Chapter 2: Architecure of AI Model to detect Credit Fraud Detection

Introduction: We will Build a Fraud Detection Model with Machine Learning Algorithms (Logistic Regression,KNN,SVM etc) & Deep Learning Algorithms with Neural Network & Auto-Encoder Decoder along side we will work on SMOTE analysis for Sampling

**Structure:**

* Problem Statement
* Python Library Upload & Visualization Analysis
* Scaling and Distributing
* Machine learning Modelling of Random Sampling
* Over-Sampling
* Artificial Neural Networks
* AutoEncoder Model Prediction Architecture
* Summary

**Objectives**

We will cover the following recipes:

* Various Statistical Analysis & Visualization using one of the best tool (Plotly)
* We will implement SMOTE +Deep learning Model for Under sampling and Oversampling to tackle Unbalance Class.
* We wil see statistical Analysis such as T-SNE.
* Evaluating the performance of Classification algorithms.
* Working on Different Deep Learning & Machine Learning Hybrid Models such as Auto-Encoder Decoder.
* We will see varius advance plotly plots for rvaluation process like ROC-Curve,

Precision-Recall Curve,Confusion-Matrix etc

Understand the TensorFlow & Keras ecosystem using various datasets and techniques complete end to end Framework of Artificial Intelligence.

**2.1 Problem Statement**

It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.

## Content: The datasets contains transactions made by credit cards in September 2013 by european cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

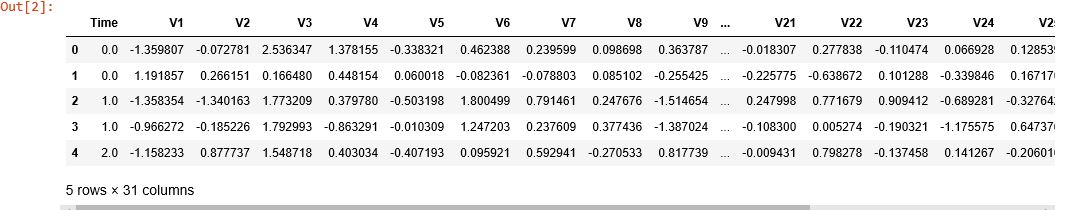
It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-senstive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

**2.2 Python Library Upload & Visualization Analysis**

* from imblearn.pipeline import make\_pipeline as imbalanced\_make\_pipeline
* from imblearn.over\_sampling import SMOTE
* from imblearn.under\_sampling import NearMiss
* from imblearn.metrics import classification\_report\_imbalanced
* tensorflow as tf
* from sklearn.model\_selection import KFold, StratifiedKFold
* from sklearn.manifold import TSNE
* from keras.layers import Input, Dense
* from keras.models import Model, Sequential

|  |
| --- |
| Rest all the imports I have showed in my Jupyter Notebook, which I gave hyperlink of my Github Account of this chapter.Note:plotly==4.0.0 plotly-express==0.4.1 Anaconda Package Python 2.x/3.x, TensorFlow, Keras.  **Note**  CodeRepository: <https://github.com/aniruddhachoudhury/Artificial-Intelligence-Projects-/tree/master/Chapter2> |

Data\_Credit = pd.read\_csv('creditcard.csv')

Data\_Credit.head()

**Table:** *Table 2.1:* *Credit Card Transaction Data-set*

We Loaded the Data-set & here we having 284,807 transactions rows & 31 columns as features.

print('No Frauds', round(Data\_Credit['Class'].value\_counts()[0]/len(Data\_Credit) \* 100,2), '% of the dataset')

print('Frauds', round(Data\_Credit['Class'].value\_counts()[1]/len(Data\_Credit) \* 100,2), '% of the dataset')

Capture

So here we found here distribution of Fraud vs Non-Fraud Cases in our Data-set.

No\_Frauds= Data\_Credit[(Data\_Credit['Class'] == 0)]

print(len(No\_Frauds))

Frauds = Data\_Credit[(Data\_Credit['Class'] == 1)]

print(len(Frauds))

284315

492

#------------COUNT-----------------------

trace = go.Bar(x = (len(No\_Frauds), len(Frauds)), y = ['No\_Frauds', 'Frauds'], orientation=

'h', opacity = 0.8, marker=dict(color=[ 'green', 'darkviolet'], line=dict(color='#000000',width=1.5)))

layout = dict(title = 'Count of diagnosis variable') fig = dict(data = [trace], layout=layout)

py.iplot(fig)

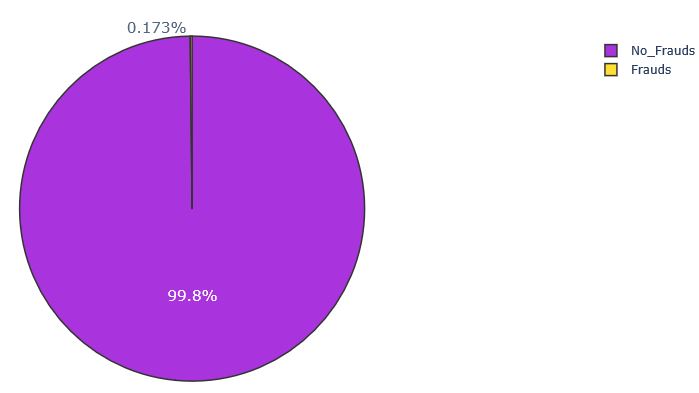
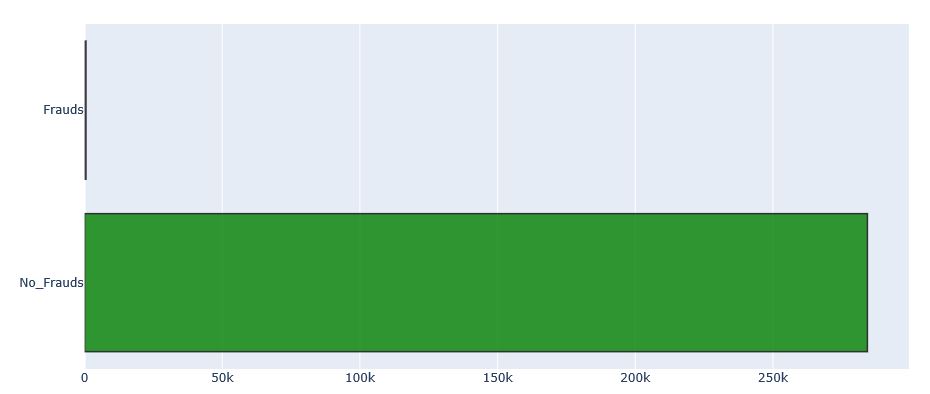
#------------PERCENTAGE-------------------

trace = go.Pie(labels = ['No\_Frauds', 'Frauds'], values = Data\_Credit['Class'].value\_counts(), textfont=dict(size=15), opacity = 0.8,

marker=dict(colors=['darkviolet', 'gold'], line=dict(color='#000000', width=1.5)))

layout = dict(title = 'Distribution of diagnosis variable')

fig = dict(data = [trace], layout=layout)

****py.iplot(fig)

*Figure 2.1: Figure 2.2:*

See How imbalanced is our original dataset! Most of the transactions are non-fraud. If we use this data frame as the base for our predictive models and analysis we might get a lot of errors and our algorithms will probably overfit since it will assume that most transactions are not fraud. But we don't want our model to assume, we want our model to detect patterns that give signs of fraud!

We will see the Distribution of all the Columns wrt to Fraud and Non-Fraud Cases to find out how skewed are these features we can also see further distributions of the other features. There are techniques that can help the distributions be less skewed.

import plotly.figure\_factory as ff

for i, cn in enumerate(Data\_Credit[v\_features[:1]]):

group\_labels = ['Fraud', 'Non-Fraud']

colors = ['slategray', 'magenta']

x1 = Data\_Credit[cn][Data\_Credit.Class == 1]

x2 = Data\_Credit[cn][Data\_Credit.Class == 0]

fig = ff.create\_distplot([x1, x2], group\_labels, bin\_size=.5,

curve\_type='normal',colors=colors)

fig.update\_layout(title\_text='Distplot with Normal Distribution'+ str(cn))

fig.show()

*Figure 2.3:* 

**2.3 Scaling and Distributing**

Time and amount should be scaled as the other columns. On the other hand, we need to also create a sub sample of the dataframe in order to have an equal amount of Fraud and Non-Fraud cases, helping our algorithms better understand patterns that determines whether a transaction is a fraud or not. Since most of our data has already been scaled we should scale the columns that are left to scale Amount and Time. RobustScaler is less prone to outliers.

from sklearn.preprocessing import StandardScaler, RobustScaler

rob\_scaler = RobustScaler()

Data\_Operation['scaled\_amount']=rob\_scaler.fit\_transform(Data\_Operation['Amount'].values.reshape(-1,1))

Data\_Operation['scaled\_time']=rob\_scaler.fit\_transform(Data\_Operation['Time'].values.reshape(-1,1))

Data\_Operation.drop(['Time','Amount'], axis=1, inplace=True)

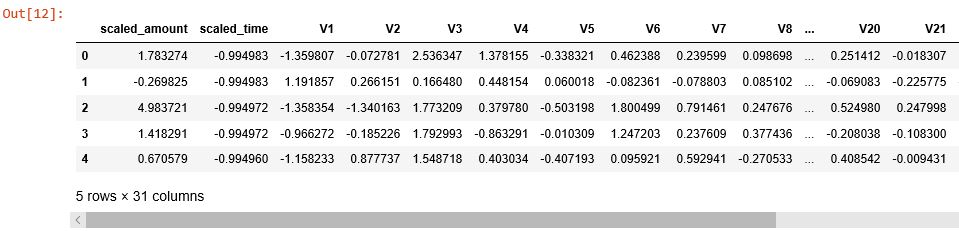
scaled\_amount = Data\_Operation['scaled\_amount']

scaled\_time = Data\_Operation['scaled\_time']

Data\_Operation.drop(['scaled\_amount', 'scaled\_time'], axis=1, inplace=True)

Data\_Operation.insert(0, 'scaled\_amount', scaled\_amount)

Data\_Operation.insert(1, 'scaled\_time', scaled\_time)

****

**Table:** *Table 2.2 Scaled Data-set*

**2.3.1  Splitting of the Original DataFrame**

We are splitting the data when implementing Random UnderSampling or OverSampling techniques, we want to test our models on the original testing set not on the testing set created by either of these techniques.The main goal is to fit the model either with the dataframes that were undersample and oversample (in order for our models to detect the patterns), and test it on the original testing set.

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import StratifiedShuffleSplit

X = Data\_Operation.drop('Class', axis=1)

y = Data\_Operation['Class']

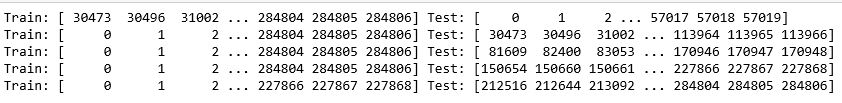
Strats = StratifiedKFold(n\_splits=5, random\_state=None, shuffle=False)

for train\_index, test\_index in Strats.split(X, y):

print("Train:", train\_index, "Test:", test\_index)

original\_Xtrain, original\_Xtest = X.iloc[train\_index], X.iloc[test\_index]

original\_ytrain, original\_ytest = y.iloc[train\_index], y.iloc[test\_index]

**Table:** *Table 2.3: Splitted Data-set*

original\_Xtrain = original\_Xtrain.values

original\_Xtest = original\_Xtest.values

original\_ytrain = original\_ytrain.values

original\_ytest = original\_ytest.values

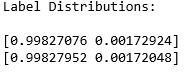
# See if both the train and test label distribution are similarly distributed

train\_unique\_label, train\_counts\_label = np.unique(original\_ytrain, return\_counts=True)

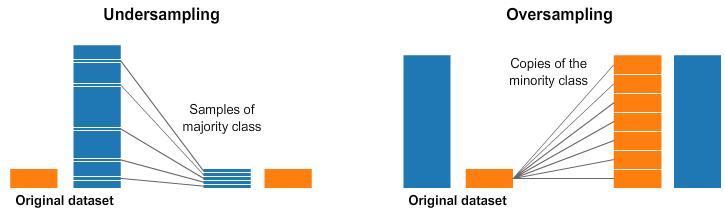
test\_unique\_label, test\_counts\_label = np.unique(original\_ytest, return\_counts=True)

print(train\_counts\_label/ len(original\_ytrain))

print(test\_counts\_label/ len(original\_ytest))

****

**2.3.2  Random Under-Sampling:**

*Figure 2.4:* 

****Steps:****

* The first thing we have to do is determine how **imbalanced** is our class (use "value\_counts()" on the class column to determine the amount for each label)
* Once we determine how many instances are considered **fraud transactions** (Fraud = "1") , we should bring the **non-fraud transactions** to the same amount as fraud transactions (assuming we want a 50/50 ratio), this will be equivalent to 492 cases of fraud and 492 cases of non-fraud transactions.
* After implementing this technique, we have a sub-sample of our dataframe with a 50/50 ratio with regards to our classes. Then the next step we will implement is to **shuffle the data** to see if our models can maintain a certain accuracy everytime we run this script.

#our classes are highly skewed we should make them equivalent in order to have a normal distribution of the classes.

Data\_Operation = Data\_Operation.sample(frac=1)

# amount of fraud classes 492 rows.

fraud\_Data\_Operation = Data\_Operation.loc[Data\_Operation['Class'] == 1]

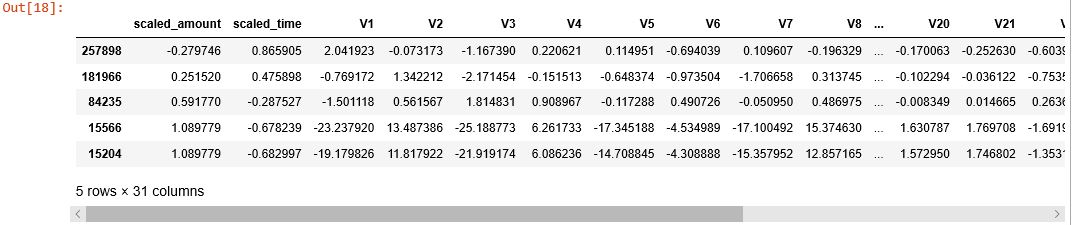
non\_fraud\_Data\_Operation = Data\_Operation.loc[Data\_Operation['Class'] == 0][:492]

normal\_dist\_op = pd.concat([fraud\_Data\_Operation, non\_fraud\_Data\_Operation])

# Shuffle dataframe rows

new\_operation = normal\_dist\_op.sample(frac=1, random\_state=42)

new\_operation.head()



**Table:** *Table 2.4: Shuffled Data-set*

**2.3.4  Equally Distributing and Correlating:**

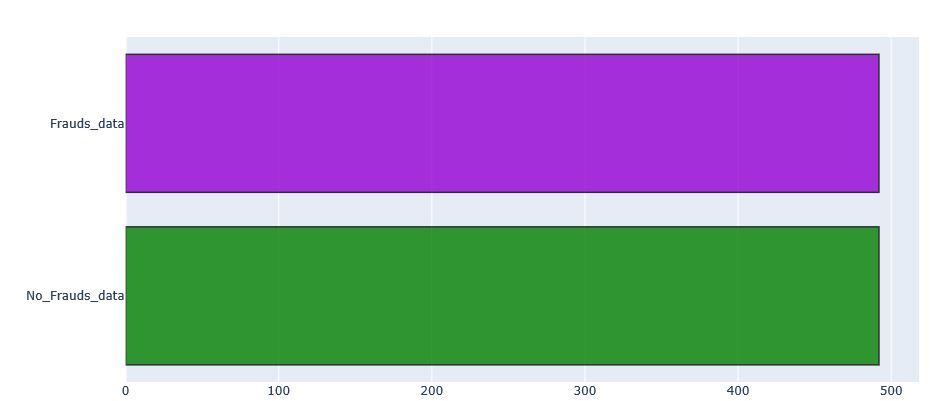
we have our dataframe correctly balanced, we can go further with our **analysis** and **data preprocessing**.

#------------COUNT-----------------------

No\_Frauds\_data= new\_operation[(new\_operation['Class'] == 0)]

Frauds\_data = new\_operation[(new\_operation['Class'] == 1)]

trace = go.Bar(x = (len(No\_Frauds\_data), len(Frauds\_data)), y = ['No\_Frauds\_data', 'Frauds\_data'], orientation = 'h', opacity = 0.8, marker=dict(

 color=[ 'green', 'darkviolet'],

line=dict(color='#000000',width=1.5)))

layout = dict(title 'Count of diagnosis variable')

fig = dict(data = [trace], layout=layout)

py.iplot(fig)

*Figure 2.5:*

**2.3.5  Correlation Matrices**

Correlation matrices are the essence of understanding our data. We want to know if there are features that influence heavily in whether a specific transaction is a fraud. However, it is important that we use the correct dataframe (subsample) in order for us to see which features have a high positive or negative correlation with regards to fraud transactions.

Negative Correlations: V14, V12, V17 and V10 are negatively correlated, the lower these values are, the more likely the end result will be a fraud transaction.

Positive Correlations: V2, V4, V11, and V19 are positively correlated, the higher these values are, the more likely the end result will be a fraud transaction.

BoxPlots:We will use box-plots to have a better understanding of the distribution of these features in fraud case and non fraud transactions.

def correlation\_plotting(data,s):

correlation = data.corr()

matrix\_cols = correlation.columns.tolist()

corr\_array = np.array(correlation)

#Plotting

trace = go.Heatmap(z = corr\_array,x = matrix\_cols,y = matrix\_cols,

xgap = 2,ygap = 2,

colorscale='Viridis',colorbar = dict())

layout = go.Layout(dict(title = 'Correlation Matrix for variables ' +s,

autosize = **False**,height = **720**, width = **800**,

margin = dict(r = **0** ,l = **210**,t = 25,b = 210),

yaxis = dict(tickfont = dict(size = 9)),

xaxis = dict(tickfont = dict(size = 9)),

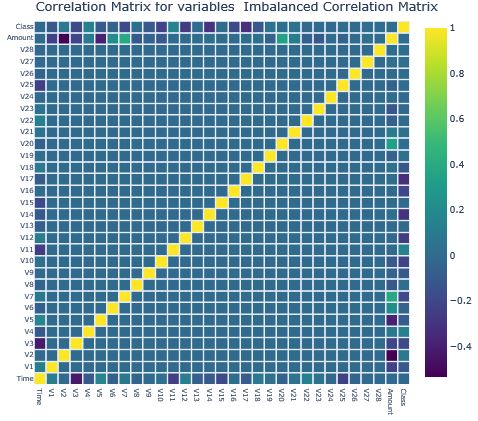
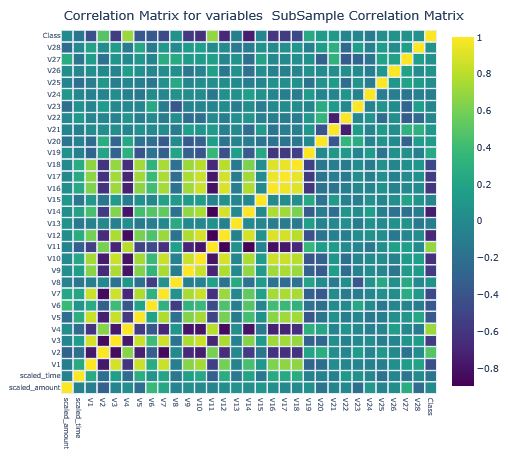
))

fig = go.Figure(data = [trace],layout = layout)

py.iplot(fig)

correlation\_plotting(Data\_Credit,"Imbalanced Correlation Matrix")

correlation\_plotting(new\_operation,"SubSample Correlation Matrix")

****

*Figure 2.6: Figure 2.7:*

import plotly.express as px

**def** box\_plot(s,a):

fig = px.box(new\_operation, x="Class", y=s, points="all",hover\_data=['Class'],

notched=True,)

fig.update\_layout(title\_text=a+s)

fig.show()

lis=['V17','V10','V14','V10']

**for** i in lis:

box\_plot(i,'Box Plot Styling for Class Negative Correlation & ' )



*Figure 2.8:*

**2.3.6  Anomaly Detection**

Our main goal in this section is to remove "extreme outliers" from features that have a high correlation with our classes. This will have a positive impact on the accuracy of our models.

****Outlier Removal Tradeoff:**** We have to be careful as to how far do we want the threshold for removing outliers. We determine the threshold by multiplying a number (ex: 1.5) by the (Interquartile Range). The higher this threshold is, the less outliers will detect (multiplying by a higher number ex: 3), and the lower this threshold is the more outliers it will detect.

v14\_fraud\_dist = new\_operation['V14'].loc[new\_operation['Class'] == 1].values

v12\_fraud\_dist = new\_operation['V12'].loc[new\_operation['Class'] == 1].values

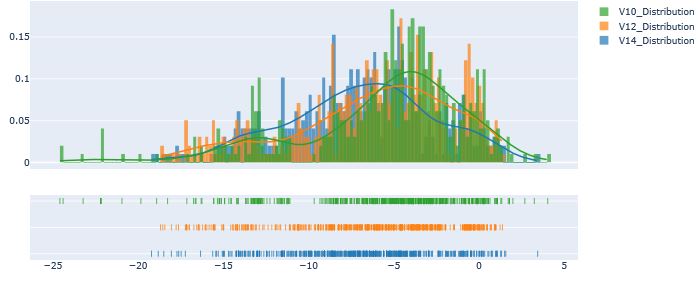
v10\_fraud\_dist = new\_operation['V10'].loc[new\_operation['Class'] == 1].values

hist\_data = [v14\_fraud\_dist, v12\_fraud\_dist,v10\_fraud\_dist]

group\_labels = ['V14\_Distribution','V12\_Distribution','V10\_Distribution']

fig = ff.create\_distplot(hist\_data, group\_labels, bin\_size=.2)

fig.show()

*Figure 2.9:*

**2.3.6.1  **Removing Outliers (Highest Negative Correlated with Labels)****

#V14 removing outliers from fraud transactions

v14\_fraud = new\_operation['V14'].loc[new\_operation['Class'] == 1].values

q25, q75 = np.percentile(v14\_fraud, 25), np.percentile(v14\_fraud, 75)

**print**('Quartile 25: {} | Quartile 75: {}'.format(q25, q75))

v14\_iqr = q75 - q25

print('iqr: {}'.format(v14\_iqr))

**Capture**

v14\_cut\_off\_value = v14\_iqr \* 1.5

v14\_lower, v14\_upper = q25 - v14\_cut\_off\_value, q75 + v14\_cut\_off\_value

**print**('Cut Off: {}'.format(v14\_cut\_off\_value))

**print**('V14 Lower: {}'.format(v14\_lower))

**print**('V14 Upper: {}'.format(v14\_upper))

Capture

outliers = [x for x in v14\_fraud if x < v14\_lower or x > v14\_upper]

**print**('Feature V14 Outliers for Fraud Cases: {}'.format(len(outliers)))

print('V14 outliers:{}'.format(outliers))

Capture

new\_operation = new\_operation.drop(new\_operation[(new\_operation['V14'] > v14\_upper) | (new\_operation['V14'] < v14\_lower)].index)

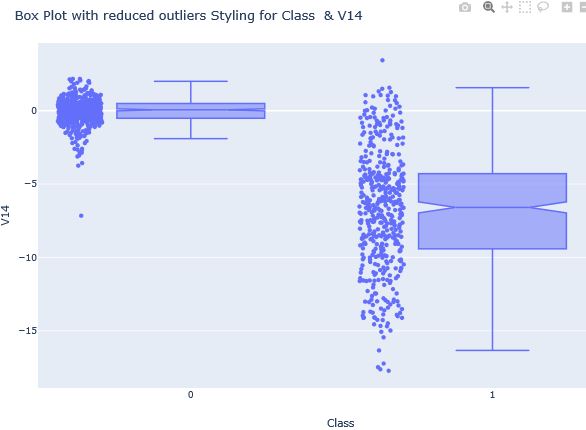
Similarly we have done the operations for other V10,V12 columns

After removing the outlier we will see the Box-Plot and we will check how it looks.

lis=['V14','V12','V10']

for i in lis:

box\_plot(i,'Box Plot with reduced outliers Styling for Class & ' )

****

*Figure 2.10:*

As we can see after plotting the Box-Plot that only few outliers are there after removal.

**2.4  Machine learning modelling of Random Sampling**

**2.4.1 Model Training**

****Learning Curves:****The **wider the gap** between the training score and the cross validation score, the more likely your model is **overfitting (high variance)**. If the score is low in both training and cross-validation sets this is an indication that our model is **underfitting (high bias)**

# Undersampling before cross validating which is prone to overfit

X = new\_operation.drop('Class', axis=1)

y = new\_operation['Class']

# Our data is already scaled we should split our training and test sets

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Turn the values into an array for feeding the classification algorithms.

X\_train = X\_train.values

X\_test = X\_test.values

y\_train = y\_train.values

y\_test = y\_test.values

Here we will run 4 Classification Model for Evaluation and find out the Best

classifiers = {

"LogisiticRegression": LogisticRegression(),

"KNearest": KNeighborsClassifier(),

"Support Vector Classifier": SVC(),

"DecisionTreeClassifier": DecisionTreeClassifier()

}

# Wow our scores are getting even high scores even when applying cross validation.

from sklearn.model\_selection import cross\_val\_score

for key, classifier in classifiers.items():

classifier.fit(X\_train, y\_train)

training\_score = cross\_val\_score(classifier, X\_train, y\_train, cv=5)

print("Classifiers: ", classifier.\_\_class\_\_.\_\_name\_\_, "Has a training score of", round(training\_score.mean(), 2) \* 100, "% accuracy score")

Capture

**2.4.2 GridSearchCV to find the best parameters**

Here we used GridSearchCV to find the best parameters for all the alogorithms.

from sklearn.model\_selection import GridSearchCV

# Logistic Regression

log\_reg\_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

grid\_log\_reg = GridSearchCV(LogisticRegression(), log\_reg\_params)

grid\_log\_reg.fit(X\_train, y\_train)

# We automatically get the logistic regression with the best parameters.

log\_reg = grid\_log\_reg.best\_estimator\_

knears\_params = {"n\_neighbors": list(range(2,5,1)), 'algorithm': ['auto', 'ball\_tree', 'kd\_tree', 'brute']}

grid\_knears = GridSearchCV(KNeighborsClassifier(), knears\_params)

grid\_knears.fit(X\_train, y\_train)

# KNears best estimator

knears\_neighbors = grid\_knears.best\_estimator\_

# Support Vector Classifier

svc\_params = {'C': [0.5, 0.7, 0.9, 1], 'kernel': ['rbf', 'poly', 'sigmoid', 'linear']}

grid\_svc = GridSearchCV(SVC(), svc\_params)

grid\_svc.fit(X\_train, y\_train)

# SVC best estimator

svc = grid\_svc.best\_estimator\_

# DecisionTree Classifier

tree\_params = {"criterion": ["gini", "entropy"], "max\_depth": list(range(2,4,1)),

"min\_samples\_leaf": list(range(5,7,1))}

grid\_tree = GridSearchCV(DecisionTreeClassifier(), tree\_params)

grid\_tree.fit(X\_train, y\_train)

# tree best estimator

tree\_clf = grid\_tree.best\_estimator\_

log\_reg\_score = cross\_val\_score(log\_reg, X\_train, y\_train, cv=5)

print('Logistic Regression Cross Validation Score: ', round(log\_reg\_score.mean() \* 100, 2).astype(str) + '%')

knears\_score = cross\_val\_score(knears\_neighbors, X\_train, y\_train, cv=5)

print('Knears Neighbors Cross Validation Score', round(knears\_score.mean() \* 100, 2).astype(str) + '%')

svc\_score = cross\_val\_score(svc, X\_train, y\_train, cv=5)

print('Support Vector Classifier Cross Validation Score', round(svc\_score.mean() \* 100, 2).astype(str) + '%')

tree\_score = cross\_val\_score(tree\_clf, X\_train, y\_train, cv=5)

print('DecisionTree Classifier Cross Validation Score', round(tree\_score.mean() \* 100, 2).astype(str) + '%')

Capture

**2.4.3  Plotting of Learning Curve**

from sklearn.model\_selection import ShuffleSplit

from sklearn.model\_selection import learning\_curve

**def plots\_trace**(Model,X\_train, y\_train,cv,n\_jobs,string):

fig = go.Figure()

train\_sizes, train\_scores, test\_scores = learning\_curve(Model, X\_train, y\_train, cv=cv, n\_jobs=n\_jobs, train\_sizes=np.linspace(.1, 1.0, 5))

train\_scores\_mean = np.mean(train\_scores, axis=1)

train\_scores\_std = np.std(train\_scores, axis=1)

test\_scores\_mean = np.mean(test\_scores, axis=1)

test\_scores\_std = np.std(test\_scores, axis=1)

fig.add\_trace(go.Scatter(x=train\_sizes, y=train\_scores\_mean,

fill=None, mode='lines',line\_color='indigo',

name="Training Score"))

fig.add\_trace(go.Scatter(x=train\_sizes,y= test\_scores\_mean,fill=None,mode='lines',

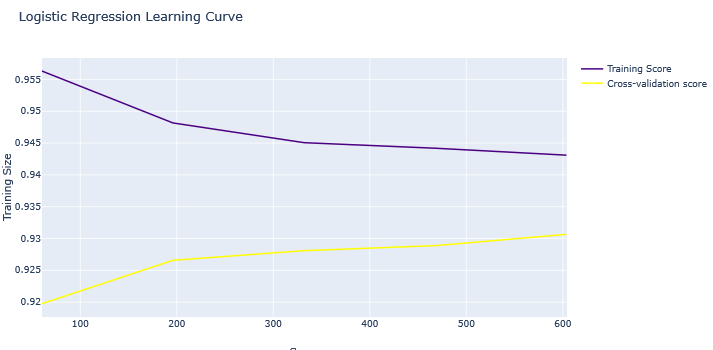
line\_color='yellow',name='Cross-validation score')) fig.update\_layout(title=string,xaxis\_title="Score",yaxis\_title="Training Size")

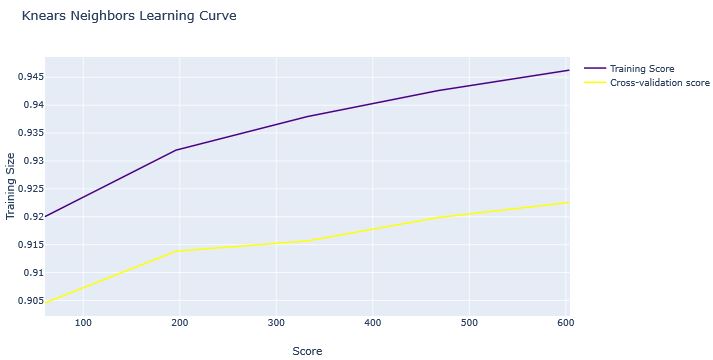
return fig.show()

cv = ShuffleSplit(n\_splits=100, test\_size=0.2, random\_state=42)

Models={log\_reg:'Logistic Regression Learning Curve', knears\_neighbors:'Knears Neighbors Learning Curve', svc:'Support Vector Classifier Learning Curve',tree\_clf:'Decision Tree Classifier Learning Curve'}

for k,v in Models.items():

 plots\_trace(k,X\_train, y\_train,cv,1,v)



*Figure 2.11: Figure 2.12:*

As we can see from the learning curve the model trained properly and its validation loss is decreasing and lower than Training Loss.

**2.4.4  ROC Curve**

from sklearn.metrics import roc\_curve

from sklearn.model\_selection import cross\_val\_predict

log\_reg\_pred = cross\_val\_predict(log\_reg, X\_train, y\_train, cv=5, method="decision\_function")

knears\_pred = cross\_val\_predict(knears\_neighbors, X\_train, y\_train, cv=5)

svc\_pred = cross\_val\_predict(svc, X\_train, y\_train, cv=5,

method="decision\_function")

tree\_pred = cross\_val\_predict(tree\_clf, X\_train, y\_train, cv=5)

from sklearn.metrics import roc\_auc\_score

print('Logistic Regression: ', roc\_auc\_score(y\_train, log\_reg\_pred))

print('KNears Neighbors: ', roc\_auc\_score(y\_train, knears\_pred))

print('Support Vector Classifier: ', roc\_auc\_score(y\_train, svc\_pred))

print('Decision Tree Classifier: ', roc\_auc\_score(y\_train, tree\_pred))

Capture

log\_fpr, log\_tpr, log\_thresold = roc\_curve(y\_train, log\_reg\_pred)

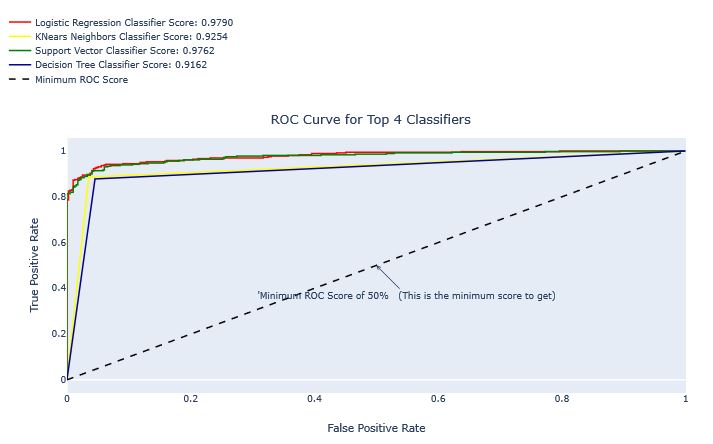
knear\_fpr, knear\_tpr, knear\_threshold = roc\_curve(y\_train, knears\_pred)

svc\_fpr, svc\_tpr, svc\_threshold = roc\_curve(y\_train, svc\_pred)

tree\_fpr, tree\_tpr, tree\_threshold = roc\_curve(y\_train, tree\_pred)

We have created a function and called it here to plots roc curve in plotly you can find out in Github the code so don’t get confuse if you see this method below.

roc\_curve\_plots()

*Figure 2.11:* ****

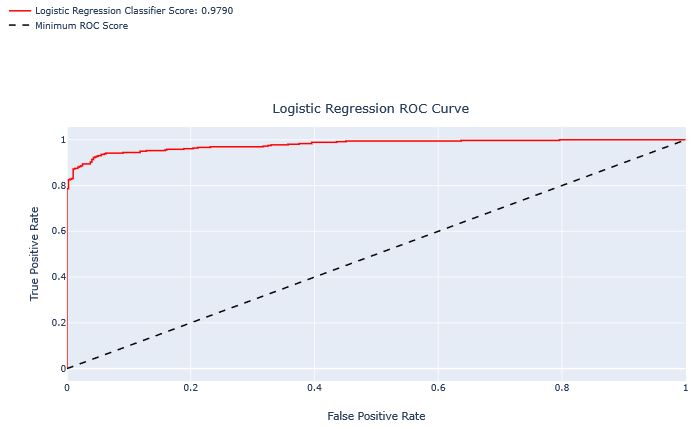
As we can see ROC Curve of Logistic Regression is the Best so we will do some further investigation in Logistic Regression.

**2.4.5  Investigation of Logistic Regression:**

**Precision/Recall Tradeoff:** The more precise (selective) our model is, the less cases it will detect. Example: Assuming that our model has a precision of 95%, Let's say there are only 5 fraud cases in which the model is 95% precise or more that these are fraud cases. Then let's say there are 5 more cases that our model considers 90% to be a fraud case, if we lower the precision there are more cases that our model will be able to detect.

We have created a function and called it here to Logistic ROC\_Curve in plotly you can find out in Github the code so don’t get confuse if you see this method below.

logistic\_roc\_curve(log\_fpr, log\_tpr)



*Figure 2.12:*

from sklearn.metrics import precision\_recall\_curve

precision, recall, threshold = precision\_recall\_curve(y\_train, log\_reg\_pred)

from sklearn.metrics import recall\_score, precision\_score, f1\_score, accuracy\_score

y\_pred = log\_reg.predict(X\_train)

# Overfitting Case

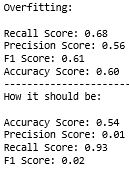
print('Overfitting: \n')

print('Recall Score: {:.2f}'.format(recall\_score(y\_train, y\_pred)))

print('Precision Score: {:.2f}'.format(precision\_score(y\_train, y\_pred)))

print('F1 Score: {:.2f}'.format(f1\_score(y\_train, y\_pred)))

print('Accuracy Score: {:.2f}'.format(accuracy\_score(y\_train, y\_pred)))

print('---' \* 30)

print('How it should be:\n')

print("Accuracy Score: {:.2f}".format(np.mean(undersample\_accuracy)))

print("Precision Score: {:.2f}".format(np.mean(undersample\_precision)))

print("Recall Score: {:.2f}".format(np.mean(undersample\_recall)))

print("F1 Score: {:.2f}".format(np.mean(undersample\_f1)))

undersample\_y\_score = log\_reg.decision\_function(original\_Xtest)

**2.4.6 Precision -Recall Analysis**

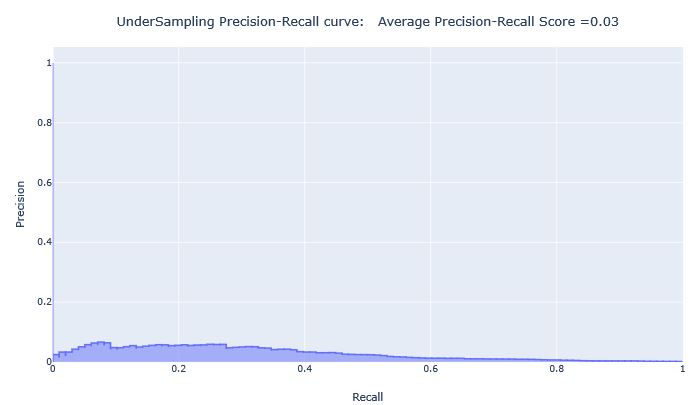
from sklearn.metrics import average\_precision\_score

undersample\_average\_precision=average\_precision\_score(original\_ytest, undersample\_y\_score)

precision, recall, \_ = precision\_recall\_curve(original\_ytest, undersample\_y\_score)

We have created a function and called it here to plots Precision-Recall curve in plotly you can find out in Github the code so don’t get confuse if you see this method below.

precision\_recall\_plot('UnderSampling Precision-Recall curve: \n Average Precision-Recall Score ={0:0.2f}',recall,precision,'hv')

****

*Figure 2.13:*

**2.5  Over-Sampling**

**2.5.1 SMOTE Technique**

**SMOTE** stands for Synthetic Minority Over-sampling Technique. Unlike Random UnderSampling, SMOTE creates new synthetic points in order to have an equal balance of the classes. This is another alternative for solving the "class imbalance problems".

* Solving the Class Imbalance: SMOTE creates synthetic points from the minority class in order to reach an equal balance between the minority and majority class.
* Location of the synthetic points:SMOTE picks the distance between the closest neighbors of the minority class, in between these distances it creates synthetic points.
* Final Effect:More information is retained since we didn't have to delete any rows unlike in random undersampling.
* Accuracy || Time Tradeoff: Although it is likely that SMOTE will be more accurate than random under-sampling, it will take more time to train since no rows are eliminated as previously stated.

from imblearn.over\_sampling import SMOTE

from sklearn.model\_selection import train\_test\_split, RandomizedSearchCV

print('Length of X (train): {} | Length of y (train): {}'.format(len(original\_Xtrain), len(original\_ytrain)))

print('Length of X (test): {} | Length of y (test): {}'.format(len(original\_Xtest), len(original\_ytest)))

# List to append the score and then find the average

accuracy\_lst = []

precision\_lst = []

recall\_lst = []

f1\_lst = []

auc\_lst = []

# Classifier with optimal parameters

log\_reg\_sm = LogisticRegression()

rand\_log\_reg = RandomizedSearchCV(LogisticRegression(), log\_reg\_params, n\_iter=4)

Implementing SMOTE Technique & Cross Validating the right way

log\_reg\_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

for train, test in Strats.split(original\_Xtrain, original\_ytrain):

pipeline = imbalanced\_make\_pipeline(SMOTE(sampling\_strategy='minority'), rand\_log\_reg) model = pipeline.fit(original\_Xtrain[train], original\_ytrain[train])

best\_est = rand\_log\_reg.best\_estimator\_

prediction = best\_est.predict(original\_Xtrain[test])

accuracy\_lst.append(pipeline.score(original\_Xtrain[test], original\_ytrain[test]))

precision\_lst.append(precision\_score(original\_ytrain[test], prediction))

recall\_lst.append(recall\_score(original\_ytrain[test], prediction))

f1\_lst.append(f1\_score(original\_ytrain[test], prediction))

auc\_lst.append(roc\_auc\_score(original\_ytrain[test], prediction))

print("accuracy: {}".format(np.mean(accuracy\_lst)))

print("precision: {}".format(np.mean(precision\_lst)))

print("recall: {}".format(np.mean(recall\_lst)))

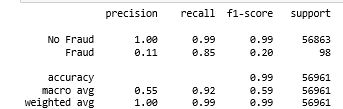
print("f1: {}".format(np.mean(f1\_lst)))



labels = ['No Fraud', 'Fraud']

smote\_prediction = best\_est.predict(original\_Xtest)

print(classification\_report(original\_ytest, smote\_prediction, target\_names=labels))



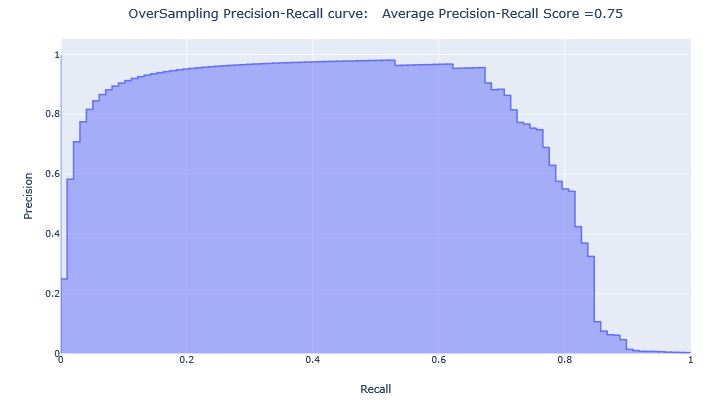
**2.5.2  Precision-recall Curve**

y\_score = best\_est.decision\_function(original\_Xtest)

average\_precision = average\_precision\_score(original\_ytest, y\_score)

precision, recall, \_ = precision\_recall\_curve(original\_ytest, y\_score)

precision\_recall\_plot('OverSampling Precision-Recall curve: \n Average Precision-Recall Score ={0:0.2f}',recall,precision,'vhv',average\_precision)



*Figure 2.14:*

Applying SMOTE Technique on OverSampling data After splitting and Cross Validating

sm = SMOTE(ratio='minority', random\_state=42)

# This will be the data were we are going to

Xsm\_train, ysm\_train = sm.fit\_sample(original\_Xtrain, original\_ytrain)

# Implement GridSearchCV and the other models, Logistic Regression

t0 = time.time()

log\_reg\_sm = grid\_log\_reg.best\_estimator\_

log\_reg\_sm.fit(Xsm\_train, ysm\_train)

t1 = time.time()

**2.5.3  Evaluation of Test Data with Logistic Regression**

****Positive/Negative:**** Type of Class (label) ["No", "Yes"] ****True/False:**** Correctly or Incorrectly classified by the model.

****True Negatives (Top-Left Square):**** This is the number of ****correctly**** classifications of the "No" (No Fraud Detected) class.

****False Negatives (Top-Right Square):**** This is the number of ****incorrectly**** classifications of the "No"(No Fraud Detected) class.

****False Positives (Bottom-Left Square):**** This is the number of ****incorrectly**** classifications of the "Yes" (Fraud Detected) class

****True Positives (Bottom-Right Square):**** This is the number of ****correctly**** classifications of the "Yes" (Fraud Detected) class.

Predicting Logistic Regression fitted using SMOTE technique

y\_pred\_log\_reg = log\_reg\_sm.predict(X\_test)

Predicting Other models fitted with UnderSampling

y\_pred\_knear = knears\_neighbors.predict(X\_test)

y\_pred\_svc = svc.predict(X\_test)

y\_pred\_tree = tree\_clf.predict(X\_test)

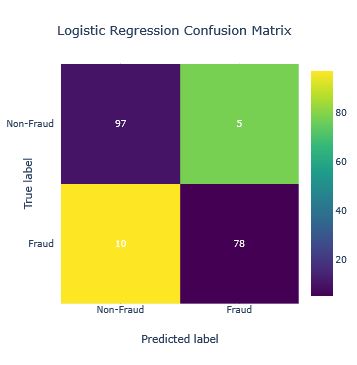
CF\_MODEL={'Logistic Regression Confusion Matrix':log\_reg\_cf, 'Knears Neighbors Confusion Matrix':kneighbors\_cf, 'Support Vector Confusion Matrix':svc\_cf,

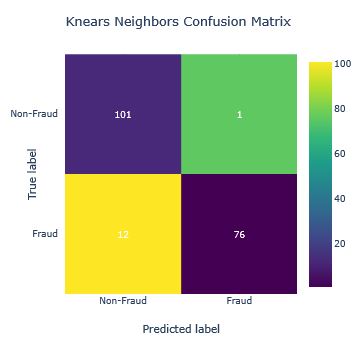
'Decision Tree Confusion Matrix':tree\_cf}

We have created a function and called it here to plots Confusion Matrix in plotly you can find out in Github the code so don’t get confuse if you see this method below.

for k,v in CF\_MODEL.items():

confusion\_matrix\_plot(v,k)

****

****

*Figure 2.15: Figure 2.16:*

We evaluated final Score in the test set of logistic regression

from sklearn.metrics import accuracy\_score

# Logistic Regression with Under-Sampling

y\_pred = log\_reg.predict(X\_test)

undersample\_score = accuracy\_score(y\_test, y\_pred)

# Logistic Regression with SMOTE Technique (Better accuracy with SMOTE t)

y\_pred\_sm = best\_est.predict(original\_Xtest)

oversample\_score = accuracy\_score(original\_ytest, y\_pred\_sm)

Capture

**2.6 Artificial Neural Networks**

**2.6.1  **Keras ~ Random UnderSampling vs Over Sampling**:**

**Dataset**: In this final phase of testing we will fit this model in both the random undersampled subset and oversampled dataset (SMOTE) in order to predict the final result using the original dataframe testing data.

**Neural Network Structure**: As stated previously, this will be a simple model composed of one input layer (where the number of nodes equals the number of features) plus bias node, one hidden layer with 32 nodes and one output node composed of two possible results 0 or 1 (No fraud or fraud).

**Other characteristics:** The learning rate will be 0.001, the optimizer we will use is the AdamOptimizer, the activation function that is used in this scenario is "Relu" and for the final outputs we will use sparse categorical cross entropy, which gives the probability whether an instance case is no fraud or fraud (The prediction will pick the highest probability between the two.)

import keras

from keras import backend as K

from keras.models import Sequential

from keras.layers import Activation

from keras.layers.core import Dense

from keras.optimizers import Adam

from keras.metrics import categorical\_crossentropy

n\_inputs = X\_train.shape[1]

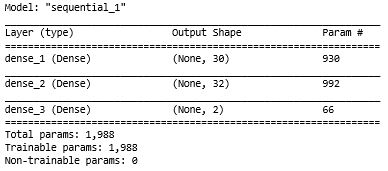
undersample\_model = Sequential([

Dense(n\_inputs, input\_shape=(n\_inputs, ), activation='relu'),

Dense(32, activation='relu'),

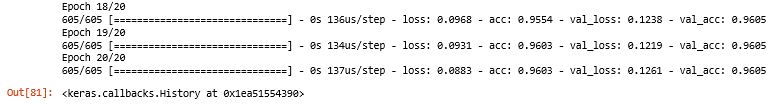
Dense(2, activation='softmax')])

undersample\_model.summary()



undersample\_model.compile(Adam(lr=0.001), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

undersample\_model.fit(X\_train, y\_train, validation\_split=0.2, batch\_size=25, epochs=20, shuffle=True, verbose=1)



undersample\_fraud\_prediction=undersample\_model.predict\_classes(original\_Xtest, batch\_size=200, verbose=0)

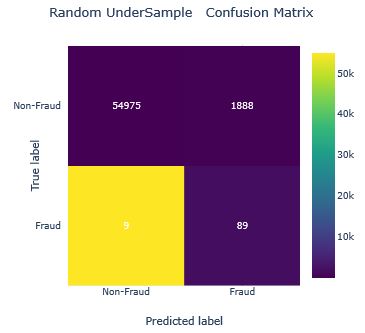
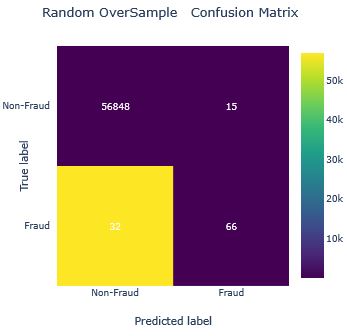
undersample\_cm = confusion\_matrix(original\_ytest, undersample\_fraud\_predictions)

actual\_cm = confusion\_matrix(original\_ytest, original\_ytest)

CF\_MODEL={'Random UnderSample \n Confusion Matrix':undersample\_cm, 'Confusion Matrix (with 100% accuracy)':actual\_cm}

for k,v in CF\_MODEL.items():

confusion\_matrix\_plot(v,k);

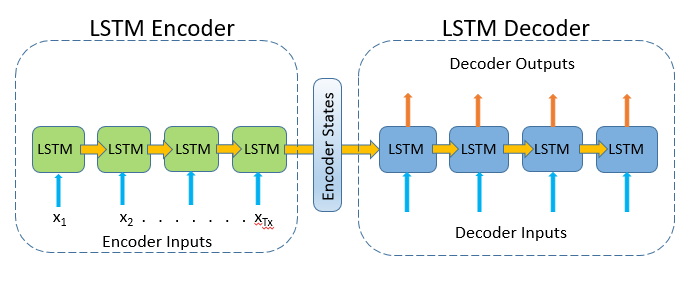
****

*Figure 2.17: Figure 2.18:*

Similarly we predicted via Neural network for over sampling with SMOTE technique above right side we plotted teh confusion Matrix.

So by doing SMOTE implementaion on our imbalanced dataset helped us with the imbalance of our labels (more no fraud than fraud transactions). Nevertheless, I still have to state that sometimes the neural network on the oversampled dataset predicts less correct fraud transactions than our model using the under-sample dataset. However, remember that the removal of outliers was implemented only on the random under-sample dataset and not on the oversampled one. Also, in our under-sample data our model is unable to detect for a large number of cases non fraud transactions correctly and instead, misclassifies those non fraud transactions as fraud cases. Imagine that people that were making regular purchases got their card blocked due to the reason that our model classified that transaction as a fraud transaction, this will be a huge disadvantage for the financial institution. The number of customer complaints and customer disatisfaction will increase.

**2.7 AutoEncoder Model Prediction Architecture**

*Figure 2.18:*

**7.1  Dataset Preparation**

from keras.layers import Input, Dense

from keras.models import Model, Sequential

from keras import regularizers

from sklearn.model\_selection import train\_test\_split

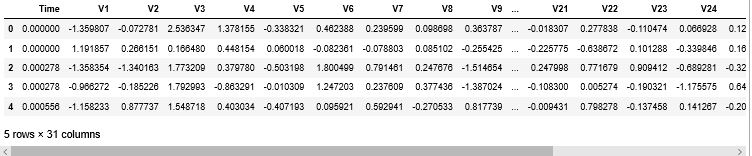
from sklearn.manifold import TSNE

from sklearn import preprocessing

data = pd.read\_csv("creditcard.csv")

data["Time"] = data["Time"].apply(lambda x : x / 3600 % 24)

data.head()

****

**Table:** *Table 2.5:* *Credit Card Transform Data-set*

One of the biggest challenge of this problem is that the target is highly imbalanced as only 0.17 % cases are fraud transactions. But the advantage of the representation learning approach is that it is still able to handle such imbalance nature of the problems. We will look how. For our use-case let's take only about 1000 rows of non-fraud transactions.

non\_fraud = data[data['Class'] == 0].sample(1000)

fraud = data[data['Class'] == 1]

df = non\_fraud.append(fraud).sample(frac=1).reset\_index(drop=True)

X = df.drop(['Class'], axis = 1).values

Y = df["Class"].values

**2.7.1  Visualize Fraud and NonFraud Transactions using T-SNE**

We visualize the nature of fraud and non-fraud transactions using T-SNE. T-SNE (t-Distributed Stochastic Neighbor Embedding) is a dataset decomposition technique which reduced the dimentions of data and produces only top n components with maximum information.

Every dot in the following represents a transaction. Non Fraud transactions are represented as blue while Fraud transactions are represented as Red. The two axis are the components extracted by tsne.

**def** tsne\_plot(x1, y1):

tsne = TSNE(n\_components=2, random\_state=0)

X\_tt = tsne.fit\_transform(x1)

Data=pd.DataFrame(y1,columns={'T'})

Data['T'] = Data['T'].apply(lambda x: 'Fraud' if x==1 else 'Non-Fraud')

Target=Data.values

d=pd.DataFrame()

d['x']=X\_tt[:,0]

d['y']=X\_tt[:,1]

d['target']=Target

fig = px.scatter(d, x="x", y="y", color="target")

fig.update\_traces(marker=dict(size=12,

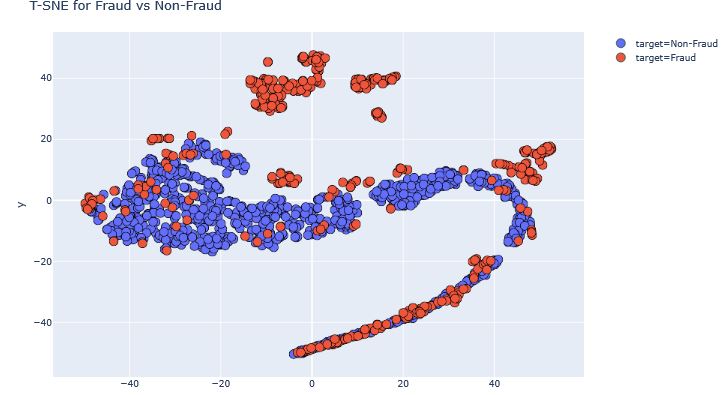
line=dict(width=1,

color='DarkSlateGrey')),

selector=dict(mode='markers'))

fig.layout.title.text = "T-SNE for Fraud vs Non-Fraud"

fig.show()

tsne\_plot(X,Y)

*Figure 2.19:*

**2.7.3  AutoEncoders Modelling**

We will create an autoencoder model in which we only show the model non-fraud cases. The model will try to learn the best representation of non-fraud cases. The same model will be used to generate the representations of fraud cases and we expect them to be different from non-fraud ones.

Create a network with one input layer and one output layer having identical dimentions ie. the shape of non-fraud cases. We will use keras package

## input layer

input\_layer = Input(shape=(X.shape[1],))

## encoding part

encoded=Dense(100,activation='tanh',activity\_regularizer=regularizers.l1(10e-5))(input\_layer)

encoded = Dense(50, activation='relu')(encoded)

## decoding part

decoded = Dense(50, activation='tanh')(encoded)

decoded = Dense(100, activation='tanh')(decoded)

## output layer

output\_layer = Dense(X.shape[1], activation='relu')(decoded)

autoencoder = Model(input\_layer, output\_layer)

autoencoder.compile(optimizer="adadelta", loss="mse")

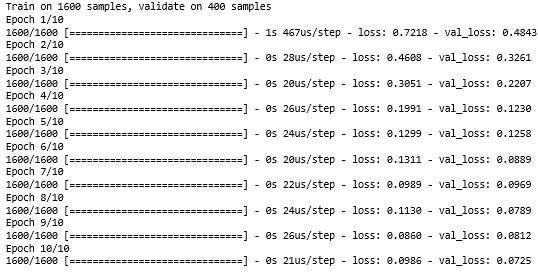
x = data.drop(["Class"], axis=1)

y = data["Class"].values

x\_scale = preprocessing.MinMaxScaler().fit\_transform(x.values)

x\_norm, x\_fraud = x\_scale[y == 0], x\_scale[y == 1]

autoencoder.fit(x\_norm[0:2000], x\_norm[0:2000],batch\_size = 256, epochs = 10, shuffle = True, validation\_split = 0.20);



**2.7.4  Obtain the Latent Representations**

Now, the model is trained. We are intereseted in obtaining latent representation of the input learned by the model. This can be accessed by the weights of the trained model. We will create another network containing sequential layers, and we will only add the trained weights till the third layer where latent representation exists.

hidden\_representation = Sequential()

hidden\_representation.add(autoencoder.layers[0])

hidden\_representation.add(autoencoder.layers[1])

hidden\_representation.add(autoencoder.layers[2])

norm\_hid\_rep = hidden\_representation.predict(x\_norm[:3000])

fraud\_hid\_rep = hidden\_representation.predict(x\_fraud)

**2.7.5  Visualize T-SNE the latent representations : Fraud Vs Non Fraud**

We will create a training dataset using the latent representations obtained and let's visualize the nature of fraud vs non-fraud cases.

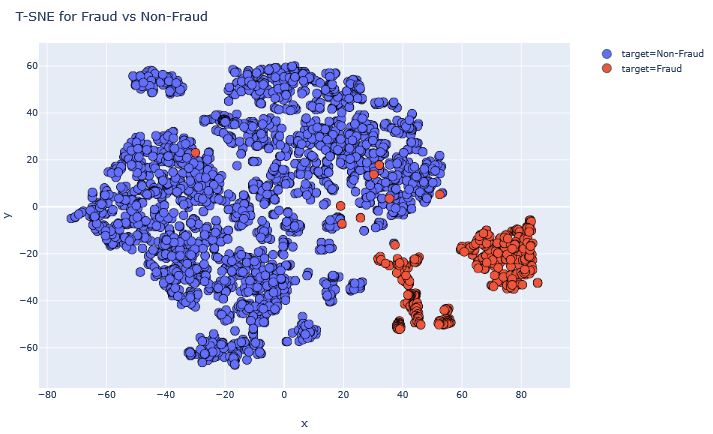
rep\_x = np.append(norm\_hid\_rep, fraud\_hid\_rep, axis = 0)

y\_n = np.zeros(norm\_hid\_rep.shape[0])

y\_f = np.ones(fraud\_hid\_rep.shape[0])

rep\_y = np.append(y\_n, y\_f)

tsne\_plot(rep\_x, rep\_y)

****

*Figure 2.20:*

Perfect graph, we can observe that now fraud and non-fraud transactions are pretty visibile and are linearly separable.

**2.8 Summary**

* Here we have learnt complete how to tackle Imbalance Dataset for a classification Problem and we have prediction for Under Sampling & Oversampling.
* We have learnt how to implement SMOTE Technique and Outlier Detection to remove them and made prediction.
* And got to know the Neural Network architecture + SMOTE prediction.
* Agarin we have learnt a new Deep learning Architecture Auto-Encoder Decoder with Keras Framework.
* We built the T-SNE plot to analyse the classificatio of our Dataset and various plotly advanced plots.