FINANCIAL FRAUD DETECTION AND IMPLEMENTATION OF LVQ ALGORITHM

INTRODUCTION

Credit card fraud happens when someone (mostly a fraudster) uses the stolen credit card or the credit card information to make unauthorized purchases. The reason I chose this topic is that this problem is real and so many people are getting scammed because of credit card fraudulent activities. I am going to use Synthetic Financial Dataset for Fraud Detection.

Link for the dataset: https://www.kaggle.com/datasets/ealaxi/paysim1.

The dataset is very large with around 6 million entries and 11 columns. Due to computational limitations, I would be making use of 100,000 entries of the dataset.

The second phase of this project involves developing Learning Vector Quantization Algorithm (LVQ) to classify fraudulent and non-fraudulent activities. LVQ algorithm works on the principle of distance between weight matrix and input vector. As we are concerned with the distance in LVQ, we do not need to scale the data for this algorithm.

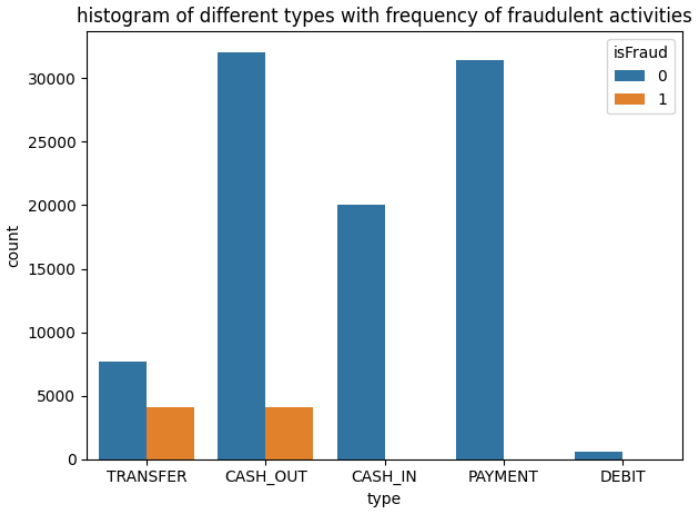
- **UNDERSTANDING THE DATASET**
  - **NUMERIC VARIABLES**
    - Amount – amount involved in the credit card transaction.
    - Oldbalanceorig – initial balance before the transaction.
    - Newbalanceorig – new balance after the transaction.
    - oldbalanceDest – initial balance recipient before the transaction.
    - newbalanceDest – new balance recipient before the transaction.
  - **CATEGORICAL VARIABLES**
    - isFraud
      - 0 – non-Fraudulent activity
      - 1 - Fraudulent activity
    - Type
      - TRANSFER – This refers to a transfer of funds between two accounts, either within the same bank or between different banks.
      - CASH-OUT – This refers to a withdrawal of funds from an ATM or at a bank teller, either by a customer or a merchant.
      - CASH-IN – This refers to a deposit of funds into an account, either by a customer or a merchant.
      - PAYMENT – This refers to a payment made by a customer to a merchant, either online or in person.
      - DEBIT – This refers to a purchase made with a debit card, either online or in person.

❖ **DESCRIPTION OF WORK**

- **Initial Work and EDA**
  - Since, the original dataset is very big. I sampled 100000 values from the dataset keeping all the values with fraudulent activity and sampling from the remaining non-fraudulent rows.

```
df = pd.read_csv('credit_card_fraud.csv')
# reading of this line will give error as the initial data file was very large
# hence I could not upload it on GitHub.

df_is_fraud = df[df['isFraud'] == 1]
print(len(df_is_fraud))
df_not_fraud = df[df['isFraud'] == 0]
print(len(df_not_fraud))

df_new = df_is_fraud.copy()
df_new = pd.concat([df_new, df_not_fraud.sample(n=91787)])

print(len(df_new))

df_new.to_csv('credit_card_fraud_updated.csv')
```

  - I removed some of the columns like 'Unnamed: 0', 'nameOrig', 'nameDest', 'isFlaggedFraud', 'step' which I felt were not relevant for our analysis and will not be helpful for our prediction.

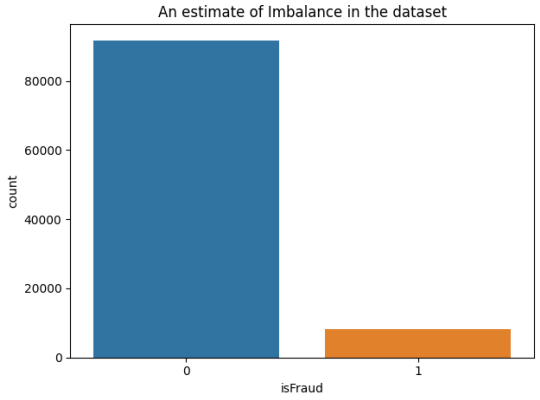


From the plot above, we can see that all the fraudulent activities are taking place when transaction type is ‘transfer’ or ‘cash-out’. More than half the transactions in type ‘transfer’ are fraudulent.

- Using columnTransformer, I have transformed the numerical variables using standard scaler and categorical variable using one-hot encoding. The code snippet is as follows:

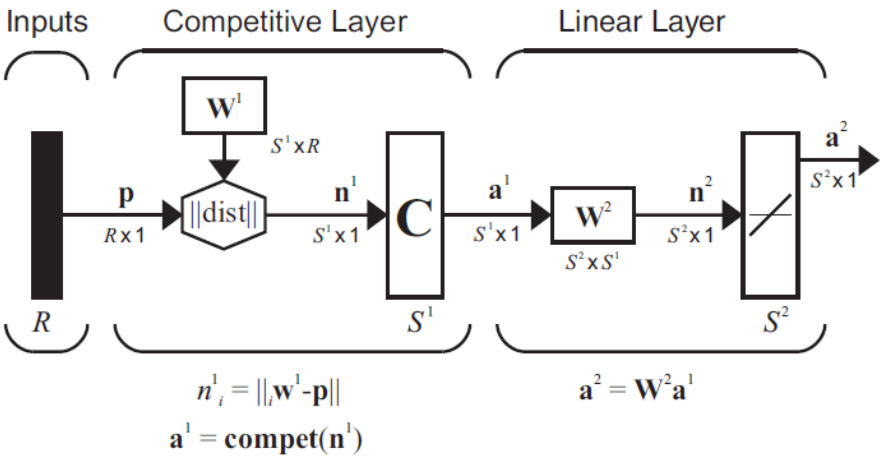
```
categorical_features = ['type']
numeric_features = ['amount', 'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest', 'newbalanceDest']

ohe = OneHotEncoder(sparse=False)
scaler = StandardScaler()
ct = ColumnTransformer([
    ('ohe', ohe, categorical_features),
    ('scaler', scaler, numeric_features)
], remainder='passthrough')
# reference: chatgpt
```

- I have divided the dataset into three sets namely train data, validation data, and test data in the ratio of 70:15:15.



- As expected from any fraud detection dataset, we have a highly imbalanced dataset as shown above in the plot. So, to deal with the imbalance, I have used SMOTE to up sample the minority class in train data.

- **LVQ ALGORITHM DEVELOPMENT**



- The main reference of the algorithm has been taken from Neural Network textbook.
- I have developed a class of LVQ (tried to follow the sklearn terminologies and incorporate them in my code development). I have essentially developed two main functions of the class called 'fit()' and 'predict()'. Other than these functions I have designed a constructor for the class to initialize the class variables called 'alpha' which is the learning rate of the algorithm and 'epochs' i.e. number of iterations for weight training.
- **Constructor initialization**:

```
class LVQ(object):
    Udbhav Kush
    def __init__(self, alpha=0.01, epochs=20):
        self.alpha = alpha
        self.epochs = epochs
        # w2 = [[1, 0, 0], # need to uncomment this for iris dataset
        #       [0, 1, 0],
        #       [0, 0, 1]]
        w2 = [[1, 0],
              [0, 1]]

        self.w2 = np.array(w2)
        # for trying, I am taking just two neurons in the first layer
        # Tried different s1 values but s1=2 worked best for this problem
```

Here I have defined a class called LVQ and a constructor function. Here, as we can see that alpha, epochs, w2 (weight of the second layer in LVQ network) are the class members and initializing them here. There is one drawback for my implementation here, that I am initializing w2 every time for the different problem statements. I wanted to automate this but the number of neurons in layer 1 will vary for different problems, so I could not find a better way to automate this.
A thing to note here is that w2 is a non-trainable parameter in this algorithm.

- **fit function**:

```
def fit(self, X, y):
    y = pd.Series(y)
    X = np.array(X)
    self.w1 = np.random.rand(self.w2.shape[0], len(X[0, :]))
```

  - The first thing I am doing in this fit function is taking X and y as input and converting y to series and converting X to an numpy array. The reason I did this as I have called different functions in the code considering y as series and X as an numpy array. Also, I am initializing weight of first layer as random values as per the shape required for the problem statement.

```
for epoch in range(self.epochs):
    print('epoch = ', epoch+1)
    for i in range(len(X)):
        p_temp = np.array(X[i, :]).reshape(-1, 1)
        n1 = np.zeros((self.w1.shape[0]))
        for j in range(self.w1.shape[0]):
            n1[j] = -np.linalg.norm(self.w1[j].reshape(-1, 1) - p_temp)
        a1 = activation_func('compet', n1)
        a2 = self.w2 @ a1.reshape(-1, 1)
        winner_neuron = np.argmax(a1)  # ref: https://numpy.org/doc/stab
        out = np.argmax(a2)
```

- Here, I am running the epoch iterations in the first loop and the second loop takes in every row in the X or input matrix.

$$n_i^1 = -\left\| {}_i\mathbf{w}^1 - \mathbf{p} \right\|,$$

- The calculation of n1 i.e. the distance between w1 and p using numpy.linalg.norm() function. This function gives the length of the vector or in our use case, distance. This n1 vector is now fed to the competitive activation function. The function is defined as follows:

```
def compet(x):
    max_val = np.max(x)
    res_arr = np.zeros_like(x)
    res_arr[x == max_val] = 1
    return res_arr
```

  I have developed a different file called toolbox.py which has all the supporting functions I use for this course. This function is defined in the toolbox.py file.
- Using simple matrix and vector multiplication, I get a2 (output of the network) and we can extract the predicted class of the input using numpy's argmax function (this function gives the index of highest number in the vector, which in our case would be 1 and rest all will be zero.)
- The code following this will be the weight update algorithm i.e., Kohonen's Rule which is as follows:

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) + \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a_{k^*}^2 = t_{k^*} = 1$$

$${}_{i^*}\mathbf{w}^1(q) = {}_{i^*}\mathbf{w}^1(q-1) - \alpha(\mathbf{p}(q) - {}_{i^*}\mathbf{w}^1(q-1)), \text{ if } a_{k^*}^2 = 1 \neq t_{k^*} = 0$$

```
if out == y.iloc[i]:
    self.w1[winner_neuron] += self.alpha * (p_temp.ravel() - self.w1[winner_neuron])
else:
    self.w1[winner_neuron] -= self.alpha * (p_temp.ravel() - self.w1[winner_neuron])
    k = len(n1)-1
    while True:
        a1 = np.zeros(n1.shape)
        ele = n1.argsort()[k]
        a1[ele] = 1
        a2 = self.w2 @ a1.reshape(-1, 1)
        out = np.argmax(a2)
        if out == y.iloc[i]:
            self.w1[ele] += self.alpha * (p_temp.ravel() - self.w1[ele])
            break
        else:
            k -= 1
```

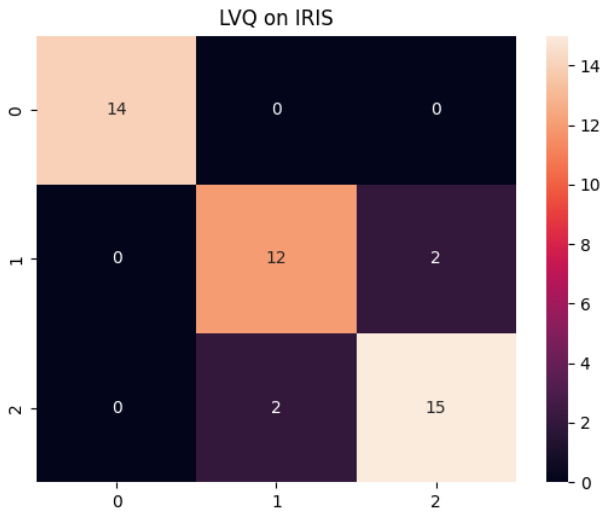
- I am using the weight update algorithm of LVQ2. So, whenever the network correctly classifies the input the weight of winning neuron is moved closer to the input, so, effectively one weight update. But if the network misclassifies the input, the weight of the winning neuron is moved away from the input and the neuron closest to the input which correctly classifies the input is moved closer to the input.
    - The first if condition is the condition of correct classification and the weight update is done as per the Kohonen rule.
    - The else condition is the misclassification condition,
        - Winning neuron is moved away from the input.
        - Using the while loop, I try to find the closest neuron which correctly classifies the input and updates the weight accordingly. (numpy.argsort() function gives the indices of the sorted array in increasing order. Since, we are taking negative of the distance using numpy.norm(), we iterate from the back to get minimum magnitude of the distance.)

- **predict function:**

```
def predict(self, X):
    X = np.array(X)
    y_pred = []
    for i in range(len(X)):
        p_temp = np.array(X[i, :]).reshape(-1, 1)
        n1 = np.zeros((self.w1.shape[0]))
        for j in range(self.w1.shape[0]):
            n1[j] = -distance(self.w1[j].reshape(-1, 1) - p_temp)

        a1 = activation_func('compet', n1)
        a2 = self.w2 @ a1.reshape(-1, 1)
        out = np.argmax(a2)
        y_pred.append(out)

    y_pred = np.array(y_pred)

    return y_pred
```

- Predict function takes X as an input.
- For all the rows in X, I take the value of each column and compute the output of the second layer as I did in the fit function.
- The output of each input is appended to an array and finally a numpy array of predicted values is returned by the function.

- **Working Class for Iris Dataset:**
  - To check the working of the developed class, I used Iris dataset.
  - Following is the confusion matrix for the IRIS dataset:



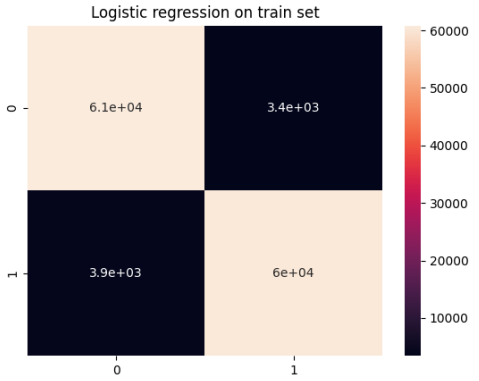
I ran the algorithm just for 5 epochs with an alpha value of 0.05. We just got two misclassified values for class 1 and class 2 each on the test set.

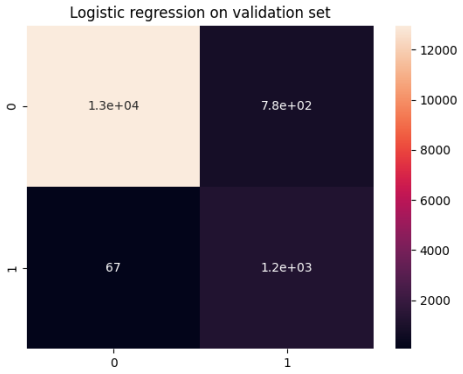
- **Modelling:** I chose 4 classification algorithms to predict the fraudulent activities for the dataset. The algorithms are:
  - Logistic Regression
  - Decision Tree
  - K-Nearest Neighbors
  - Learning Vector Quantization

I ran Logistic Regression, Decision Tree, and KNN using sklearn library. I used the default parameters for these three algorithms and compared them with my implementation of LVQ algorithm. The results of all these algorithms will be displayed in the next section.

❖ RESULTS

- **LOGISTIC REGRESSION**



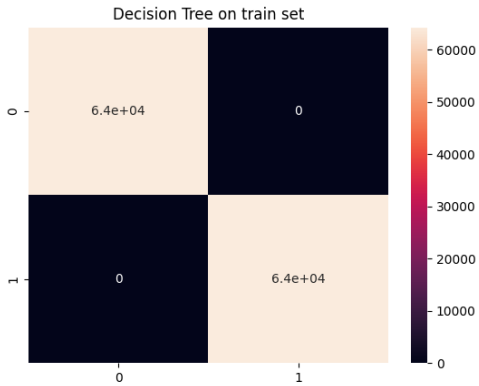


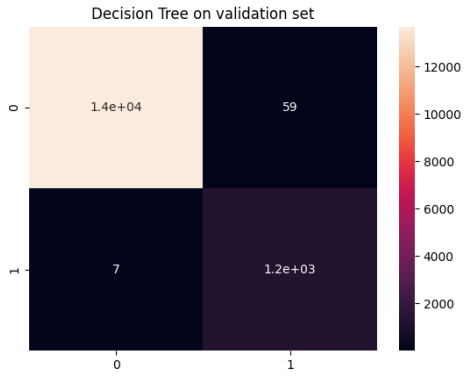

```
Results for logistic regression on train set:
[[60850  3437]
 [ 3908 60379]]
              precision    recall  f1-score   support

           0       0.94      0.95      0.94     64287
           1       0.95      0.94      0.94     64287

    accuracy                           0.94    128574
   macro avg       0.94      0.94      0.94    128574
weighted avg       0.94      0.94      0.94    128574
```

```
Results for logistic regression on validation set:
[[12963   782]
 [   67  1188]]
              precision    recall  f1-score   support

           0       0.99      0.94      0.97     13745
           1       0.60      0.95      0.74      1255

    accuracy                           0.94     15000
   macro avg       0.80      0.94      0.85     15000
weighted avg       0.96      0.94      0.95     15000
```

As we can see from the confusion matrix, we are getting a lot more of false negative as compared to the false positives. This is due to the imbalance of the dataset. I tried to fix the imbalance using SMOTE, but the results are not very satisfactory even though the F1 score for validation set is decent. We can also see that on the training set, precision for class 1 is very good but on the validation set, precision for class is not that great. This is due to the overfitting of the model.

- **DECISION TREE**



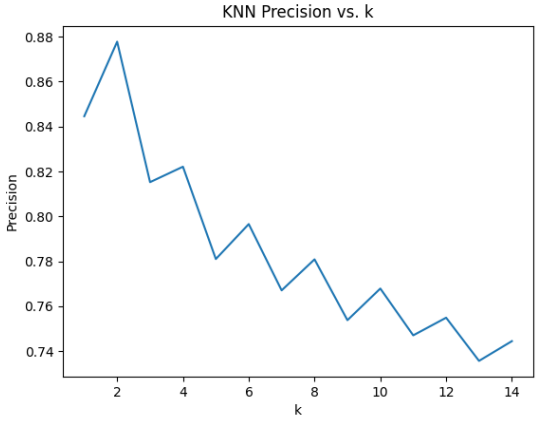


```
Results of decision tree on the train set:
[[64287     0]
 [    0 64287]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     64287
           1       1.00      1.00      1.00     64287

    accuracy                           1.00    128574
   macro avg       1.00      1.00      1.00    128574
weighted avg       1.00      1.00      1.00    128574
```

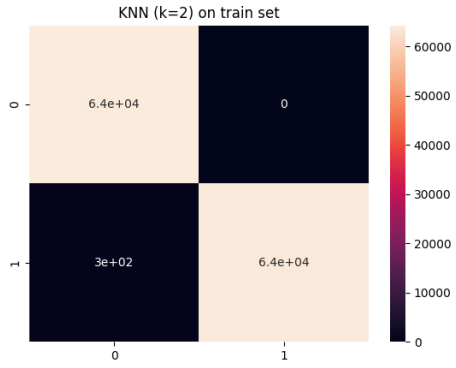
```
Results of decision tree on the validation set:
[[13686    59]
 [    7  1248]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     13745
           1       0.95      0.99      0.97      1255

    accuracy                           1.00     15000
   macro avg       0.98      1.00      0.99     15000
weighted avg       1.00      1.00      1.00     15000
```

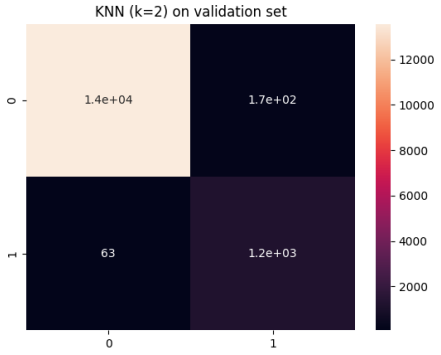
From the heat map of confusion matrix, we can see that we get 100 percent results on the train set and very decent results on the validation set. There might be some overfitting on decision tree as we are getting 100 percent results using this algorithm.

- **K-NEAREST NEIGHBORS**



From the plot precision VS k plot, we can see that we are getting the maximum precision at k=2 on the validation set.





From the heatmap above, we can see that for the train set, there are no false negatives (considering 0 as negative class and 1 as positive class) but there are a few false positives while on the validation dataset, opposite observed i.e., there are a lot more false negatives and very few false positives. The reason for this could be overfitting on the training data. since the validation data has more false negatives and fewer false positives, it could mean that the model is not able to generalize well to new, unseen data and is missing some of the positive cases.

Results of KNN on the train set:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	64287
1	1.00	1.00	1.00	64287
accuracy			1.00	128574
macro avg	1.00	1.00	1.00	128574
weighted avg	1.00	1.00	1.00	128574

Results of KNN on the validation set:

	precision	recall	f1-score	support
0	1.00	0.99	0.99	13745
1	0.88	0.95	0.91	1255
accuracy			0.98	15000
macro avg	0.94	0.97	0.95	15000
weighted avg	0.99	0.98	0.99	15000

The same thing is reflected in the classification report. The train set has 100 percent results while the validation is not performing very well on class 1.

- **LEARNING VECTOR QUANTIZATION**

LVQ classification on train set:

	precision	recall	f1-score	support
0	0.75	0.89	0.82	64287
1	0.87	0.70	0.78	64287
accuracy			0.80	128574
macro avg	0.81	0.80	0.80	128574
weighted avg	0.81	0.80	0.80	128574

LVQ classification on validation set:

	precision	recall	f1-score	support
0	0.97	0.89	0.93	13745
1	0.38	0.71	0.50	1255
accuracy			0.88	15000
macro avg	0.68	0.80	0.71	15000
weighted avg	0.92	0.88	0.90	15000

From the results above, we can clearly see that this is a case of overfitting as precision for class in training set is satisfactory but precision for class1 for validation set is bad. I used two neurons in the first layer and 2 neurons in the second layer.

Since, LVQ algorithm does not require the inputs to be normalized, I tried to use my implementation of LVQ algorithm on non-normalized dataset.

LVQ classification on train set:

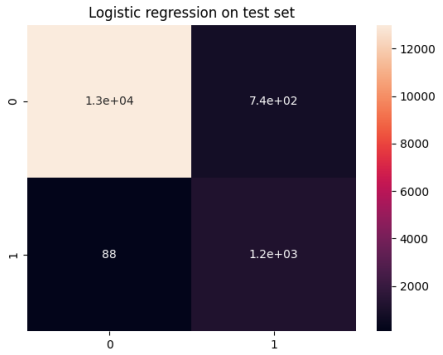
	precision	recall	f1-score	support
0	0.78	0.88	0.82	64287
1	0.86	0.75	0.80	64287
accuracy			0.81	128574
macro avg	0.82	0.81	0.81	128574
weighted avg	0.82	0.81	0.81	128574

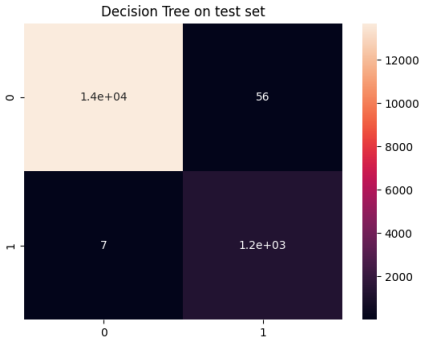
LVQ classification on validation set:

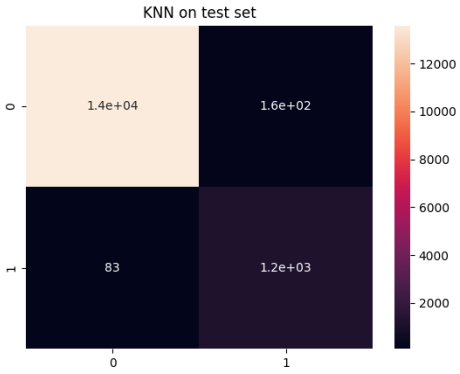
	precision	recall	f1-score	support
0	0.97	0.88	0.92	13745
1	0.35	0.75	0.48	1255
accuracy			0.86	15000
macro avg	0.66	0.81	0.70	15000
weighted avg	0.92	0.86	0.89	15000

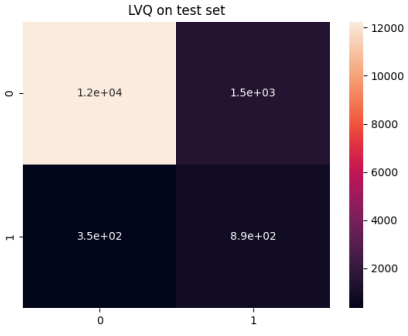
We got similar results on validation set on non-normalized dataset, which was expected.

- **RESULTS ON TEST SET**









If we compare the confusion matrices of different models on the test, we can see that decision tree performs the best on the test. All other models are overfitting on the train set hence the high number of type 1 error and type 2 error.

```
Results for logistic regression on test set:
[[13010   745]
 [   88  1157]]
              precision    recall  f1-score   support

           0       0.99      0.95      0.97     13755
           1       0.61      0.93      0.74      1245

    accuracy                           0.94     15000
   macro avg       0.80      0.94      0.85     15000
weighted avg       0.96      0.94      0.95     15000
```

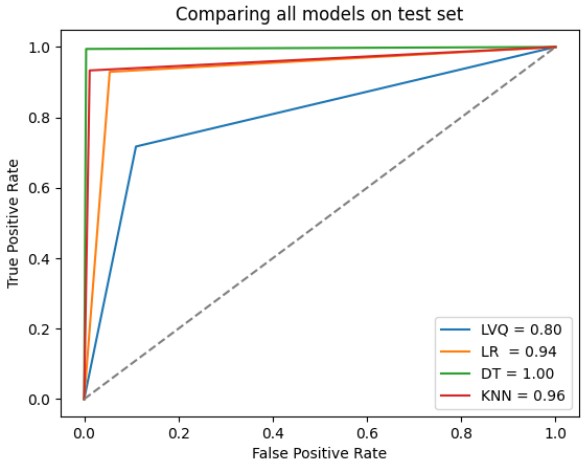
```
Results for decision tree on test set:
[[13699    56]
 [    7  1238]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     13755
           1       0.96      0.99      0.98      1245

    accuracy                           1.00     15000
   macro avg       0.98      1.00      0.99     15000
weighted avg       1.00      1.00      1.00     15000
```

```
Results for KNN on test set:
[[13595   160]
 [   83  1162]]
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     13755
           1       0.88      0.93      0.91      1245

    accuracy                           0.98     15000
   macro avg       0.94      0.96      0.95     15000
weighted avg       0.98      0.98      0.98     15000
```

```
Results for LVQ on test set:
[[12244  1511]
 [  352   893]]
              precision    recall  f1-score   support

           0       0.97      0.89      0.93     13755
           1       0.37      0.72      0.49      1245

    accuracy                           0.88     15000
   macro avg       0.67      0.80      0.71     15000
weighted avg       0.92      0.88      0.89     15000
```

```
Results for LVQ on test set non standardized:
[[11998  1757]
 [  287   958]]
              precision    recall  f1-score   support

           0       0.98      0.87      0.92     13755
           1       0.35      0.77      0.48      1245

    accuracy                           0.86     15000
   macro avg       0.66      0.82      0.70     15000
weighted avg       0.92      0.86      0.89     15000
```

Classification report tells the same story of over fitting on all the models. Decision tree giving satisfactory results on the test set. All other models have reduced precision for class1 hence, affecting the F1-score for that class.



From the AUC curve, we see that decision tree is giving 100 percent results which is probably due to overfitting and my implementation of LVQ is giving 80% area under the curve. As we observed above, that the model is overfitting too, we cannot rely much on the AUC plot curve.

❖ SUMMARY AND CONCLUSION

- The major cause of the fraudulent activities which were observed from the EDA of the dataset was the type of transaction occurring. Transaction type which involved transfer of money from one bank to another resulted in the most fraudulent transactions and transaction type involving taking cash-out from the payment system resulted in fraudulent transactions.
- Almost all the models I tried on the dataset were overfitting even when keeping the simplest algorithm structure.
- LVQ algorithm can be a good classification algorithm and we do not need standardized dataset to implement LVQ as we are dealing with distances.

❖ LIMITATIONS AND FUTURE SCOPE

- My implementation is overfitting on the train dataset. So, I need to come up with some mechanism to avoid overfitting like early stopping in my implementation.
- The implementation is not generalized for the number of subclasses i.e., number of neurons in the first layer. I need to generalize the code by taking the number of neurons by the user as the class object.
- Classical machine learning models are overfitting too. I need to make the models simpler by hyperparameter tuning or K-fold validation to avoid overfitting.

❖ PERCENTAGE OF CODE FROM INTERNET

Lines of code copied from the internet: around 25.
Total lines of code excluding spaces and comments: around 490
Percentage copied from the internet: 5.10%

❖ REFERENCES

- Hagan, M. T. (Second Edition). *Neural network design*. Martin Hagan.
- (ChatGPT, personal communication, May 2nd, 2023).
- sklearn.linear_model.LogisticRegression. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- sklearn.neighbors.KNeighborsClassifier. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
- sklearn.tree.DecisionTreeClassifier. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
- SMOTE — Version 0.10.1. (n.d.). https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html#imblearn.over_sampling.SMOTE.fit_resample
- numpy.argmax — NumPy v1.24 Manual. (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.argmax.html
- numpy.argsort — NumPy v1.24 Manual. (n.d.). https://numpy.org/doc/stable/reference/generated/numpy.argsort.html
- sklearn.metrics.roc_curve. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html
- sklearn.metrics.roc_auc_score. (n.d.). Scikit-learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html