probability of the features given the class, P(class) is the prior probability of the class, and P(features) is the prior probability of the features.

To classify a new document, the Naive Bayes Classifier calculates the probability of each class given the features and chooses the class with the highest probability.

1.4 Pseudocode

1. Import the required libraries.
2. Load the dataset using pandas `read_csv` method.
3. Create feature vectors using `CountVectorizer` to convert text data into numerical data.

4. Split the dataset into training and testing sets using train_test_split method from `sklearn.model_selection`.
5. Train the Naive Bayesian classifier using `MultinomialNB` method from `sklearn.naive_bayes` with training set.
6. Make predictions on the testing set using predict method of classifier.
7. Calculate accuracy, precision, and recall scores.
8. Print scores.

2 Import Libraries

[ ]:
```python
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.model_selection import train_test_split
```

3 Load the Dataset

[ ]:
```python
# Load the dataset
data = pd.read_csv('./data/document-classification.txt')
```

4 Clean the Dataset

[ ]:
```python
list_1 = []

for i in data['5485']:
    list_1.append(int(i[0]))

data['Target'] = list_1

data = data.rename({'5485': 'Text'}, axis=1)
```

[ ]:
```python
data.head()
```

[ ]:
```
                                                Text  Target
0  1 champion products ch approves stock split ch…       1
1  2 computer terminal systems cpml completes sal…       2
2  1 cobanco inc cbco year net shr cts vs dlrs ne…       1
3  1 am international inc am nd qtr jan oper shr …       1
4  1 brown forman inc bfd th qtr net shr one dlr …       1
```

5 Create Vectorizer

```
[ ]: # create feature vectors using bag of words model
     vectorizer = CountVectorizer(stop_words='english')
     X = vectorizer.fit_transform(data['Text'])
```

6 Train-test Split

```
[ ]: # Split the dataset into training and testing sets
     X_train, X_test, y_train, y_test = train_test_split(X, data['Target'],␣
      ↪test_size=0.3, random_state=42)
```

7 Declare MulitnomialNB Classifier

```
[ ]: # Train the classifier
     classifier = MultinomialNB()
     classifier.fit(X_train, y_train)

     # Make predictions on the testing set
     y_pred = classifier.predict(X_test)
```

8 Calculate Classifier Scores

```
[ ]: # Calculate accuracy, precision and recall
     accuracy = accuracy_score(y_test, y_pred)
     precision = precision_score(y_test, y_pred, average='weighted')
     recall = recall_score(y_test, y_pred, average='weighted')
```

```
[ ]: print('Accuracy:', accuracy)
     print('Precision:', precision)
     print('Recall:', recall)
```

```
Accuracy: 0.9495747266099636
Precision: 0.9518234292669937
Recall: 0.9495747266099636
```

Experiment 7 - Modelling Medical Data with a Bayesian Network

April 30, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Write a program to construct a Bayesian network considering medical data. Use this model to demonstrate the diagnosis of heart patients using standard *Heart Disease Dataset*.

1.3 Theory

Bayesian networks are probabilistic graphical models that use Bayesian inference to model probabilistic relationships between a set of variables. In medical diagnosis, Bayesian networks are commonly used to represent the probabilistic dependencies between the symptoms and the underlying diseases. The network consists of a set of nodes representing variables and edges representing the probabilistic dependencies between them.

The Heart Disease Data Set is a standard data set that contains information about patients who have heart disease or not. The data set consists of 14 attributes including age, sex, chest pain type, blood pressure, serum cholesterol level, fasting blood sugar, electrocardiographic results, maximum heart rate achieved, exercise-induced angina, ST depression induced by exercise, slope of the peak exercise ST segment, number of major vessels colored by fluoroscopy, thal, and the target variable that indicates whether or not the patient has heart disease.

The goal of the Bayesian network is to use the probabilistic dependencies between the attributes to predict the probability of heart disease given the observed symptoms.

1.4 Steps to construct a Bayesian network for medical data

1. Identify the set of relevant variables: In this case, the relevant variables are the attributes in the Heart Disease Data Set.
2. Define the structure of the network: The structure of the network can be determined based on domain knowledge or by using a learning algorithm such as the K2 algorithm or the hill climbing algorithm.
3. Assign probabilities to the nodes: Once the structure of the network is determined, probabilities need to be assigned to the nodes based on the data. This can be done using maximum likelihood estimation or Bayesian estimation.

4. Inference: After constructing the network and assigning probabilities, the network can be used for inference to predict the probability of heart disease given the observed symptoms.

2 Import libraries

```
[ ]: import pandas as pd
     import bnlearn as bn
     from sklearn.metrics import accuracy_score
     from sklearn.model_selection import train_test_split
     import warnings
     warnings.filterwarnings("ignore")
```

3 Read Data from .csv File

```
[ ]: data = pd.read_csv(r'./data.csv')
     data
```

4 Exploring the Data

```
[ ]: data.columns
```

```
[ ]: Index(['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
            'exang', 'oldpeak', 'slope', 'ca', 'thal', 'num'],
           dtype='object')
```

```
[ ]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 303 entries, 0 to 302
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       303 non-null    int64
 1   sex       303 non-null    int64
 2   cp        303 non-null    int64
 3   trestbps  303 non-null    int64
 4   chol      303 non-null    int64
 5   fbs       303 non-null    int64
 6   restecg   303 non-null    int64
 7   thalach   303 non-null    int64
 8   exang     303 non-null    int64
 9   oldpeak   303 non-null    float64
 10  slope     303 non-null    int64
 11  ca        303 non-null    object
 12  thal      303 non-null    object
 13  num       303 non-null    int64
dtypes: float64(1), int64(11), object(2)
memory usage: 33.3+ KB
```

[ ]:
```python
data.describe()
```

[ ]:
```
              age         sex          cp    trestbps        chol         fbs  \
count  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000
mean    54.438944    0.679868    3.158416  131.689769  246.693069    0.148515
std      9.038662    0.467299    0.960126   17.599748   51.776918    0.356198
min     29.000000    0.000000    1.000000   94.000000  126.000000    0.000000
25%     48.000000    0.000000    3.000000  120.000000  211.000000    0.000000
50%     56.000000    1.000000    3.000000  130.000000  241.000000    0.000000
75%     61.000000    1.000000    4.000000  140.000000  275.000000    0.000000
max     77.000000    1.000000    4.000000  200.000000  564.000000    1.000000

          restecg     thalach       exang     oldpeak       slope         num
count  303.000000  303.000000  303.000000  303.000000  303.000000  303.000000
mean     0.990099  149.607261    0.326733    1.039604    1.600660    0.937294
std      0.994971   22.875003    0.469794    1.161075    0.616226    1.228536
min      0.000000   71.000000    0.000000    0.000000    1.000000    0.000000
25%      0.000000  133.500000    0.000000    0.000000    1.000000    0.000000
50%      1.000000  153.000000    0.000000    0.800000    2.000000    0.000000
75%      2.000000  166.000000    1.000000    1.600000    2.000000    2.000000
max      2.000000  202.000000    1.000000    6.200000    3.000000    4.000000
```

5 Extracting X and y

[ ]:
```python
X = data.iloc[:, :-1]
```

[ ]:
```python
y = data['num']
```

6 Train-test Split

[ ]:
```python
X_train, X_test, y_train, y_test = train_test_split(X, y, shuffle=True,␣
↪random_state=42)
```

[ ]:
```python
training = pd.concat([X_train, y_train], axis='columns')
testing = pd.concat([X_test, y_test], axis='columns')
```

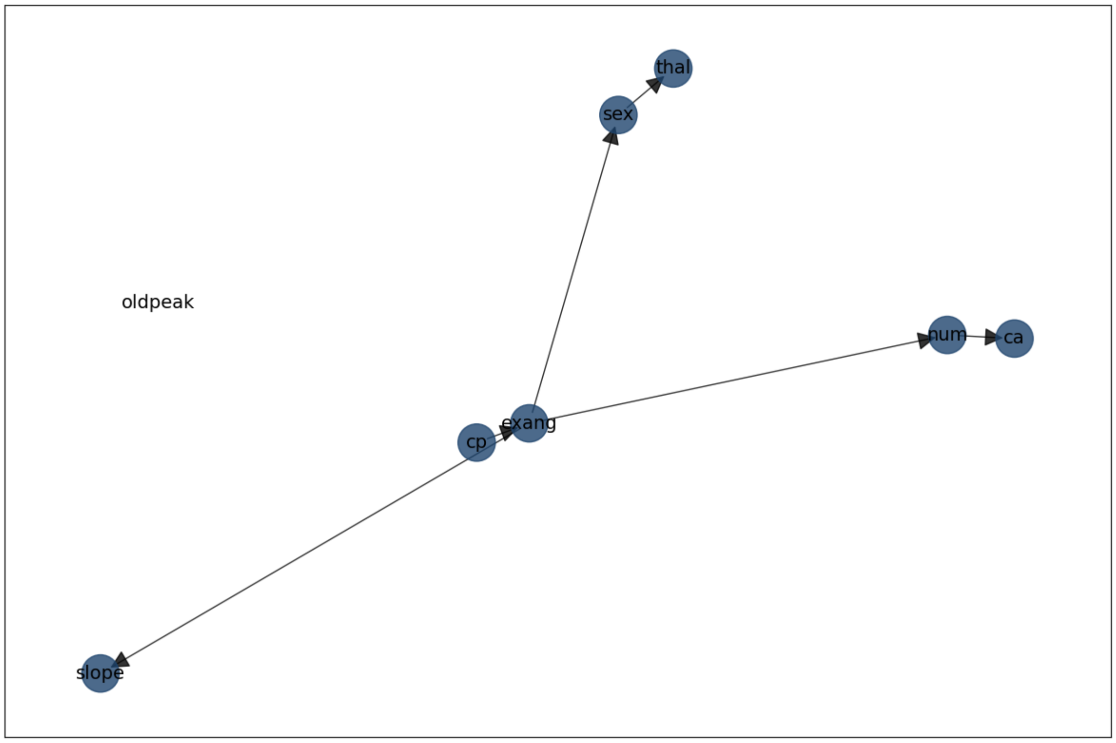
7 Plotting Bayesian Network

```
[ ]: DAG = bn.structure_learning.fit(training, methodtype='hc', root_node='sex',␣
      ↪bw_list_method='nodes', verbose=3)

     # Plot
     G = bn.plot(DAG)

     # Parameter learning
     model = bn.parameter_learning.fit(DAG, training, verbose=3)
```

```
[bnlearn] >Warning: Computing DAG with 14 nodes can take a very long time!
[bnlearn] >Computing best DAG using [hc]
[bnlearn] >Set scoring type at [bic]
[bnlearn] >Compute structure scores ['k2', 'bds', 'bic', 'bdeu'] for model
comparison (higher is better).
[bnlearn] >Set node properties.
[bnlearn] >Set edge properties.
[bnlearn] >Plot based on Bayesian model
```



```
[bnlearn] >Parameter learning> Computing parameters using [bayes]
[bnlearn] >Converting [<class 'pgmpy.base.DAG.DAG'>] to BayesianNetwork model.
[bnlearn] >Converting adjmat to BayesianNetwork.
[bnlearn] >CPD of sex:
```

Experiment 8 - Comparison of Clustering Algorithms

May 1, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Apply the Expectation-Maximization (EM) algorithm to cluster a set of data stored in a .CSV file. Use the same dataset for clustering using the k-Means algorithm. Compare the results of these two algorithms and comment on the quality of clustering.

1.3 Theory

Clustering is a technique used in unsupervised learning to group together similar data points. There are many different clustering algorithms, including k-Means and EM. The k-Means algorithm is a simple and widely-used clustering algorithm that partitions data into k clusters based on distance. The EM algorithm is a more complex clustering algorithm that involves estimating the probability distribution of the data.

The EM algorithm consists of two main steps: the E-step and the M-step. In the E-step, the algorithm estimates the probabilities of each data point belonging to each cluster. In the M-step, the algorithm updates the parameters of the probability distribution based on these probabilities. These two steps are repeated until convergence.

The k-Means algorithm also has two main steps: the assignment step and the update step. In the assignment step, each data point is assigned to the nearest centroid. In the update step, the centroids are updated based on the mean of the data points assigned to them. These two steps are repeated until convergence.

1.4 Steps

Here are the steps to apply the EM and k-Means algorithms for clustering:

1.4.1 EM Algorithm

1. Load the data from the .CSV file.
2. Initialize the parameters of the probability distribution.
3. Repeat until convergence:
    1. E-step: Calculate the probability of each data point belonging to each cluster.

    2. M-step: Update the parameters of the probability distribution based on these probabilities.
4. Assign each data point to the cluster with the highest probability.

1.4.2 k-Means Algorithm

1. Load the data from the .CSV file.
2. Initialize k centroids randomly.
3. Repeat until convergence:
    1. Assignment step: Assign each data point to the nearest centroid.
    2. Update step: Update the centroids based on the mean of the data points assigned to them.
4. Assign each data point to the cluster with the nearest centroid.

2 Import Required Libraries

[ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.mixture import GaussianMixture
```

[ ]:
```
# Load the IRIS dataset
iris = datasets.load_iris()
X = iris.data
```

[ ]:
```
# Define the number of clusters
k = 3
```

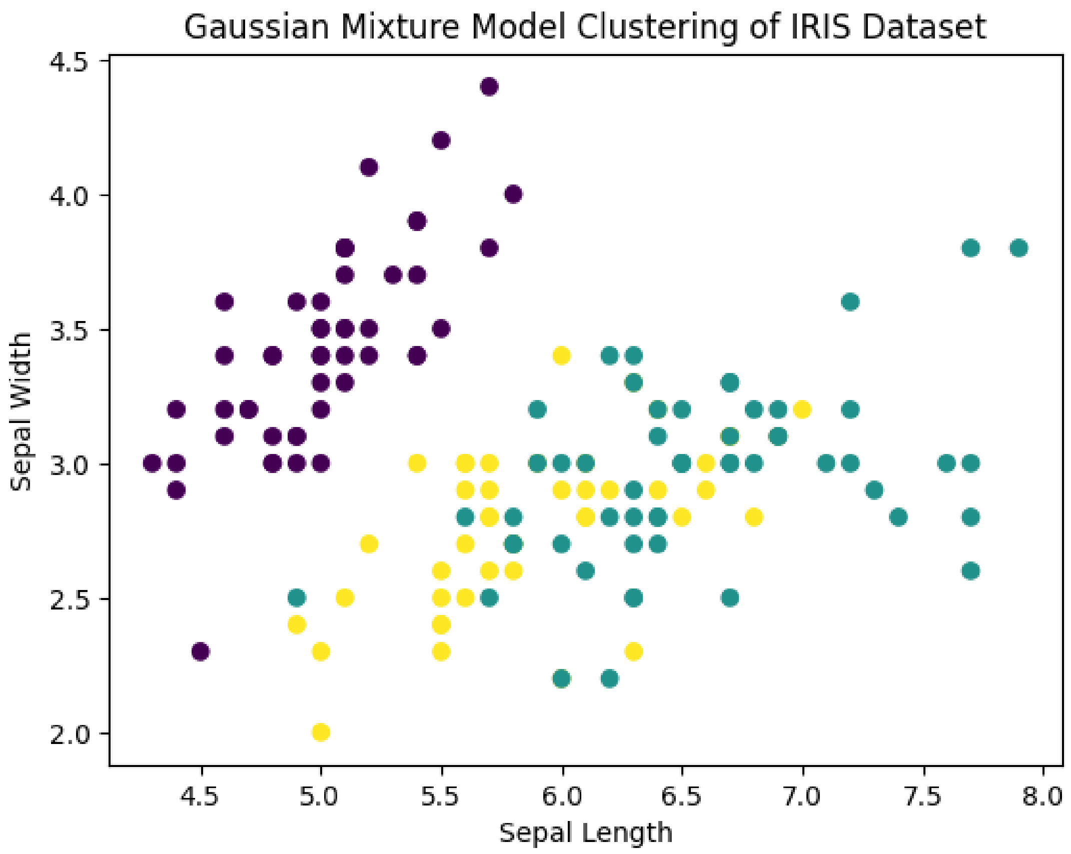
3 Clustering using a Gaussian Mixture and EM Algorithm

[ ]:
```
# Fit the Gaussian mixture model using the EM algorithm
gmm = GaussianMixture(n_components=k, covariance_type='full', random_state=0)
gmm.fit(X)
```

[ ]:
```
GaussianMixture(n_components=3, random_state=0)
```

[ ]:
```
# Get the predicted cluster labels
labels = gmm.predict(X)
```

[ ]:
```
# Create a scatter plot of the IRIS dataset with the predicted cluster labels
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Gaussian Mixture Model Clustering of IRIS Dataset')
plt.show()
```



4 Clustering using KNN

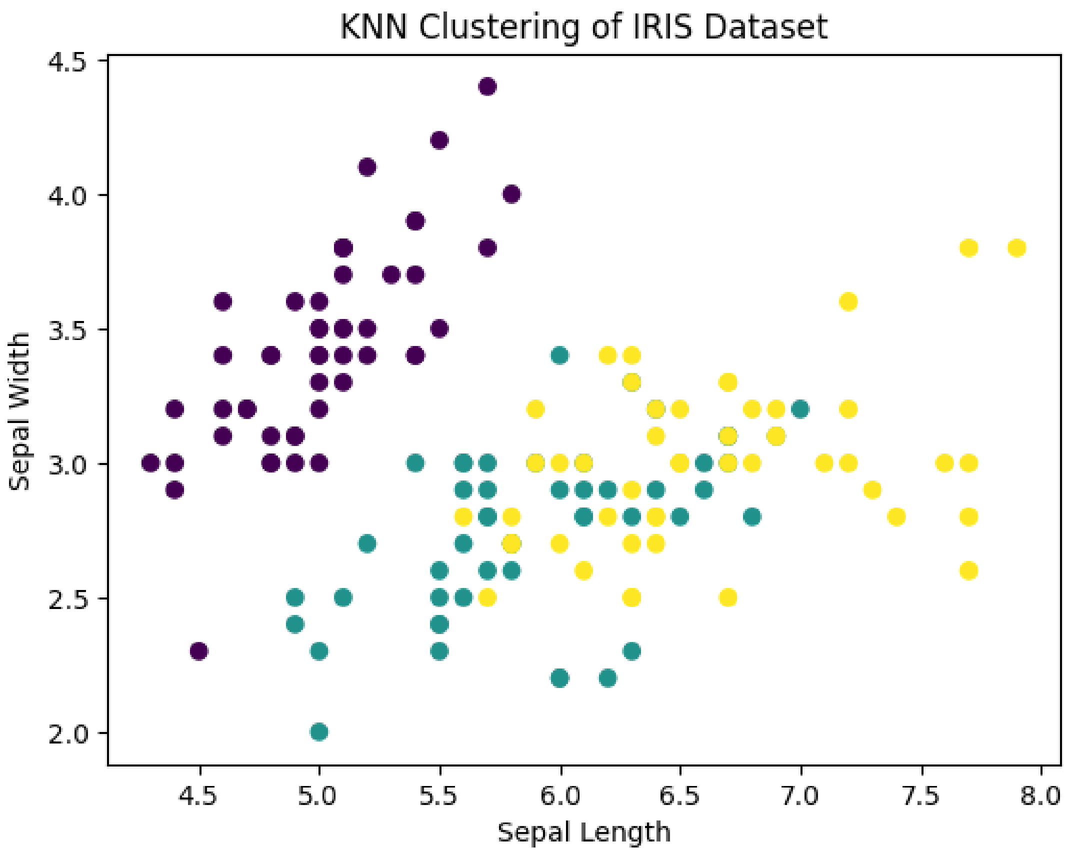
```
[ ]: # Define the number of clusters
     k = 3
```

```
[ ]: # Fit the KNN model
     knn = KNeighborsClassifier(n_neighbors=k)
     knn.fit(X, iris.target)
```

```
[ ]: KNeighborsClassifier(n_neighbors=3)
```

```
[ ]: # Get the predicted cluster labels
     labels = knn.predict(X)
```

```
[ ]: # Create a scatter plot of the IRIS dataset with the predicted cluster labels
     plt.scatter(X[:, 0], X[:, 1], c=labels)
     plt.xlabel('Sepal Length')
     plt.ylabel('Sepal Width')
     plt.title('KNN Clustering of IRIS Dataset')
plt.show()
```



5 Conclusion

5.1 Quality of Clustering:

- GMM/EM: The GMM/EM algorithm is a probabilistic clustering method that models each cluster as a Gaussian distribution. It can capture complex nonlinear relationships between variables and can work well with high-dimensional data. It also provides a measure of uncertainty in the form of posterior probabilities, which can be used to identify ambiguous points. However, it assumes that the data is generated from a mixture of Gaussian distributions, which may not always be true, and the results can be sensitive to the choice of initialization and the number of clusters.
- KNN: The KNN algorithm is a distance-based clustering method that assigns each data point to the nearest cluster center. It is simple and easy to implement and can work well with small datasets and simple patterns. However, it can be sensitive to the choice of distance metric, the number of neighbors, and the distribution of the data. It also does not provide a measure of uncertainty or the underlying probability distribution.

5.2 Pros and Cons:

- GMM/EM:
  - Pros:
    - Provides a measure of uncertainty and posterior probabilities.
    - Can capture complex nonlinear relationships between variables.
    - Can work well with high-dimensional data.
  - Cons:
    - Assumes that the data is generated from a mixture of Gaussian distributions.
    - Results can be sensitive to the choice of initialization and the number of clusters.
    - Can be computationally expensive for large datasets.
- KNN:
  - Pros:
    - Simple and easy to implement.
    - Can work well with small datasets and simple patterns.
    - Does not assume any underlying distribution of the data.
  - Cons:
    - Can be sensitive to the choice of distance metric and the number of neighbors.
    - Does not provide a measure of uncertainty or the underlying probability distribution.
    - Can be computationally expensive for large datasets with many features.

Overall, the choice between GMM/EM and KNN depends on the specific characteristics of the data and the problem at hand. If the data is high-dimensional and has complex nonlinear relationships, GMM/EM may be a better choice. On the other hand, if the data is low-dimensional and has simple patterns, KNN may be a better choice.

Experiment 9 - kNN Classifier

May 1, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Write a program to implement k-Nearest Neighbour (k-NN) algorithm to classify the iris dataset. Print both correct and wrong predictions. Java/Python ML library/classes can be used for this problem.

1.3 Theory

The k-NN algorithm is a type of supervised machine learning algorithm used for classification and regression. In the k-NN algorithm, a new data point is classified based on the k-nearest neighbours to that data point in the training set. The algorithm determines the k-nearest neighbours by calculating the distance between the new data point and all the data points in the training set.

The iris dataset is a commonly used dataset in machine learning for classification problems. It consists of 150 samples of iris flowers, with 50 samples from each of three different species of iris flowers. Each sample contains four features: sepal length, sepal width, petal length, and petal width.

1.4 Steps

Here are the steps to implement the k-NN algorithm to classify the iris dataset:

1. Load the iris dataset.
2. Split the dataset into training and testing sets.
3. Define the value of k.
4. For each data point in the testing set:
    1. Calculate the distance between the data point and all the data points in the training set.
    2. Select the k-nearest neighbours.
    3. Classify the data point based on the majority class of the k-nearest neighbours.
    4. Print whether the prediction is correct or wrong.
5. Calculate the accuracy of the classifier.

1.5 Pseudocode

Here's the pseudocode for the k-NN algorithm:

```
function knn_algorithm(train_set, test_set, k):
    predictions = []
    for i in range(len(test_set)):
        distances = []
        for j in range(len(train_set)):
            dist = euclidean_distance(test_set[i], train_set[j])
            distances.append((train_set[j], dist))
        distances.sort(key=lambda x: x[1])
        neighbors = [distances[m][0] for m in range(k)]
        prediction = majority_vote(neighbors)
        if prediction == test_set[i][-1]:
            print("Correct prediction:", prediction)
        else:
            print("Wrong prediction:", prediction)
        predictions.append(prediction)
    accuracy = calculate_accuracy(test_set, predictions)
    print("Accuracy:", accuracy)
```

2 Import libraries and data

[ ]:
```
# Import libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target
```

```
/home/volt/.local/lib/python3.10/site-packages/scipy/__init__.py:146:
UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this version
of SciPy (detected version 1.24.3
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

3 Train-test Split

[ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
```

4 Fitting data to a KNN Model

5 Determining the optimal K value through the elbow method

[ ]:
```
# Determine the best value for K using the elbow method
k_range = range(1, 21)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    scores.append(knn.score(X_test, y_test))

best_k = k_range[scores.index(max(scores))]
```

[ ]:
```
print(f'Scores: {scores}\nOptimal K value: {best_k}')
```

```
Scores: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
Optimal K value: 1
```

6 Creating the model

[ ]:
```
knn = KNeighborsClassifier(n_neighbors=best_k)
```

7 Fitting the data

[ ]:
```
knn.fit(X_train, y_train)
```

[ ]:
```
KNeighborsClassifier(n_neighbors=1)
```

8 Make predictions on the testing data

[ ]:
```
y_pred = knn.predict(X_test)
```

9 Printing wrong and right predictions

[ ]:
```
# Print correct and wrong predictions
for i in range(len(y_pred)):
    if y_pred[i] == y_test[i]:
        print(f"Correct prediction: actual={y_test[i]}, predicted={y_pred[i]}")
    else:
        print(f"Wrong prediction: actual={y_test[i]}, predicted={y_pred[i]}")
```

```python
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

```
Correct prediction: actual=1, predicted=1
Correct prediction: actual=0, predicted=0
Correct prediction: actual=2, predicted=2
Correct prediction: actual=1, predicted=1
Correct prediction: actual=1, predicted=1
Correct prediction: actual=0, predicted=0
Correct prediction: actual=1, predicted=1
Correct prediction: actual=2, predicted=2
Correct prediction: actual=1, predicted=1
Correct prediction: actual=1, predicted=1
Correct prediction: actual=2, predicted=2
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=1, predicted=1
Correct prediction: actual=2, predicted=2
Correct prediction: actual=1, predicted=1
Correct prediction: actual=1, predicted=1
Correct prediction: actual=2, predicted=2
Correct prediction: actual=0, predicted=0
Correct prediction: actual=2, predicted=2
Correct prediction: actual=0, predicted=0
Correct prediction: actual=2, predicted=2
Correct prediction: actual=2, predicted=2
Correct prediction: actual=2, predicted=2
Correct prediction: actual=2, predicted=2
Correct prediction: actual=2, predicted=2
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=1, predicted=1
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=2, predicted=2
Correct prediction: actual=1, predicted=1
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Correct prediction: actual=2, predicted=2
Correct prediction: actual=1, predicted=1
Correct prediction: actual=1, predicted=1
Correct prediction: actual=0, predicted=0
Correct prediction: actual=0, predicted=0
Accuracy: 1.0
```

10 Conclusion

The iris dataset has relatively few features and classes, with a clear separation between them. This means that the decision boundaries for a KNN classifier will be relatively simple, leading to high accuracy. A more complex dataset with overlapping classes and more ambiguous feature relationships would pose a greater challenge for a KNN classifier.

Experiment 10 - Non-Parametric Locally Weighted Regression

May 1, 2023

1 Experiment Details

1.1 Submitted By

Desh Iyer, 500081889, Year III, AI/ML(H), B5

1.2 Problem Statement

Implement the non-parametric Locally Weighted Regression algorithm to fit a curve to a set of data points. Select an appropriate dataset for your experiment and draw graphs to visualize the results.

1.3 Theory

Locally Weighted Regression (LWR) is a non-parametric algorithm used to fit a curve to a set of data points. Unlike parametric algorithms, LWR does not assume any specific functional form for the underlying data. Instead, it fits a curve to the data by using a weighted linear regression. The weights are determined by a kernel function that assigns higher weights to points that are closer to the target point and lower weights to points that are farther away.

The basic idea behind LWR is to use a linear regression to fit a curve to the data points that are closest to the target point. The weights are determined by a kernel function that assigns higher weights to points that are closer to the target point and lower weights to points that are farther away. This allows the algorithm to capture the local structure of the data and to fit a curve that is more flexible than a simple linear regression.

1.3.1 Parametric vs Non-Parametric Learning Algorithms

Parametric — In a Parametric Algorithm, we have a fixed set of parameters such as theta that we try to find(the optimal value) while training the data. After we have found the optimal values for these parameters, we can put the data aside or erase it from the computer and just use the model with parameters to make predictions. Remember, the model is just a function.

Non-Parametric — In a Non-Parametric Algorithm, you always have to keep the data and the parameters in your computer memory to make predictions. And that's why this type of algorithm may not be great if you have a really really massive dataset.

1.3.2 Need for NPLW Regression

We specifically apply this regression technique when the data to fit is non-linear. In Linear Regression we would fit a straight line to this data but that won't work here because the data is non-linear and our predictions would end up having large errors. We need to fit a curved line so that our error is minimized.

1.3.3 Under the Hood

In Locally weighted linear regression, we give the model the `x` where we want to make the prediction, then the model gives all the `x(i)`'s around that `x` a higher weight close to one, and the rest of `x(i)`'s get a lower weight close to zero and then tries to fit a straight line to that weighted `x(i)`'s data.

This means that if want to make a prediction for the green point on the x-axis (see figure below), the model gives higher weight to the input data i.e. `x(i)`'s near or around the circle above the green point and all else `x(i)` get a weight close to zero, which results in the model fitting a straight line only to the data which is near or close to the circle. The same goes for the purple, yellow, and grey points on the x-axis.

1.3.4 Calculating Error

In the loss function, it translates to error terms for the `x(i)`'s which are far from `x` being multiplied by almost zero and for the `x(i)`'s which are close to `x` get multiplied by almost 1. In short, it only sums over the error terms for the `x(i)`'s which are close to `x`.

1.4 Steps

Here are the steps to implement the Locally Weighted Regression algorithm:

1. Load the dataset.
2. Define the kernel function.
3. For each point in the dataset:
    1. Calculate the weights for the neighbouring points using the kernel function.
    2. Fit a linear regression to the neighbouring points using the weights.
    3. Predict the target value using the linear regression.
4. Plot the predicted values along with the original data points to visualize the results.

1.5 Advantages

- LWR is a non-parametric algorithm, which means it can fit a curve to the data without assuming any specific functional form for the underlying data.
- LWR can capture the local structure of the data and fit a curve that is more flexible than a simple linear regression.

1.6 Limitations

- LWR can be computationally expensive, especially for large datasets.
- LWR can be sensitive to the choice of kernel function and its parameters.

1.7 Pseudocode

Here's the pseudocode for the Locally Weighted Regression algorithm:

1. Load the dataset.
2. Define the kernel function.

3. For each point in the dataset:
    1. Calculate the weights for the neighbouring points using the kernel function.
    2. Fit a linear regression to the neighbouring points using the weights.
    3. Predict the target value using the linear regression.
4. Plot the predicted values along with the original data points to visualize the results.

2 Import Libraries

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
```

3 Define the Parameters of the Model

[ ]:
```
# Define global variables
numberSamples = 500

# Lambda function to de-linearize input function
deLinearize = lambda X: np.cos(1.5 * np.pi * X) + np.cos(5 * np.pi * X)

# Define X and y
X = np.sort(np.random.rand(numberSamples)) * 2
y = deLinearize(X) + np.random.randn(numberSamples) * 0.1

X = X.reshape(X.shape[0], 1)
y = y.reshape(y.shape[0], 1)

# Define tau
tauList = np.arange(0, 0.1, step=0.01)
```

4 Define a Function to Calculate the Weight Matrix

[ ]:
```
# Function to calculate weight matrix
def calculateWeightMatrix(point, X, tau):
    '''
    The parameters of this function are,
    tau --> bandwidth
    X --> Training data.
    point --> the x where we want to make the prediction.
    '''

    # m is the number of training examples.
    m = X.shape[0]
    # Initializing W as an identity matrix.
    w = np.mat(np.eye(m))

    # Calculating weights for all training examples [x(i)'s].
    for i in range(m):
        xi = X[i]
        d = (-2 * tau * tau)
        w[i, i] = np.exp(np.dot((xi - point), (xi - point).T) / d)

    return w
```

5 Define a Function to Predict for a Point in the Input Vector

[ ]:
```
# Function to predict for a single point in the input vector
def predictSinglePoint(X, y, point, tau):
    # Calculating the weight matrix using the wm function we wrote earlier.
    w = calculateWeightMatrix(point, X, tau)

    # Calculating parameter theta using the formula.
    theta = np.linalg.pinv(X.T * (w * X)) * (X.T * (w * y))

    # Calculating predictions.
    pointPrediction = np.dot(point, theta)

    # Returning the theta and predictions
    return theta, pointPrediction
```

6 Define a Function to Predict all Points for a Single Value of Tau

[ ]:
```
# Function to predict for a single tau value for all the points in the input␣
↪vector
def predictSingleTau(XTest, tau):
    # Empty list for storing predictions.
    predictionForSingleTau = []

    # Predicting for all numberPredictions values and storing them in␣
↪predictions.
    for point in XTest:
        _, pointPrediction = predictSinglePoint(X, y, point, tau)
        predictionForSingleTau.append(pointPrediction)

    # Reshaping predictions
    predictionForSingleTau = np.array(predictionForSingleTau).
↪reshape(numberSamples, 1)
```

```
    return predictionForSingleTau
```

7 Define Test Data

[ ]:
```
# Define testing data to predict
XTest = np.sort(np.random.rand(numberSamples)) * 2
XTest = np.array(XTest).reshape(numberSamples, 1)
```

8 Plot Predictions for Multiple Tau Values

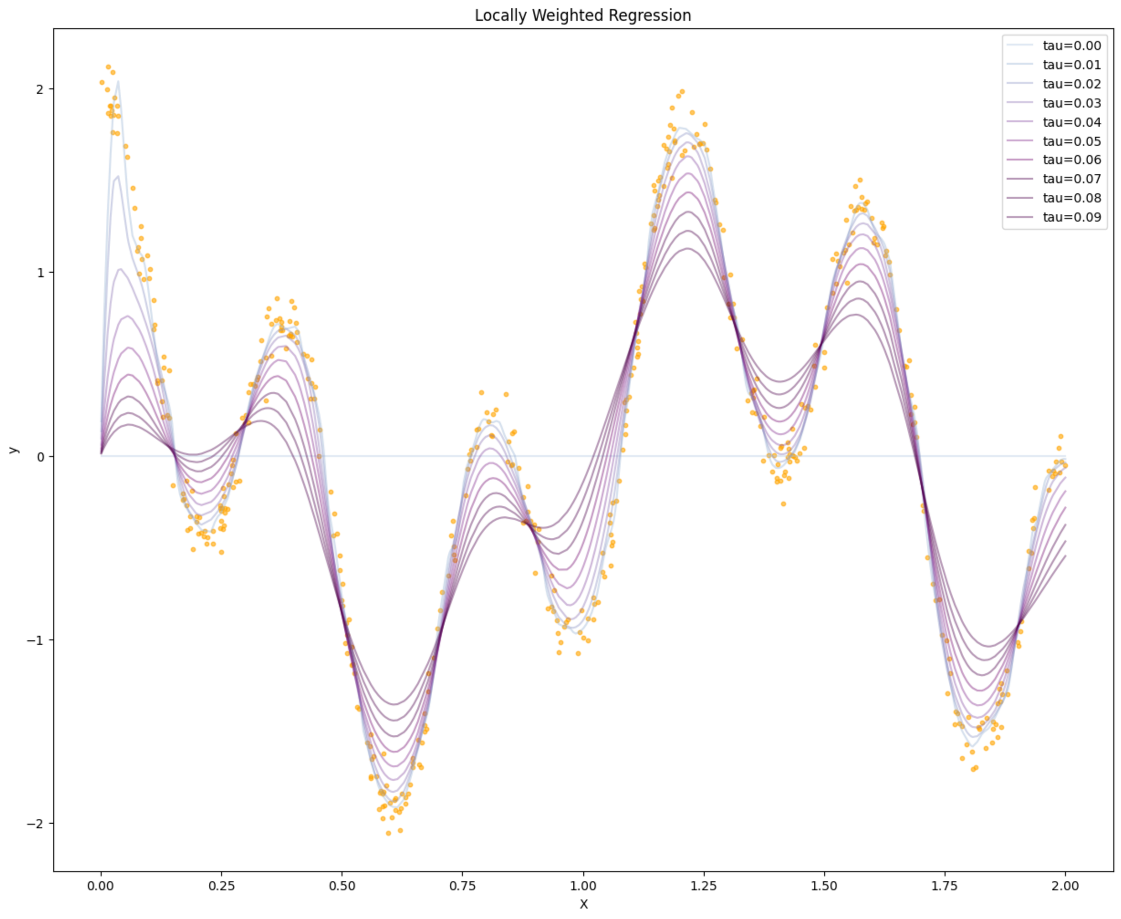
[ ]:
```
# Plot the training data
plt.figure(figsize=(15, 12))
# plt.figure(figsize=(20, 15), dpi=80)

plt.scatter(X, y, s=10, c='orange', marker='o', alpha=0.6)
# plt.plot(X, y, c='orange', marker='o', alpha=0.6)

# Lower value means that the higher tau values are darker in the gradient
colorDelta = 0.3

# Predict for each tau value and plot
for i, tau in enumerate(tauList):
    prediction = predictSingleTau(XTest, tau)
    color = plt.cm.BuPu(colorDelta + (i / len(tauList)))
    plt.plot(XTest, prediction, color=color, alpha=0.4, label=f'tau={tau:.2f}')

# Set plot attributes
plt.title("Locally Weighted Regression")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```



9 Conclusion

Implementing the Locally Weighted Regression algorithm is a powerful technique for fitting a curve to a set of data points. With the right dataset and parameters, LWR can capture the local structure of the data and fit a curve that is more flexible than a simple linear regression. Visualizing the results using plots can help to gain insights into the underlying data and the quality of the fit.