IMPLEMENT A LINEAR REGRESSION WITH A REAL DATASET

Aim:

The aim of building a regression model is to predict a continuous numerical outcome variable based on one or more input variables. There are several algorithms that can be used to build regression models, including linear regression, polynomial regression, decision trees, random forests, and neural networks.

Algorithm:

1. Collecting and cleaning the data: The first step in building a regression model is to gather the data needed for analysis and ensure that it is clean and consistent. This may involve removing missing values, outliers, and other errors.
2. Exploring the data: Once the data is cleaned, it is important to explore it to gain an understanding of the relationships between the input and outcome variables. This may involve calculating summary statistics, creating visualizations, and testing for correlations.
3. Choosing the algorithm: Based on the nature of the problem and the characteristics of the data, an appropriate regression algorithm is chosen.
4. Preprocessing the data: Before applying the regression algorithm, it may be necessary to preprocess the data to ensure that it is in a suitable format. This may involve standardizing or normalizing the data, encoding categorical variables, or applying feature engineering techniques.
5. Training the model: The regression model is trained on a subset of the data, using an optimization algorithm to find the values of the model parameters that minimize the difference between the predicted and actual values.
6. Evaluating the model: Once the model is trained, it is evaluated using a separate test dataset to determine its accuracy and generalization performance. Metrics such as mean squared error, R-squared, or root mean squared error can be used to assess the model's performance.
7. Improving the model: Based on the evaluation results, the model can be refined by adjusting the model parameters or using different algorithms.
8. Deploying the model: Finally, the model can be deployed to make predictions on new data.

Program:

```
# importing packages
import numpy as np # to perform calculations
import pandas as pd # to read data
import matplotlib.pyplot as plt # to visualise
#load dataset
# In read_csv() function, we have passed the location to where the file is located at dphi official github page
boston_data = pd.read_csv("https://raw.githubusercontent.com/dphi-official/Datasets/master/Boston_Housing/Training_set_boston.csv")
#view data
boston_data.head()
#separate input and output features
X = boston_data.drop('MEDV', axis = 1)    # Input Variables/features
y = boston_data.MEDV      # output variables/features
#splitting the data
# import train_test_split
from sklearn.model_selection import train_test_split

# Assign variables to capture train test split output
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# X_train: independent/input feature data for training the model
# y_train: dependent/output feature data for training the model
# X_test: independent/input feature data for testing the model; will be used to predict the output values
# y_test: original dependent/output values of X_test; We will compare this values with our predicted values to check the performance of our built model.
# test_size = 0.20: 20% of the data will go for test set and 70% of the data will go for train set
# random_state = 42: this will fix the split i.e. there will be same split for each time you run the code

# find the number of input features
n_features = X.shape[1]
print(n_features)

#training the model
from tensorflow.keras import Sequential    # import Sequential from tensorflow.keras
from tensorflow.keras.layers import Dense  # import Dense from tensorflow.keras.layes
from numpy.random import seed    # seed helps you to fix the randomness in the neural network.
import tensorflow

# define the model
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(n_features,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
```

```
#compile the model
# import RMSprop optimizer
from tensorflow.keras.optimizers import RMSprop
optimizer = RMSprop(0.01)    # 0.01 is the learning rate
model.compile(loss='mean_squared_error',optimizer=optimizer) # compile the model
#fitting the model
seed_value = 42
seed(seed_value)
# 1. Set `PYTHONHASHSEED` environment variable at a fixed value
import os
os.environ['PYTHONHASHSEED']=str(seed_value)
# 2. Set `python` built-in pseudo-random generator at a fixed value
import random
random.seed(seed_value)
# 3. Set `numpy` pseudo-random generator at a fixed value
import numpy as np
np.random.seed(seed_value)
# 4. Set the `tensorflow` pseudo-random generator at a fixed value
tensorflow.random.set_seed(seed_value)
model.fit(X_train, y_train, epochs=200, batch_size=30, verbose = 1)
#evaluate the model
model.evaluate(X_test, y_test)

#Hyperparameter Tunning
#learning rate
# define the model
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(n_features,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
optimizer = RMSprop(0.1)    # 0.1 is the learning rate
model.compile(loss='mean_squared_error',optimizer=optimizer)    # compile the model

# fit the model
model.fit(X_train, y_train, epochs=200, batch_size=30, verbose = 1)
# evaluate the model
print('The MSE value is: ', model.evaluate(X_test, y_test))
#epochs
# define the model
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(n_features,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
optimizer = RMSprop(0.01)    # 0.1 is the learning rate
model.compile(loss='mean_squared_error',optimizer=optimizer)    # compile the model
```

```
# fit the model
model.fit(X_train, y_train, epochs=10, batch_size=30, verbose = 1)
# evaluate the model
print('The MSE value is: ', model.evaluate(X_test, y_test))
#batch size
# define the model
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(n_features,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
optimizer = RMSprop(0.01)    # 0.1 is the learning rate
model.compile(loss='mean_squared_error',optimizer=optimizer)    # compile the model

# fit the model
model.fit(X_train, y_train, epochs=200, batch_size=40, verbose = 1)
# evaluate the model
print('The MSE value is: ', model.evaluate(X_test, y_test))
#hyperparameter tuning with keras
# Goal: tune the learning rate
# 0. Install and import all the packages needed
!pip install -q -U keras-tuner
import kerastuner as kt
# 1. Define the general architecture of the model through a creation user-defined function
def model_builder(hp):
  model = Sequential()
  model.add(Dense(10, activation='relu', input_shape=(n_features,)))
  model.add(Dense(8, activation='relu'))
  model.add(Dense(1))
  hp_learning_rate = hp.Choice('learning_rate', values = [1e-1, 1e-2, 1e-3, 1e-4]) # Tuning the lea
rning rate (four different values to test: 0.1, 0.01, 0.001, 0.0001)
  optimizer = RMSprop(learning_rate = hp_learning_rate)
  model.compile(loss='mse',metrics=['mse'], optimizer=optimizer)
  return model
# 2. Define the hyperparameters grid to be validated
tuner_rs = kt.RandomSearch(
            model_builder,              # Takes hyperparameters (hp) and returns a Model instance
            objective = 'mse',          # Name of model metric to minimize or maximize
            seed = 42,                  # Random seed for replication purposes
            max_trials = 5,             # Total number of trials (model configurations) to test at most.
Note that the oracle may interrupt the search before max_trial models have been tested.
            directory='random_search')    # Path to the working directory (relative).
# 3. Run the GridSearchCV process
tuner_rs.search(X_train, y_train, epochs=10, validation_split=0.2, verbose=1)

# 4.1. Print the summary results of the hyperparameter tuning procedure
tuner_rs.results_summary()

# 4.2. Print the results of the best model
```

```
best_model = tuner_rs.get_best_models(num_models=1)[0]
best_model.evaluate(X_test, y_test)

# 4.3. Print the best model's architecture
best_model.summary()

#prediction
# Load new test data
new_test_data = pd.read_csv('https://raw.githubusercontent.com/dphi-official/Datasets/master/Boston_Housing/Testing_set_boston.csv')

# make a prediction
best_model.predict(new_test_data)
```

Output:

```
Using Linear Regression
Training data accuracy 0.7574698746154127
Testing data accuracy 0.7576318049777959
```

Result:

Thus, the program for Linear regression models is executed successfully and output is verified.

IMPLEMENT A BINARY CLASSIFICATION MODEL

Aim:

To write a program of binary classification model using a California housing dataset.

Algorithm:

1. Select the feature that best splits the data: The first step is to select the feature that best separates the data into groups with different target values.
2. Recursively split the data: For each group created in step 1, repeat the process of selecting the best feature to split the data until a stopping criterion is met. The stopping criterion may be a maximum tree depth, a minimum number of samples in a leaf node, or another condition.
3. Assign a prediction value to each leaf node: Once the tree is built, assign a prediction value to each leaf node. This value may be the mean or median target value of the samples in the leaf node.

Random Forest

1. Randomly select a subset of features: Before building each decision tree, randomly select a subset of features to consider for splitting the data.
2. Build multiple decision trees: Build multiple decision trees using the process described above, each with a different subset of features.
3. Aggregate the predictions: When making predictions on new data, aggregate the predictions from all decision trees to obtain a final prediction value. This can be done by taking the average or majority vote of the predictions.

Program:

```
#@title Load the imports
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import layers
from matplotlib import pyplot as plt

# The following lines adjust the granularity of reporting.
pd.options.display.max_rows = 10
pd.options.display.float_format = "{:.1f}".format
# tf.keras.backend.set_floatx('float32')
print("Ran the import statements.")
train_df = pd.read_csv("https://download.mlcc.google.com/mledu-
datasets/california_housing_train.csv")
test_df = pd.read_csv("https://download.mlcc.google.com/mledu-
datasets/california_housing_test.csv")
train_df = train_df.reindex(np.random.permutation(train_df.index)) # shuffle the training set
# Calculate the Z-scores of each column in the training set and
```

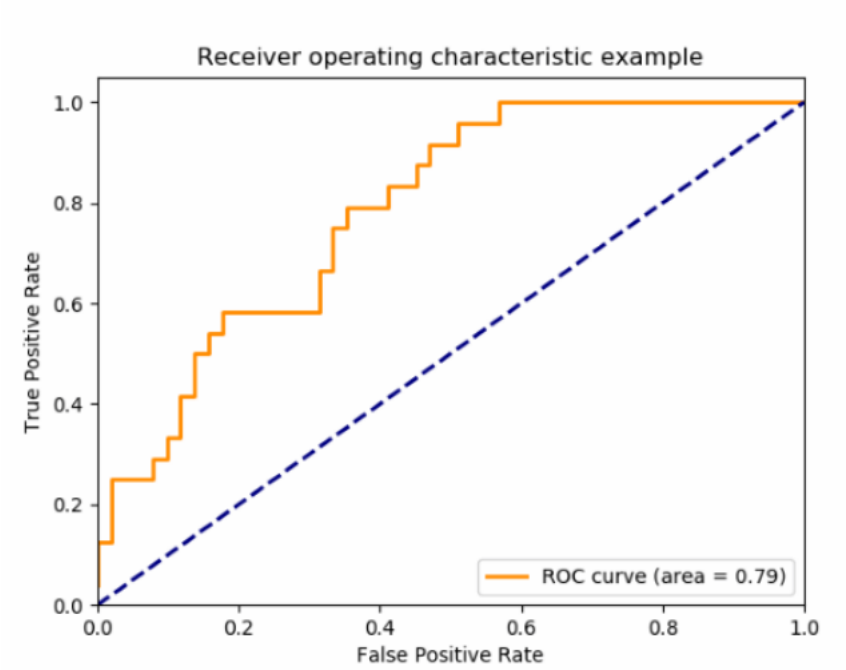
```
# write those Z-scores into a new pandas DataFrame named train_df_norm.
train_df_mean = train_df.mean()
train_df_std = train_df.std()
train_df_norm = (train_df - train_df_mean)/train_df_std
# Examine some of the values of the normalized training set. Notice that most
# Z-scores fall between -2 and +2.
train_df_norm.head()
# Calculate the Z-scores of each column in the test set and
# write those Z-scores into a new pandas DataFrame named test_df_norm.
test_df_mean = test_df.mean()
test_df_std  = test_df.std()
test_df_norm = (test_df - test_df_mean)/test_df_std
threshold = 265000 # This is the 75th percentile for median house values.
train_df_norm["median_house_value_is_high"] = ? Your code here
test_df_norm["median_house_value_is_high"] = ? Your code here
train_df_norm["median_house_value_is_high"].head(8000)
threshold = 265000
train_df_norm["median_house_value_is_high"] = (train_df["median_house_value"] >
threshold).astype(float)
test_df_norm["median_house_value_is_high"] = (test_df["median_house_value"] >
threshold).astype(float)
train_df_norm["median_house_value_is_high"].head(8000)

# threshold_in_Z = 1.0
# train_df_norm["median_house_value_is_high"] = (train_df_norm["median_house_value"] >
threshold_in_Z).astype(float)
# test_df_norm["median_house_value_is_high"] = (test_df_norm["median_house_value"] >
threshold_in_Z).astype(float)
inputs = {
# Features used to train the model on.
  'median_income': tf.keras.Input(shape=(1,)),
  'total_rooms': tf.keras.Input(shape=(1,))
}
#@title Define the functions that create and train a model.
def create_model(my_inputs, my_learning_rate, METRICS):
  # Use a Concatenate layer to concatenate the input layers into a single tensor.
  # as input for the Dense layer. Ex: [input_1[0][0], input_2[0][0]]
  concatenated_inputs = tf.keras.layers.Concatenate()(my_inputs.values())
  dense = layers.Dense(units=1, input_shape=(1,),name='dense_layer', activation=tf.sigmoid)
  dense_output = dense(concatenated_inputs)
  """Create and compile a simple classification model."""
  my_outputs = {
    'dense': dense_output,
  }
  model = tf.keras.Model(inputs=my_inputs, outputs=my_outputs)
```

```
# Call the compile method to construct the layers into a model that
  # TensorFlow can execute.  Notice that we're using a different loss
  # function for classification than for regression.
  model.compile(optimizer=tf.keras.optimizers.experimental.RMSprop(learning_rate=my_learni
ng_rate),
                loss=tf.keras.losses.BinaryCrossentropy(),
                metrics=METRICS)
  return model
def train_model(model, dataset, epochs, label_name,
                batch_size=None, shuffle=True):
  """Feed a dataset into the model in order to train it."""
  # The x parameter of tf.keras.Model.fit can be a list of arrays, where
  # each array contains the data for one feature.  Here, we're passing
  # every column in the dataset. Note that the feature_layer will filter
  # away most of those columns, leaving only the desired columns and their
  # representations as features.
  features = {name:np.array(value) for name, value in dataset.items()}
  label = np.array(features.pop(label_name))
  history = model.fit(x=features, y=label, batch_size=batch_size,
                      epochs=epochs, shuffle=shuffle)

  # The list of epochs is stored separately from the rest of history.
  epochs = history.epoch

  # Isolate the classification metric for each epoch.
  hist = pd.DataFrame(history.history)
  return epochs, hist
print("Defined the create_model and train_model functions.")
#@title Define the plotting function.
def plot_curve(epochs, hist, list_of_metrics):
  """Plot a curve of one or more classification metrics vs. epoch."""
  # list_of_metrics should be one of the names shown in:
  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Value")
  for m in list_of_metrics:
    x = hist[m]
    plt.plot(epochs[1:], x[1:], label=m)
  plt.legend()
print("Defined the plot_curve function.")
# The following variables are the hyperparameters.
learning_rate = 0.001
epochs = 20
batch_size = 100
label_name = "median_house_value_is_high"
```

```
classification_threshold = 0.35
# Establish the metrics the model will measure.
METRICS = [
          tf.keras.metrics.BinaryAccuracy(name='accuracy',
                                          threshold=classification_threshold),
         ]

# Establish the model's topography.
my_model = create_model(inputs, learning_rate, METRICS)

# To view a PNG of this model's layers, uncomment the call to
# `tf.keras.utils.plot_model` below. After running this code cell, click
# the file folder on the left, then the `my_classification_model.png` file.
# tf.keras.utils.plot_model(my_model, "my_classification_model.png")
# Train the model on the training set.
epochs, hist = train_model(my_model, train_df_norm, epochs,
                           label_name, batch_size)

# Plot a graph of the metric(s) vs. epochs.
list_of_metrics_to_plot = ['accuracy']

plot_curve(epochs, hist, list_of_metrics_to_plot)
features = {name:np.array(value) for name, value in test_df_norm.items()}
label = np.array(features.pop(label_name))

my_model.evaluate(x = features, y = label, batch_size=batch_size)
# The following variables are the hyperparameters.
learning_rate = 0.001
epochs = 20
batch_size = 100
classification_threshold = 0.35
label_name = "median_house_value_is_high"

# Modify the following definition of METRICS to generate
# not only accuracy and precision, but also recall:
METRICS = [
     tf.keras.metrics.BinaryAccuracy(name='accuracy',
                                     threshold=classification_threshold),
     tf.keras.metrics.Precision(thresholds=classification_threshold,
                                name='precision'
                                ),
     ? # write code here ]

# Establish the model's topography.
my_model = create_model(inputs, learning_rate, METRICS)
```

```
# Train the model on the training set.
epochs, hist = train_model(my_model, train_df_norm, epochs,
                           label_name, batch_size)

# Plot metrics vs. epochs
list_of_metrics_to_plot = ['accuracy', 'precision', 'recall']
plot_curve(epochs, hist, list_of_metrics_to_plot)
features = {name:np.array(value) for name, value in test_df_norm.items()}
label = np.array(features.pop(label_name))

my_model.evaluate(x = features, y = label, batch_size=batch_size)
#@title Double-click to view the solution for Task 5.

# The following variables are the hyperparameters.
learning_rate = 0.001
epochs = 20
batch_size = 100
label_name = "median_house_value_is_high"

# AUC is a reasonable "summary" metric for
# classification models.
# Here is the updated definition of METRICS to
# measure AUC:
METRICS = [
    tf.keras.metrics.AUC(num_thresholds=100, name='auc'),
]

# Establish the model's topography.
my_model = create_model(inputs, learning_rate, METRICS)

# Train the model on the training set.
epochs, hist = train_model(my_model, train_df_norm, epochs,
                           label_name, batch_size)

# Plot metrics vs. epochs
list_of_metrics_to_plot = ['auc']
plot_curve(epochs, hist, list_of_metrics_to_plot)
```

Output:


Receiver operating characteristic example
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.79)

Result:

Thus, the program to implement binary classification model using California housing dataset was executed successfully.

IMPLEMENT CLASSIFICATION WITH NEAREST NEIGHBORS

Aim:

To implement a program for Nearest Neighbours classification using real dataset.

Algorithm:

Step-1: Select the number K of the neighbors

Step-2: Calculate the Euclidean distance of **K number of neighbors**

Step-3: Take the K nearest neighbors as per the calculated Euclidean distance.

Step-4: Among these k neighbors, count the number of the data points in each category.

Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.

Step-6: model is ready.

Program:

```
#Load the necessary python libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.style.use('ggplot')
#Load the dataset
df = pd.read_csv('../input/diabetes.csv')

#Print the first 5 rows of the dataframe.
df.head()\
#Let's observe the shape of the dataframe.
df.shape
#Let's create numpy arrays for features and target
X = df.drop('Outcome',axis=1).values
y = df['Outcome'].values
#importing train_test_split
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.4,random_state=42, stratify=y)
#import KNeighborsClassifier
from sklearn.neighbors import KNeighborsClassifier

#Setup arrays to store training and test accuracies
```

```
neighbors =  np.arange(1,9)
train_accuracy =np.empty(len(neighbors))
test_accuracy = np.empty(len(neighbors))

for i,k in enumerate(neighbors):
    #Setup a knn classifier with k neighbors
    knn = KNeighborsClassifier(n_neighbors=k)

    #Fit the model
    knn.fit(X_train, y_train)

    #Compute accuracy on the training set
    train_accuracy[i] = knn.score(X_train, y_train)

    #Compute accuracy on the test set
    test_accuracy[i] = knn.score(X_test, y_test)
#Generate plot
plt.title('k-NN Varying number of neighbors')
plt.plot(neighbors, test_accuracy, label='Testing Accuracy')
plt.plot(neighbors, train_accuracy, label='Training accuracy')
plt.legend()
plt.xlabel('Number of neighbors')
plt.ylabel('Accuracy')
plt.show()
#Setup a knn classifier with k neighbors
knn = KNeighborsClassifier(n_neighbors=7)

#Fit the model
knn.fit(X_train,y_train)

Out[10]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=7, p=2,
           weights='uniform')
#Get accuracy. Note: In case of classification algorithms score method represents accuracy.
knn.score(X_test,y_test)

#import confusion_matrix
from sklearn.metrics import confusion_matrix

#let us get the predictions using the classifier we had fit above
y_pred = knn.predict(X_test)

confusion_matrix(y_test,y_pred)
pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True)
#import classification_report
from sklearn.metrics import classification_report

print(classification_report(y_test,y_pred))
```

OUTPUT:

Confusion matrix

array([[165, 36],
[47, 60]])

	precision	recall	f1-score	support
0	0.78	0.82	0.80	201
1	0.62	0.56	0.59	107
avg / total	0.73	0.73	0.73	308

Result:

Thus, the program to implement nearest neighbor classifier using real dataset was executed successfully.

IMPLEMENT TRAINING AND TESTING DETERMINE WHETHER OVERFITTING OR NOT USING VALIDATION AND TEST DATASET

Aim:

To determine whether the trained model is overfitting or not using test data.

Program:

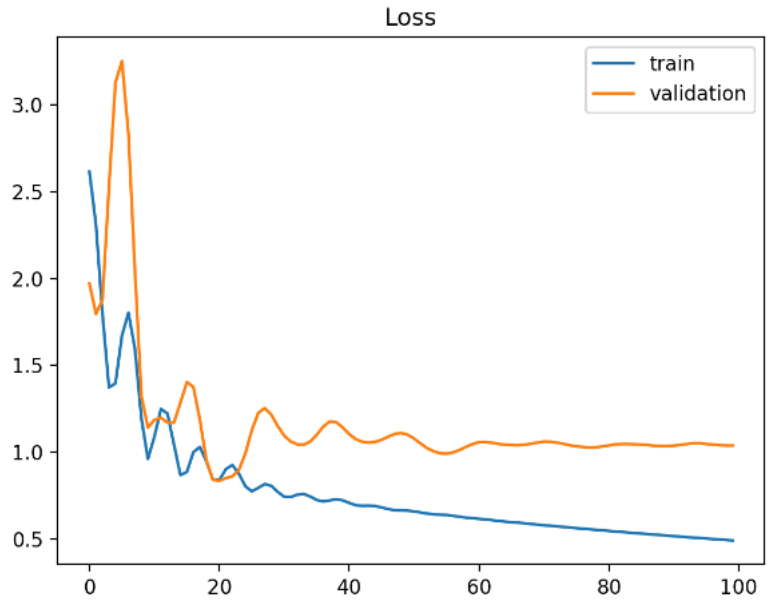
```
#@title Run on TensorFlow 2.x
%tensorflow_version 2.x
#@title Import modules
import numpy as np
import pandas as pd
import tensorflow as tf
from matplotlib import pyplot as plt

pd.options.display.max_rows = 10
pd.options.display.float_format = "{:.1f}".format
In [0]:
train_df = pd.read_csv("https://download.mlcc.google.com/mledu-
datasets/california_housing_train.csv")
test_df = pd.read_csv("https://download.mlcc.google.com/mledu-
datasets/california_housing_test.csv")
In [0]:
scale_factor = 1000.0

# Scale the training set's label.
train_df["median_house_value"] /= scale_factor

# Scale the test set's label
test_df["median_house_value"] /= scale_factor
In [0]:
#@title Define the functions that build and train a model
def build_model(my_learning_rate):
  """Create and compile a simple linear regression model."""
  # Most simple tf.keras models are sequential.
  model = tf.keras.models.Sequential()

  # Add one linear layer to the model to yield a simple linear regressor.
  model.add(tf.keras.layers.Dense(units=1, input_shape=(1,)))

  # Compile the model topography into code that TensorFlow can efficiently
  # execute. Configure training to minimize the model's mean squared error.
  model.compile(optimizer=tf.keras.optimizers.RMSprop(lr=my_learning_rate),
                loss="mean_squared_error",
                metrics=[tf.keras.metrics.RootMeanSquaredError()])
```

```
  return model

def train_model(model, df, feature, label, my_epochs,
                my_batch_size=None, my_validation_split=0.1):
  """Feed a dataset into the model in order to train it."""

  history = model.fit(x=df[feature],
                      y=df[label],
                      batch_size=my_batch_size,
                      epochs=my_epochs,
                      validation_split=my_validation_split)

  # Gather the model's trained weight and bias.
  trained_weight = model.get_weights()[0]
  trained_bias = model.get_weights()[1]

  # The list of epochs is stored separately from the
  # rest of history.
  epochs = history.epoch

  # Isolate the root mean squared error for each epoch.
  hist = pd.DataFrame(history.history)
  rmse = hist["root_mean_squared_error"]

  return epochs, rmse, history.history

print("Defined the build_model and train_model functions.")
In [0]:
#@title Define the plotting function

def plot_the_loss_curve(epochs, mae_training, mae_validation):
  """Plot a curve of loss vs. epoch."""

  plt.figure()
  plt.xlabel("Epoch")
  plt.ylabel("Root Mean Squared Error")

  plt.plot(epochs[1:], mae_training[1:], label="Training Loss")
  plt.plot(epochs[1:], mae_validation[1:], label="Validation Loss")
  plt.legend()

  # We're not going to plot the first epoch, since the loss on the first epoch
  # is often substantially greater than the loss for other epochs.
  merged_mae_lists = mae_training[1:] + mae_validation[1:]
  highest_loss = max(merged_mae_lists)
  lowest_loss = min(merged_mae_lists)
```

```
  delta = highest_loss - lowest_loss
  print(delta)

  top_of_y_axis = highest_loss + (delta * 0.05)
  bottom_of_y_axis = lowest_loss - (delta * 0.05)

  plt.ylim([bottom_of_y_axis, top_of_y_axis])
  plt.show()

print("Defined the plot_the_loss_curve function.")
# The following variables are the hyperparameters.
learning_rate = 0.08
epochs = 30
batch_size = 100

# Split the original training set into a reduced training set and a
# validation set.
validation_split=0.2

# Identify the feature and the label.
my_feature="median_income" # the median income on a specific city block.
my_label="median_house_value" # the median value of a house on a specific city block.
# That is, you're going to create a model that predicts house value based
# solely on the neighborhood's median income.

# Discard any pre-existing version of the model.
my_model = None

# Invoke the functions to build and train the model.
my_model = build_model(learning_rate)
epochs, rmse, history = train_model(my_model, train_df, my_feature,
                                    my_label, epochs, batch_size,
                                    validation_split)

plot_the_loss_curve(epochs, history["root_mean_squared_error"],
                    history["val_root_mean_squared_error"])
shuffled_train_df = train_df.reindex(np.random.permutation(train_df.index))

epochs, rmse, history = train_model(my_model, shuffled_train_df, my_feature, my_label,
epochs, batch_size, validation_split)

#@title Double-click to view the complete implementation.

# The following variables are the hyperparameters.
learning_rate = 0.08
epochs = 70
batch_size = 100
```

```
# Split the original training set into a reduced training set and a
# validation set.
validation_split=0.2

# Identify the feature and the label.
my_feature="median_income" # the median income on a specific city block.
my_label="median_house_value" # the median value of a house on a specific city block.
# That is, you're going to create a model that predicts house value based
# solely on the neighborhood's median income.

# Discard any pre-existing version of the model.
my_model = None

# Shuffle the examples.
shuffled_train_df = train_df.reindex(np.random.permutation(train_df.index))

# Invoke the functions to build and train the model. Train on the shuffled
# training set.
my_model = build_model(learning_rate)
epochs, rmse, history = train_model(my_model, shuffled_train_df, my_feature,
                            my_label, epochs, batch_size,
                            validation_split)

plot_the_loss_curve(epochs, history["root_mean_squared_error"],
                history["val_root_mean_squared_error"])
In [0]:
x_test = test_df[my_feature]
y_test = test_df[my_label]

results = my_model.evaluate(x_test, y_test, batch_size=batch_size)
```

Output:



Result:

Thus, the program to determine the trained model is overfitted or not by testing using valid and test data is implemented successfully.

IMPLEMENT THE K-MEANS ALGORITHM USING CODON DATASET

Aim:

To implement a program for k means algorithm using codon dataset

Algorithm:

Step-1: Select the number K to decide the number of clusters.

Step-2: Select random K points or centroids. (It can be other from the input dataset).

Step-3: Assign each data point to their closest centroid, which will form the predefined K clusters.

Step-4: Calculate the variance and place a new centroid of each cluster.

Step-5: Repeat the third steps, which means reassign each datapoint to the new closest centroid of each cluster.

Step-6: If any reassignment occurs, then go to step-4 else go to FINISH.

Step-7: The model is ready.

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
from sklearn.cluster import KMeans
from yellowbrick.cluster import KElbowVisualizer
from sklearn import metrics
dataset = pd.read_csv('../input/seed-from-uci/Seed_Data.csv')
dataset.head()
dataset.describe(include = "all")
features = dataset.iloc[:, 0:7]
target = dataset.iloc[:, -1]

model = KMeans()
visualizer = KElbowVisualizer(model, k=(1,10))

visualizer.fit(features)    # Fit the data to the visualizer
visualizer.poof()    # Draw/show/poof the data
kmeans = KMeans(n_clusters=3)
kmeans.fit(features)
```

```
cluster_labels = kmeans.fit_predict(features)

kmeans.cluster_centers_
silhouette_avg = metrics.silhouette_score(features, cluster_labels)
print ('silhouette coefficient for the above clutering = ', silhouette_avg)
def purity_score(y_true, y_pred):
    # compute contingency matrix (also called confusion matrix)
    contingency_matrix = metrics.cluster.contingency_matrix(y_true, y_pred)
    return np.sum(np.amax(contingency_matrix, axis=0)) / np.sum(contingency_matrix)

purity = purity_score(target, cluster_labels)
print ('Purity for the above clutering = ', purity)
!pip install pyclustering

from pyclustering.cluster.kmedoids import kmedoids
# Randomly pick 3 indexs from the original sample as the mediods
initial_medoids = [1, 50, 170]

# Create instance of K-Medoids algorithm with prepared centers.
kmedoids_instance = kmedoids(features.values.tolist(), initial_medoids)

# Run cluster analysis.
kmedoids_instance.process()

# predict function is not availble in the release branch yet.
# cluster_labels = kmedoids_instance.predict(features.values)

clusters = kmedoids_instance.get_clusters()

# Prepare cluster labels
cluster_labels = np.zeros([210], dtype=int)
for x in np.nditer(np.asarray(clusters[1])):
  cluster_labels[x] = 1
for x in np.nditer(np.asarray(clusters[2])):
  cluster_labels[x] = 2

cluster_labels
# Mediods found in above clustering, indexes are shown below.
kmedoids_instance.get_medoids()
```

Output:

```
array([[11.96441558, 13.27480519,  0.8522    ,  5.22928571,  2.87292208,
         4.75974026, 5.08851948],
       [18.72180328, 16.29737705,  0.88508689,  6.20893443,  3.72267213,
         3.60359016, 6.06609836],
       [14.64847222, 14.46041667,  0.87916667,  5.56377778,  3.27790278,
         2.64893333,  5.19231944]])


[118, 48, 144]
```

Result:

Thus, the program to implement k-means algorithm using codon dataset has been executed successfully.

IMPLEMENT THE NAÏVE BAYES CLASSIFIER USING GAIT CLASSIFICATION DATASET

Aim:

The aim of the Naïve Bayes algorithm is to classify a given set of data points into different classes based on the probability of each data point belonging to a particular class. This algorithm is based on the Bayes theorem, which states that the probability of an event occurring given the prior knowledge of another event can be calculated using conditional probability.

Algorithm:

1. Collect the dataset: The first step in using Naïve Bayes is to collect a dataset that contains a set of data points and their corresponding classes.
2. Prepare the data: The next step is to preprocess the data and prepare it for the Naïve Bayes algorithm. This involves removing any unnecessary features or attributes and normalizing the data.
3. Compute the prior probabilities: The prior probabilities of each class can be computed by calculating the number of data points belonging to each class and dividing it by the total number of data points.
4. Compute the likelihoods: The likelihoods of each feature for each class can be computed by calculating the conditional probability of the feature given the class. This involves counting the number of data points in each class that have the feature and dividing it by the total number of data points in that class.
5. Compute the posterior probabilities: The posterior probabilities of each class can be computed by multiplying the prior probability of the class with the product of the likelihoods of each feature for that class.
6. Make predictions: Once the posterior probabilities have been computed for each class, the Naïve Bayes algorithm can be used to make predictions by selecting the class with the highest probability.
7. Evaluate the model: The final step is to evaluate the performance of the Naïve Bayes model. This can be done by computing various performance metrics such as accuracy, precision, recall, and F1 score.

Program:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt # for data visualization purposes
import seaborn as sns # for statistical data visualization
%matplotlib inline

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the i
nput directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

```
/kaggle/input/adult-dataset/adult.csv
```

In [2]:

linkcode

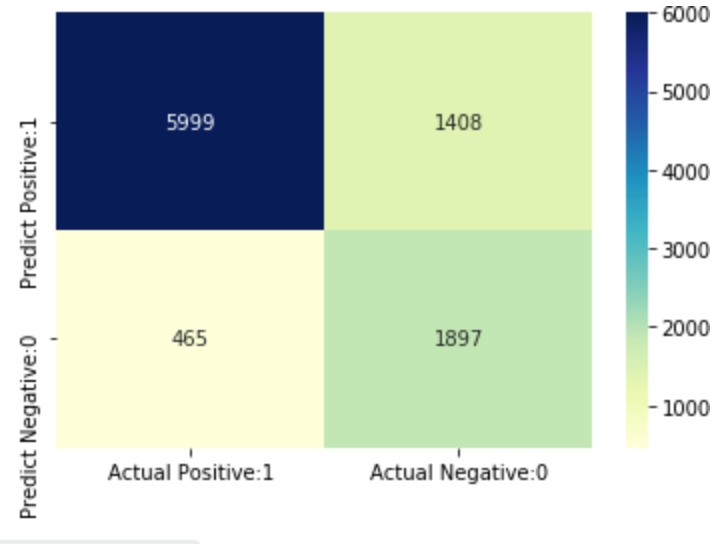
```
import warnings

warnings.filterwarnings('ignore')

data = '/kaggle/input/adult-dataset/adult.csv'

df = pd.read_csv(data, header=None, sep=',\s')
# view dimensions of dataset

df.shape
# preview the dataset

df.head()
col_names = ['age', 'workclass', 'fnlwgt', 'education', 'education_num', 'marital_status', 'occupatio
n', 'relationship',
            'race', 'sex', 'capital_gain', 'capital_loss', 'hours_per_week', 'native_country', 'income']

df.columns = col_names

df.columns
Index(['age', 'workclass', 'fnlwgt', 'education', 'education_num',
       'marital_status', 'occupation', 'relationship', 'race', 'sex',
       'capital_gain', 'capital_loss', 'hours_per_week', 'native_country',
       'income'],
      dtype='object')
# find categorical variables
```

```
categorical = [var for var in df.columns if df[var].dtype=='O']

print('There are {} categorical variables\n'.format(len(categorical)))

print('The categorical variables are :\n\n', categorical)
X = df.drop(['income'], axis=1)

y = df['income']
# split X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
# check the shape of X_train and X_test

X_train.shape, X_test.shape
# train a Gaussian Naive Bayes classifier on the training set
from sklearn.naive_bayes import GaussianNB


# instantiate the model
gnb = GaussianNB()


# fit the model
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)

y_pred
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_pred)))
y_pred_train = gnb.predict(X_train)

y_pred_trai
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_train, y_pred_train)))

print('Training set score: {:.4f}'.format(gnb.score(X_train, y_train)))

print('Test set score: {:.4f}'.format(gnb.score(X_test, y_test)))
y_test.value_counts()
null_accuracy = (7407/(7407+2362))

print('Null accuracy score: {0:0.4f}'. format(null_accuracy))
from sklearn.metrics import confusion_matrix
```

```
cm = confusion_matrix(y_test, y_pred)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])

cm_matrix = pd.DataFrame(data=cm, columns=['Actual Positive:1', 'Actual Negative:0'],
                         index=['Predict Positive:1', 'Predict Negative:0'])

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')

from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
classification_accuracy = (TP + TN) / float(TP + TN + FP + FN)

print('Classification accuracy : {0:0.4f}'.format(classification_accuracy))
```

Output:

```
accuracy score: 0.7582
Confusion matrix

 [[5999 1408]
 [ 465 1897]]

True Positives(TP) = 5999

True Negatives(TN) = 1897

False Positives(FP) =  1408

False Negatives(FN) =  465
<matplotlib.axes._subplots.AxesSubplot at 0x7fd6899b6a58>
```



	precision	recall	f1-score	support
<=50K	0.93	0.81	0.86	7407
>50K	0.57	0.80	0.67	2362
accuracy			0.81	9769
macro avg	0.75	0.81	0.77	9769
weighted avg	0.84	0.81	0.82	9769

Classification accuracy : 0.8083

Result:

Thus, the program to implement naïve bayes classification using Gait dataset has been executed successfully.