```
1.import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
import sys
2. df=pd.read_csv("creditcard.csv")
Df
3. df.head()
4. df.tail()
5. df.isnull().any()
6. df['Class'].value_counts()
7. frauds = df[df.Class == 1]
normal = df[df.Class == 0]
frauds.shape
8. normal.shape
9. frauds.Amount.describe()
10. normal.Amount.describe()
11. f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
f.suptitle('Amount per transaction by class')

bins = 50

ax1.hist(frauds.Amount, bins = bins)
ax1.set_title('Fraud')

ax2.hist(normal.Amount, bins = bins)
ax2.set_title('Normal')

plt.xlabel('Amount ($)')
plt.ylabel('Number of Transactions')
```

```
plt.xlim((0, 20000))
plt.yscale('log')
plt.show();
```

12. f, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

```
f.suptitle('Time of transaction vs Amount by class')

ax1.scatter(frauds.Time, frauds.Amount)
ax1.set_title('Fraud')

ax2.scatter(normal.Time, normal.Amount)
ax2.set_title('Normal')

plt.xlabel('Time (in Seconds)')
plt.ylabel('Amount')
plt.show()
```

13. %matplotlib inline

```
import matplotlib.gridspec as gridspec
import seaborn as sns
from scipy.stats import norm, multivariate_normal

plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12



import random
random.seed(0)

import warnings
warnings.filterwarnings('ignore')
```

14. #fraud vs. normal transactions

counts = df.Class.value_counts()

normal = counts[0]

fraudulent = counts[1]

perc_normal = (normal/(normal+fraudulent))*100

perc_fraudulent = (fraudulent/(normal+fraudulent))*100

print('There were {} non-fraudulent transactions ({:.3f}%) and {} fraudulent transactions ({:.3f}%).'.format(normal, perc_normal, fraudulent, perc_fraudulent))

15. plt.figure(figsize=(8,6))

sns.barplot(x=counts.index, y=counts)

plt.title('Count of Fraudulent vs. Non-Fraudulent Transactions')

plt.ylabel('Count')

plt.xlabel('Class (0:Non-Fraudulent, 1:Fraudulent)')

16. #numerical summary -> only non-anonymized columns of interest

pd.set_option('precision', 3)

df.loc[:, ['Time', 'Amount']].describe()

17. #visualizations of time and amount

plt.figure(figsize=(10,8))

plt.title('Distribution of Time Feature')

sns.distplot(df.Time)

18. from sklearn.model_selection import train_test_split

from sklearn import preprocessing

19. # Creating Train Set, Dev Set & Train set

Converting the csv data into matrix

columns = "Time V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20 V21 V22 V23 V24 V25 V26 V27 V28 Amount".split()

X = pd.DataFrame.as_matrix(df,columns=columns)

Y = df.Class

Y=Y.values.reshape(Y.shape[0],1)

X.shape

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.06)

X_test, X_dev, Y_test, Y_dev = train_test_split(X_test,Y_test, test_size=0.5)

20. # Check if there is Classification Values - 0/1 in training set and other set

```
np.where(Y_train == 1)

np.where(Y_test == 1)

np.where(Y_dev == 1)
```

21. # Checking the shape's of the new data set as matrix

```
print("No of training Examples : "+str(X_train.shape[0]))  # 94% data

print("No of test Examples : "+str(X_test.shape[0]))       # 3% data

print("No of dev Examples : "+str(X_dev.shape[0]))         # 3% data

print("Shape of training data : "+str(X_train.shape))

print("Shape of test data : "+str(X_test.shape))

print("Shape of dev data : "+str(X_dev.shape))

print("Shape of Y test data : "+str(Y_test.shape))

print("Shape of Y dev data : "+str(Y_dev.shape))
```

22. #Flatten the data to so that all Features/X Variables

```
X_train_flatten = X_train.reshape(X_train.shape[0],-1).T

Y_train_flatten = Y_train.reshape(Y_train.shape[0],-1).T

X_dev_flatten = X_dev.reshape(X_dev.shape[0],-1).T

Y_dev_flatten = Y_dev.reshape(Y_dev.shape[0],-1).T

X_test_flatten = X_test.reshape(X_test.shape[0],-1).T

Y_test_flatten = Y_test.reshape(Y_test.shape[0],-1).T


print("No of training Examples : "+str(X_train_flatten.shape))

print("No of test Examples : "+str(Y_train_flatten.shape))

print("No of X_dev Examples : "+str(X_dev_flatten.shape))

print("No of Y_dev test Examples : "+str(Y_dev_flatten.shape))

print("No of X_test Examples : "+str(X_test_flatten.shape))

print("No of Y_test Examples : "+str(Y_test_flatten.shape))

print("No of Sanity_test : "+str(X_train_flatten[0:5,0]))
```

23. # Normalize features and create final Train set

```
X_train_set = preprocessing.normalize(X_train_flatten)

Y_train_set = Y_train_flatten
```

```
print("No of X_train_set shape : "+str(X_train_set.shape))
print("No of Y_train_set shape : "+str(Y_train_set.shape))
```

24. # Funcation to intialize weights for forward propogration

```
def intialize_parameters(layer_dims):
    parameters = {}
    L = len(layer_dims)
    for l in range(1,L):
        parameters['W'+str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])*0.01
        parameters['b'+str(l)] = np.zeros((layer_dims[l],1))

    return parameters
```

25. # Testing if the function works

```
parameters = intialize_parameters([30,20,10,5,2])
print("W1 =" + str(parameters["W1"]))
print("b1 =" + str(parameters["b1"]))
print("W2 =" + str(parameters["W2"]))
print("b2 =" + str(parameters["b2"]))
print("W3 =" + str(parameters["W3"]))
print("b3 =" + str(parameters["b3"]))
print("W4 =" + str(parameters["W4"]))
print("b4 =" + str(parameters["b4"]))
```

26. # create the sigmoid function

```
def sigmoid(z):

    s = 1/(1+np.exp(-z))
    cache = z
    return s,cache
```

27. # test sigmoid function

```
sigmoid(np.array(([2,7])))
```

28. # create the relu function

```
def relu(z):
```

```
    r = np.maximum(0,z)
    cache = z
    return r,cache
```

29. # testing relu function

```
relu([1,-1,21])
```

30. # Relu Backward and Sigmoid Backward

```
def relu_backward(dA, cache):

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ
```

31. # Linear_forward

```
def linear_forward(A, W, b):

    Z = np.dot(W,A)+b
    cache = (A, W, b)

    return Z, cache
```

32. #linear_activation_forward

```
def linear_activation_forward(A_prev, W, b, activation):

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev,W,b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

33. # L layers forward propagation

```
def forward_propagation(X, parameters):
    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in the neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
```

```
        A, cache = linear_activation_forward(A,parameters["W" + str(l)],parameters["b" +
str(l)],activation="relu")
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A,parameters["W" + str(L)],parameters["b" +
str(L)],activation="sigmoid")
    caches.append(cache)

    return AL, caches
```

34. # Cost function

```
def cost_function(AL, Y):
    m = Y.shape[1]

    cost = (-1/m)*np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns
[[17]] into 17).

    return cost
```

35. # linear_backward

```
def linear_backward(dZ, cache):
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = (1/m)*np.dot(dZ,A_prev.T)
    db = (1/m)*np.sum(dZ,axis=1,keepdims=True)
    dA_prev = np.dot(W.T,dZ)

    return dA_prev, dW, db
```

36. # linear_activation_backward

```
def linear_activation_backward(dA, cache, activation):

    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA,activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA,activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

37. # backward propagation

```
def backward_propagation(AL, Y, caches):

    grads = {}
    L = len(caches) # the number of layers
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outputs: "grads["dAL"], 
grads["dWL"], grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = 
linear_activation_backward(dAL,current_cache,activation="sigmoid")
```

```
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 2)], caches". Outputs: "grads["dA" + str(l + 1)] , grads["dW" + str(l
+ 1)] , grads["db" + str(l + 1)]
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp =
linear_activation_backward(grads["dA"+str(l+2)],current_cache,activation="relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

38. # update parameters

```
def update_parameters(parameters, grads, learning_rate):

    L = len(parameters) // 2 # number of layers in the neural network
    for l in range(1,L+1):
        parameters["W"+str(l)]=parameters["W" + str(l)]-learning_rate*grads["dW" + str(l)]
        parameters["b"+str(l)]=parameters["b" + str(l)]-learning_rate*grads["db" + str(l)]
    return parameters
```

39. # setting the size of the network

```
layer_dims = [30,20,10,5,1] #5 Layer model with 3 hidden layers

# Deep Learning network to classify frauds and normal
layer_dims = [30,20,10,5,1] #5 Layer model with 3 hidden layers

# Deep Learning network to classify frauds and normal
def nn_model(X,Y,layer_dims,learning_rate=.0065, num_iterations=2500,print_cost=False):
    costs = []

    #initialize parameters
```

```
    parameters = intialize_parameters(layer_dims)
    # for loop for iterations/epoch
    for i in range(0,num_iterations):
        #forward_propagation
        AL, caches = forward_propagation(X, parameters)

        #compute cost
        cost = cost_function(AL, Y)

        #backward_propagation
        grads = backward_propagation(AL, Y, caches)
        #update parameters
        parameters = update_parameters(parameters,grads,learning_rate)

        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

40. X_train_set.shape

Y_train_set.shape

41. # running a model

parameters = nn_model(X_train_set,Y_train_set,layer_dims,learning_rate=.0065,num_iterations = 2500, print_cost = True)

42. # predict Function

```
def predict(X, y, parameters):


    m = X.shape[1]

    p = np.zeros((1,m))


    # Forward propagation

    probas, caches = forward_propagation(X, parameters)



    # convert probas to 0/1 predictions

    for i in range(0, probas.shape[1]):

        if probas[0,i] > 0.5:

            p[0,i] = 1

        else:

            p[0,i] = 0


    #print results

    #print ("predictions: " + str(p))

    #print ("true labels: " + str(y))

    print("Accuracy: "  + str(np.sum((p == y)/m)))


    return p
```

43. pred_train = predict(X_train_set, Y_train_set, parameters)

44. pred_test = predict(X_test_flatten, Y_test_flatten, parameters)

45. pred_dev = predict(X_dev_flatten, Y_dev_flatten, parameters)

46. print('X_shapes:\n', 'X_train:', 'X_validation:\n', X_train.shape, X_validation.shape, '\n')

print('Y_shapes:\n', 'Y_train:', 'Y_validation:\n', y_train.shape, y_validation.shape)

47. from sklearn.model_selection import KFold

```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from xgboost import XGBClassifier
from sklearn.ensemble import RandomForestClassifier
```

48. ##Spot-Checking Algorithms

```
models = []

models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('SVM', SVC()))
models.append(('XGB', XGBClassifier()))
models.append(('RF', RandomForestClassifier()))

#testing models

results = []
names = []

for name, model in models:
   kfold = KFold(n_splits=10, random_state=42)
   cv_results = cross_val_score(model, X_train, y_train, cv=kfold, scoring='roc_auc')
```

```
    results.append(cv_results)
    names.append(name)
    msg = '%s: %f (%f)' % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

49. #Compare Algorithms

```
fig = plt.figure(figsize=(12,10))
plt.title('Comparison of Classification Algorithms')
plt.xlabel('Algorithm')
plt.ylabel('ROC-AUC Score')
plt.boxplot(results)
ax = fig.add_subplot(111)
ax.set_xticklabels(names)
plt.show()
```