Programs

1. **Write a Python program to load iris data set and apply Naïve-Bayes algorithm for classification of Iris flowers.**

Overview of Naive Bayes Classification:

Naive Bayes is one such algorithm in classification that can never be overlooked upon due to its special characteristic of being “naive”. It makes the assumption that features of a measurement are independent of each other.

For example, an animal may be considered as a cat if it has cat eyes, whiskers and a long tail. Even if these features depend on each other or upon the existence of the other features, all of these properties independently contribute to the probability that this animal is a cat and that is why it is known as ‘Naive’.

According to Bayes Theorem, the various features are mutually independent. For two independent events, P(A,B) = P(A)P(B). This assumption of Bayes Theorem is probably never encountered in practice, hence it accounts for the “naive” part in Naive Bayes. Bayes’ Theorem is stated as: P(a|b) = (P(b|a) * P(a)) / P(b). Where P(a|b) is the probability of a given b.

Let us understand this algorithm with a simple example. The Student will be a pass if he wears a “red” color dress on the exam day. We can solve it using above discussed method of posterior probability.

By Bayes Theorem, P(Pass| Red) = P(Red| Pass) * P(Pass) / P (Red).

From the values, let us assume P (Red|Pass) = 3/9 = 0.33, P(Red) = 5/14 = 0.36, P(Pass)= 9/14 = 0.64. Now, P (Pass| Red) = 0.33 * 0.64 / 0.36 = 0.60, which has higher probability.

In this way, Naive Bayes uses a similar method to predict the probability of different class based on various attributes.

Problem Analysis:

To implement the Naive Bayes Classification, we shall use a very famous Iris Flower Dataset that consists of 3 classes of flowers. In this, there are 4 independent variables namely the, sepal_length, sepal_width, petal_length and petal_width. The dependent variable is the species which we will predict using the four independent features of the flowers.

There are 3 classes of species namely setosa, versicolor and the virginica. This dataset was originally introduced in 1936 by Ronald Fisher. Using the various features of the flower (independent variables), we have to classify a given flower using Naive Bayes Classification model.

Step 1: Importing the Libraries

As always, the first step will always include importing the libraries which are the NumPy, Pandas and the Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Step 2: Importing the dataset

In this step, we shall import the Iris Flower dataset which is stored in my github repository as IrisDataset.csv and save it to the variable dataset. After this, we assign the 4 independent variables to X and the dependent variable ‘species’ to Y. The first 5 rows of the dataset are displayed.

```
dataset = pd.read_csv('iris.csv')



X = dataset.iloc[:,:4].values
y = dataset['species'].values



dataset.head(5)



>>
sepal_length  sepal_width  petal_length  petal_width   species
5.1           3.5          1.4           0.2           setosa
4.9           3.0          1.4           0.2           setosa
4.7           3.2          1.3           0.2           setosa
4.6           3.1          1.5           0.2           setosa
5.0           3.6          1.4           0.2           setosa
```

Step 3: Splitting the dataset into the Training set and Test set

Once we have obtained our data set, we have to split the data into the training set and the test set. In this data set, there are 150 rows with 50 rows of each of the 3 classes. As each class is given in a continuous order, we need to randomly split the dataset. Here, we have the test_size=0.2, which means that 20% of the dataset will be used for testing purpose as the test set and the remaining 80% will be used as the training set for training the Naive Bayes classification model.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.2)
```

Step 4: Feature Scaling

The dataset is scaled down to a smaller range using the Feature Scaling option. In this, both the X_train and X_test values are scaled down to smaller values to improve the speed of the program.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Step 5: Training the Naive Bayes Classification model on the Training Set

In this step, we introduce the class GaussianNB that is used from the sklearn.naive_bayes library. Here, we have used a Gaussian model, there are several other models such as Bernoulli, Categorical and Multinomial. Here, we assign the GaussianNB class to the variable classifier and fit the X_train and y_train values to it for training purpose.

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
```

Step 6: Predicting the Test set results

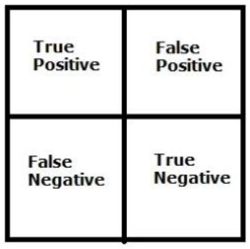
Once the model is trained, we use the the classifier.predict() to predict the values for the Test set and the values predicted are stored to the variable y_pred.

```
y_pred = classifier.predict(X_test)
y_pred
```

Step 7: Confusion Matrix and Accuracy

This is a step that is mostly used in classification techniques. In this, we see the Accuracy of the trained model and plot the confusion matrix.

The confusion matrix is a table that is used to show the number of correct and incorrect predictions on a classification problem when the real values of the Test Set are known. It is of the format.



The True values are the number of correct predictions made.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)

from sklearn.metrics import accuracy_score
print ("Accuracy : ", accuracy_score(y_test, y_pred))
cm

>>Accuracy :  0.9666666666666667

>>array([[14,  0,  0],
       [ 0,  7,  0],
       [ 0,  1,  8]])
```

From the above confusion matrix, we infer that, out of 30 test set data, 29 were correctly classified and only 1 was incorrectly classified. This gives us a high accuracy of 96.67%.

Step 8: Comparing the Real Values with Predicted Values

In this step, a Pandas DataFrame is created to compare the classified values of both the original Test set (y_test) and the predicted results (y_pred).

```
df = pd.DataFrame({'Real Values':y_test, 'Predicted Values':y_pred})

df

>>

Real Values   Predicted Values
setosa        setosa
setosa        setosa
virginica     virginica
versicolor    versicolor
setosa        setosa
setosa        setosa
...  ...   ...  ...  ...
virginica     versicolor
virginica     virginica
setosa        setosa
setosa        setosa
versicolor    versicolor
versicolor    versicolor
```

This step is an additional step which is not much informative as the Confusion matrix and is mainly used in regression to check the accuracy of the predicted value.

As you can see, there is one incorrect prediction that has predicted versicolor instead of virginica.

2. Write a Python program to extract iris.csv file. Apply k-Nearest Neighbor technique to identify the users who purchased the item or not.

KNN (K Nearest Neighbors) algorithm is a supervised Machine Learning classification algorithm. It is one of the simplest and widely used classification algorithms in which a new data point is classified based on similarity in the specific group of neighboring data points. This gives a competitive result.

Working:

For a given data point in the set, the algorithms find the distances between this and all other K numbers of data point in the dataset close to the initial point and votes for that category that has the most frequency. Usually, Euclidean distance is taking as a measure of distance. Thus the end resultant model is just the labeled data placed in a space. This algorithm is popularly known for various applications like genetics, forecasting, etc. The algorithm is best when more features are present.

KNN reducing over fitting is a fact. On the other hand, there is a need to choose the best value for K. So now how do we choose K? Generally we use the Square root of the number of samples in the dataset as value for K. An optimal value has to be found out since lower value may lead to overfitting and higher value may require high computational complication in distance. So using an error plot may help. Another method is the elbow method. You can prefer to take root else can also follow the elbow method.

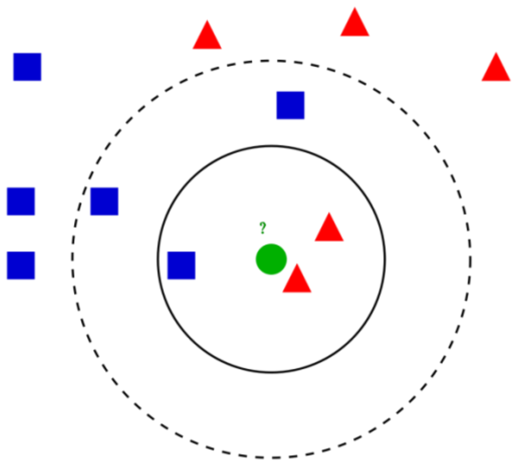
Different steps of K-NN for classifying a new data point:

Step 1: Select the value of K neighbors (say k=5)

Step 2: Find the K (5) nearest data point for our new data point based on Euclidean distance.

Step 3: Among these K data points count the data points in each category.

Step 4: Assign the new data point to the category that has the most neighbors of the new data point.



Example:

Consider an example problem for getting a clear intuition on the K -Nearest Neighbor classification. We are using the Social network ad dataset. The dataset contains the details of users in a social networking site to find whether a user buys a product by clicking the ad on the site based on their salary, age, and gender.

Importing essential libraries:

```
import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import sklearn
```

Importing of the dataset and slicing it into independent and dependent variables:

```
dataset = pd.read_csv('iris.csv')

X = dataset.iloc[:, [1, 2, 3]].values

y = dataset.iloc[:, -1].values
```

Since the dataset containing character variables, need to encode it using LabelEncoder.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

X[:,0] = le.fit_transform(X[:,0])
```

Split the dataset into train and test set. Providing the test size as 0.20, that means training sample contains 320 training set and test sample contains 80 tests set.

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =
0.20, random_state = 0)
```

Next, feature scaling is done to the training and test set of independent variables for reducing the size to smaller values.

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)
```

Build and train the K Nearest Neighbor model with the training set.

```
from sklearn.neighbors import KNeighborsClassifier

classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski',
p = 2)

classifier.fit(X_train, y_train)
```

Three different parameters are used in the model creation. n_neighbors is setting as 5, which means 5 neighborhood points are required for classifying a given point. The distance metric used is Minkowski. Equation for the same is given below.

$$\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$$

As per the equation, we need to select the p-value.

p = 1 , Manhattan Distance

p = 2 , Euclidean Distance

p = infinity , Cheybchev Distance

In this example, we are choosing the p value as 2. Machine Learning model is created, now we have to predict the output for the test set.

```
y_pred = classifier.predict(X_test)
```

Comparing true and predicted value:

```
y_test
```

>>

```
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1,

       0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,

       1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,

       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1,

       1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1], dtype=int64)

y_pred
```

>>

```
array([0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1,

       0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,

       1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1,

       0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1,

       1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1], dtype=int64)
```

Evaluating the model using the confusion matrix and accuracy score by comparing the predicted and actual test values.

```
from sklearn.metrics import confusion_matrix,accuracy_score

cm = confusion_matrix(y_test, y_pred)

ac = accuracy_score(y_test,y_pred)
```

Confusion Matrix :

```
cm
```

>>

```
[[64  4]
 [ 3 29]]
```

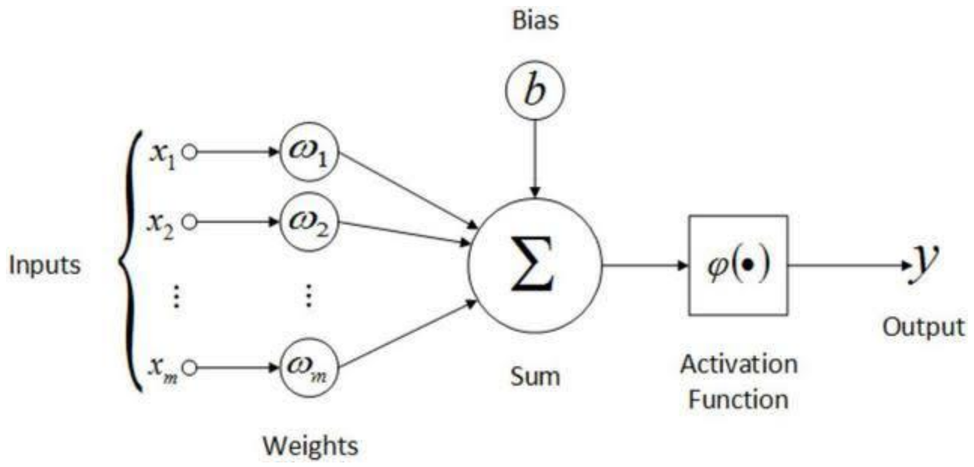
```
ac
```

>>

```
0.95
```

3. Write a Python program to load iris data set and apply a perceptron learning algorithm .

Artificial Neural Networks (ANNs) are the new trend for all data scientists. From classical machine learning techniques, it is now shifted towards deep learning. Neural networks mimic the human brain which passes information through neurons. Perceptron is the first neural network to be created. It was designed by Frank Rosenblatt in 1957. Perceptron is a single layer neural network. This is the only neural network without any hidden layer. Perceptron is used in supervised learning generally for binary classification.



The above picture is of a perceptron where inputs are acted upon by weights and summed to bias and lastly passes through an activation function to give the final output.

Python code to implement perceptron learning algorithm:

```
import numpy as np

from sklearn.datasets import load_iris

iris = load_iris()

iris.target_names
```

OUTPUT:

```
    array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

The classes 'versicolor' and 'virginica' are merged into one class. This means that only two classes are left. So we can differentiate with the classifier between

- `Iris setosa`
- `not Iris setosa, or in other words either 'viriginica' od 'versicolor'`

We accomplish this with the following command:

```
targets = (iris.target==0).astype(np.int8)
print(targets)
```

OUTPUT:

```
    [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
     1 1 1 1 1 1 1 1 1 1 1 1 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 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 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
     0 0]
```

We split the data into train_data and test_set:

```
from sklearn.model_selection import train_test_split
datasets = train_test_split(iris.data,
                            targets,
                            test_size=0.2)

train_data, test_data, train_labels, test_labels = datasets
```

`Now, we create a Perceptron instance and fit the training data:`

```
from sklearn.linear_model import Perceptron
p = Perceptron(random_state=42,
               max_iter=10,
               tol=0.001)
p.fit(train_data, train_labels)
```

OUTPUT:

```
    Perceptron(max_iter=10, random_state=42)
```

Now, we are ready for predictions and we will look at some randomly chosen random X values:

```
import random
```

```
sample = random.sample(range(len(train_data)), 10)
for i in sample:
    print(i, p.predict([train_data[i]]))
```

OUTPUT:

```
   102 [0]
   86 [0]
   89 [0]
   16 [0]
   108 [0]
   87 [1]
   98 [1]
   82 [0]
   39 [0]
   118 [0]
```

```
from sklearn.metrics import classification_report

print(classification_report(p.predict(train_data), train_labels))
```

OUTPUT:

```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00        79
               1       1.00      1.00      1.00        41

        accuracy                           1.00       120
       macro avg       1.00      1.00      1.00       120
    weighted avg       1.00      1.00      1.00       120
```

```
from sklearn.metrics import classification_report

print(classification_report(p.predict(test_data), test_labels))
```

OUTPUT:

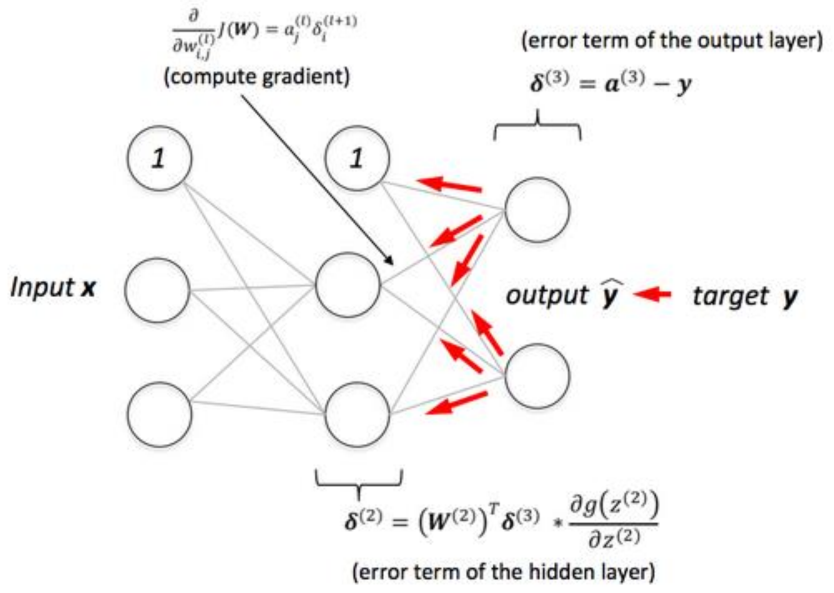
```
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00        21
               1       1.00      1.00      1.00         9

        accuracy                           1.00        30
       macro avg       1.00      1.00      1.00        30
    weighted avg       1.00      1.00      1.00        30
```

4. Implement the Backpropagation algorithm in Python to classify iris data set.

Backpropagation Neural Network (BPN) is used to improve the accuracy of neural network and make them capable of self-learning. Backpropagation means "backward propagation of errors". Here error is spread into the reverse direction in order to achieve better performance.

Backpropagation is an algorithm for supervised learning of artificial neural networks that uses the gradient descent method to minimize the cost function. It searches for optimal weights that optimize the mean-squared distance between the predicted and actual labels.

BPN was discovered by Rumelhart, Williams & Honton in 1986. The core concept of BPN is to backpropagate or spread the error from units of output layer to internal hidden layers in order to tune the weights to ensure lower error rates. It is considered a practice of fine-tuning the weights of neural networks in each iteration. Proper tuning of the weights will make a sure minimum loss and this will make a more robust, and generalizable trained neural network.


$\frac{\partial}{\partial w_{i,j}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

BPN learns in an iterative manner. In each iteration, it compares training examples with the actual target label. Target label can be a class label or continuous value. The backpropagation algorithm works in the following steps:

Initialize Network: BPN randomly initializes the weights.

Forward Propagate: After initialization, we will propagate into the forward direction. In this phase, we will compute the output and calculate the error from the target output.
Back Propagate Error: For each observation, weights are modified in order to reduce the error in a technique called the delta rule or gradient descent. It modifies weights in a “backward” direction to all the hidden layers.

Import Libraries:

```
#Import Libraries

import numpy as np

import pandas as pd

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt
```

Load Dataset:

```
# Load dataset

data = load_iris()

# Get features and target

X=data.data

y=data.target
```

Prepare Dataset:

```
# Get dummy variable

y = pd.get_dummies(y).values

y[:3]
```

>>

```
array([[1, 0, 0],

       [1, 0, 0],

       [1, 0, 0]], dtype=uint8)
```

Split train and test set:

```
#Split data into train and test data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=20,
random_state=4)
```

Initialize Hyper parameters and Weights:

```
# Initialize variables

learning_rate = 0.1

iterations = 5000

N = y_train.size

# number of input features

input_size = 4

# number of hidden layers neurons

hidden_size = 2

# number of neurons at the output layer

output_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])
```

Initialize the weights for hidden and output layers with random values.

```
# Initialize weights

np.random.seed(10)

# initializing weight for the hidden layer

W1 = np.random.normal(scale=0.5, size=(input_size, hidden_size))

# initializing weight for the output layer

W2 = np.random.normal(scale=0.5, size=(hidden_size , output_size))
```

Helper Functions:

Create helper functions such as sigmoid, mean_square_error, and accuracy.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def mean_squared_error(y_pred, y_true):
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)


def accuracy(y_pred, y_true):
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
```

Backpropagation Neural Network:

In this phase, we are creating BPN in three steps feedforward propagation, error calculation and backpropagation phase. To do this, we are creating a for loop for given number of iterations that execute the three steps (feedforward propagation, error calculation and backpropagation phase) and update the weights in each iteration.

```
for itr in range(iterations):
    # feedforward propagation
    # on hidden layer
    Z1 = np.dot(x_train, W1)
    A1 = sigmoid(Z1)
    # on output layer
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)


    # Calculating error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results=results.append({"mse":mse, "accuracy":acc},ignore_index=True
)
```

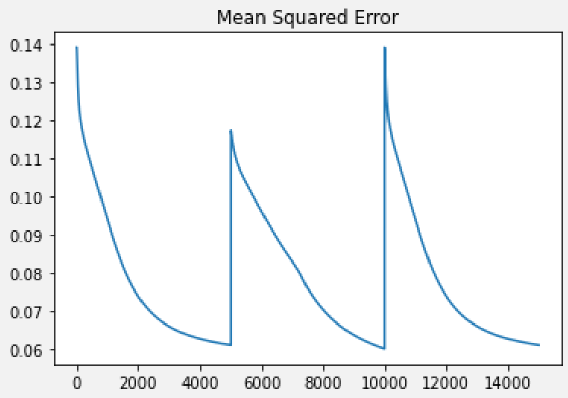
```
    # backpropagation

    E1 = A2 - y_train

    dW1 = E1 * A2 * (1 - A2)

    E2 = np.dot(dW1, W2.T)

    dW2 = E2 * A1 * (1 - A1)
```

```
    # weight updates

    W2_update = np.dot(A1.T, dW1) / N

    W1_update = np.dot(x_train.T, dW2) / N

    W2 = W2 - learning_rate * W2_update

    W1 = W1 - learning_rate * W1_update
```

Plot MSE and Accuracy:

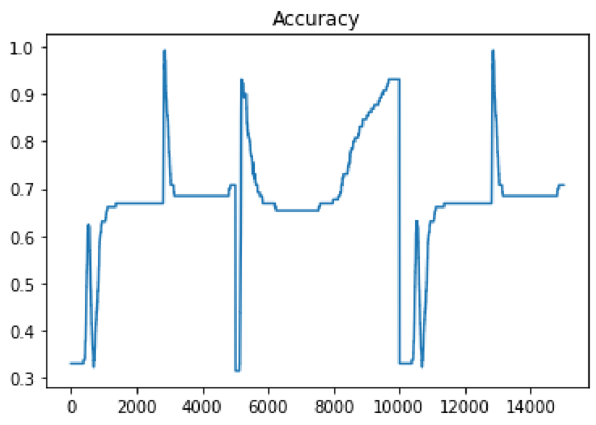
Let's plot mean squared error in each iteration using pandas plot() function.

```
results.mse.plot(title="Mean Squared Error")
```



Lets plot accuracy in each iteration using pandas plot() function.

```
results.accuracy.plot(title="Accuracy")
```



Predict for Test Data and Evaluate the Performance:

Let’s make prediction for the test data and assess the performance of Backpropagation neural network.

```
# feedforward

Z1 = np.dot(x_test, W1)

A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)

A2 = sigmoid(Z2)

acc = accuracy(A2, y_test)

print("Accuracy: {}".format(acc))

>>

Accuracy: 0.8
```