Person No: 50441793
Saran Rohit Tananki
rangasai@buffalo.edu

Machine Learning Assignment 3

University at Buffalo, Buffalo NY 14260

Hidden Markov Model Report

1 Hidden Markov Model :

A **hidden Markov Model** (**HMM**) is a statistical Markov model in which the system being modeled is assumed to be a Markov process call it with unobservable ("*hidden*") states. As part of the definition, HMM requires that there be an observable process whose outcomes are "influenced" by the outcomes of in a known way. Hidden Markov Models (HMMs) are a class of probabilistic graphical model that allow us to predict a sequence of unknown (hidden) variables from a set of observed variables. A simple example of an HMM is predicting the weather (hidden variable) based on the type of clothes that someone wears (observed). An HMM can be viewed as a Bayes Net unrolled through time with observations made at a sequence of time steps being used to predict the best sequence of hidden states. The reason it is called a Hidden Markov Model is because we are constructing an inference model based on the assumptions of a Markov process. The Markov process assumption is simply that the “future is independent of the past given the present”. In other words, assuming we know our present state, we do not need any other historical information to predict the future state.

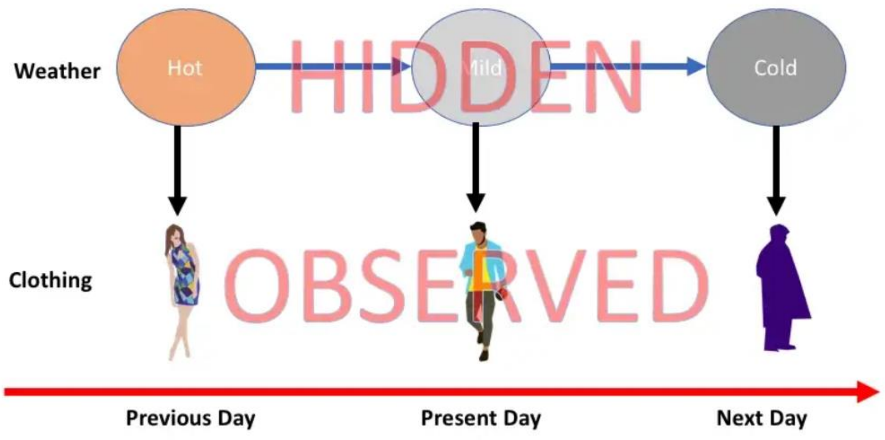


Figure 1: HMM

Generally, the term “states” are used to refer to the hidden states and “observations” are used to refer to the observed states.

1. Transition data — the probability of transitioning to a new state conditioned on a present state.
2. Emission data — the probability of transitioning to an observed state conditioned on a hidden state.
3. Initial state information — the initial probability of transitioning to a hidden state. This can also be looked at as the prior probability.

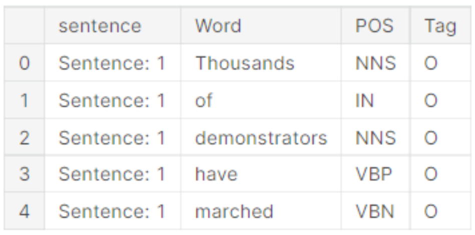
The above information can be computed directly from our training data. For example, in the case of our weather example in Figure 1, our training data would consist of the hidden state and observations for a number of days. We could build our transition matrices of transitions, emissions and initial state probabilities directly from this training data.

Note that as the number of observed states and hidden states gets large the computation gets more computationally intractable. If there are k possible values for each hidden sequence and we have a sequence length of n, there there are n^k total possible sequences that must be all scored and ranked in order to determine a winning candidate.

2 Experiment

2.1 Dataset

The dataset used for this experiment is Name Entity Recognition (NER) Dataset . This is a very clean dataset and is for anyone who wants to try their hand on the NER (Named Entity recognition) task of NLP. The dataset with 1M x 4 dimensions contains columns = ['# Sentence', 'Word', 'POS', 'Tag'] and is grouped by #Sentence. This dataset is taken from Kaggle - https://www.kaggle.com/datasets/debasisdotcom/name-entity-recognition-ner-dataset



	sentence	Word	POS	Tag
0	Sentence: 1	Thousands	NNS	O
1	Sentence: 1	of	IN	O
2	Sentence: 1	demonstrators	NNS	O
3	Sentence: 1	have	VBP	O
4	Sentence: 1	marched	VBN	O

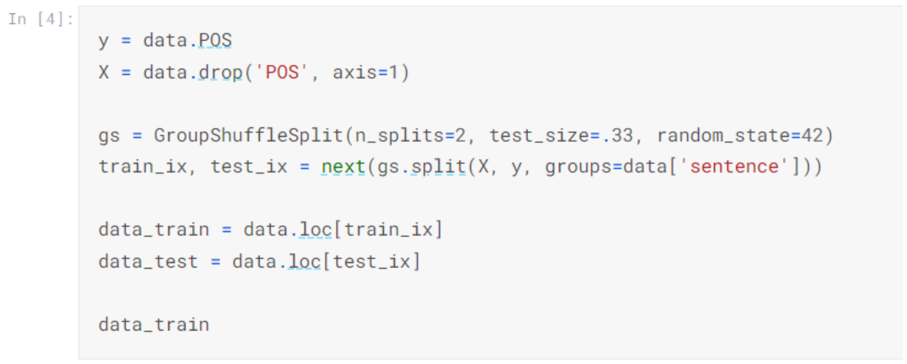
Figure 2: head of dataset

2.1.1 Data Engineering

Data cleaning is the process that removes data that does not belong in your dataset. Data transformation is the process of converting data from one format or structure into another. Transformation processes can also be referred to as data wrangling .For proceeding further, first we need to analyse the data. For this, we dealt with some missing values and null values. This can make analysis more efficient and minimize distraction from your primary target—as well as creating a more manageable and more performant dataset. Often, there will be one-off observations where, at a glance, they do not appear to fit within the data you are analyzing. If you have a legitimate reason to remove an outlier, like improper data-entry, doing so will help the performance of the data you are working with. you can drop observations that have missing values, but doing this will drop or lose information, so be mindful of this before you remove it.

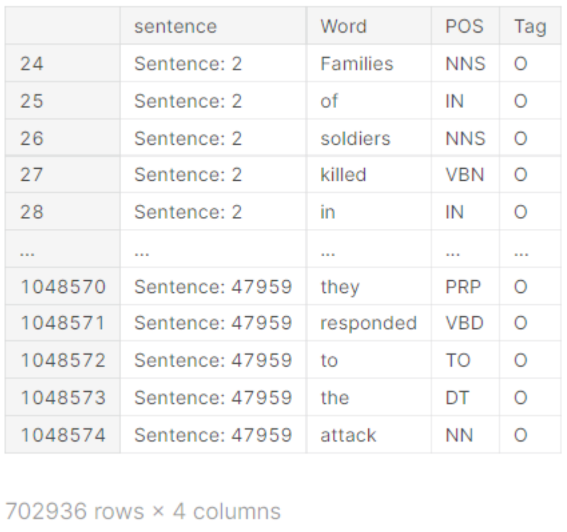
2.1.2 Splitting into test and train

We cannot split data normally with train_test_split because doing that makes some parts of a sentence in the training set while some others in the testing set. Instead, we use GroupShuffleSplit .



```
In [4]:
y = data.POS
X = data.drop('POS', axis=1)

gs = GroupShuffleSplit(n_splits=2, test_size=.33, random_state=42)
train_ix, test_ix = next(gs.split(X, y, groups=data['sentence']))

data_train = data.loc[train_ix]
data_test = data.loc[test_ix]

data_train
```

Figure 3: splitting into test and train

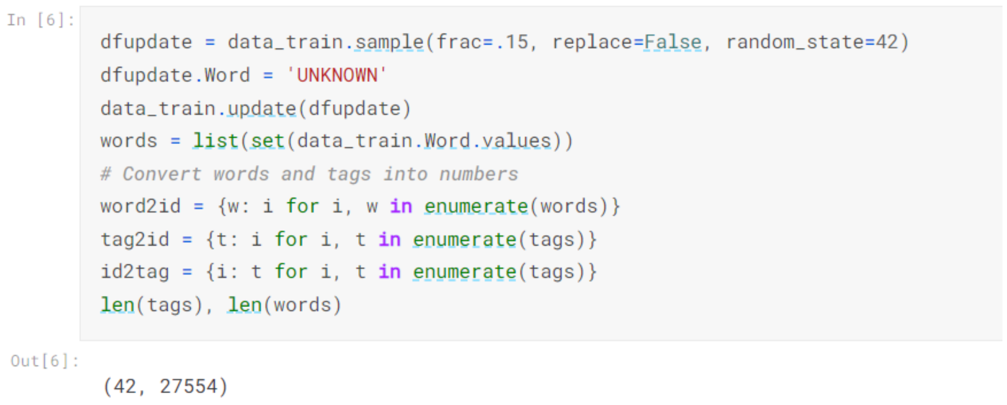


	sentence	Word	POS	Tag
24	Sentence: 2	Families	NNS	O
25	Sentence: 2	of	IN	O
26	Sentence: 2	soldiers	NNS	O
27	Sentence: 2	killed	VBN	O
28	Sentence: 2	in	IN	O
...	...	...	...	...
1048570	Sentence: 47959	they	PRP	O
1048571	Sentence: 47959	responded	VBD	O
1048572	Sentence: 47959	to	TO	O
1048573	Sentence: 47959	the	DT	O
1048574	Sentence: 47959	attack	NN	O

702936 rows × 4 columns

Figure 4: Training data after splitting

After splitting the data, we now can check the numbers of tags & words in the training set. The number of tags is enough but the number of words is not enough (~29k vs ~35k). Because of that we need to randomly add some UNKNOWN words into the training dataset then we recalculate the word list and create map from them to number.

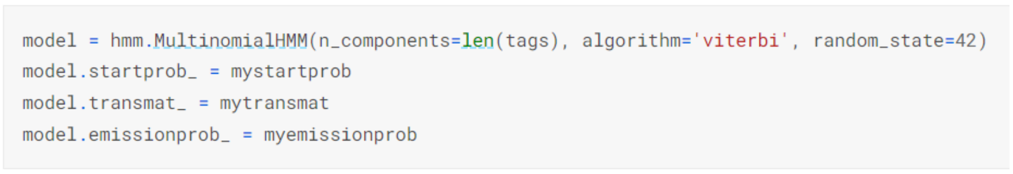


```
In [6]:
dfupdate = data_train.sample(frac=.15, replace=False, random_state=42)
dfupdate.Word = 'UNKNOWN'
data_train.update(dfupdate)
words = list(set(data_train.Word.values))
# Convert words and tags into numbers
word2id = {w: i for i, w in enumerate(words)}
tag2id = {t: i for i, t in enumerate(tags)}
id2tag = {i: t for i, t in enumerate(tags)}
len(tags), len(words)

Out[6]:
(42, 27554)
```

Figure 5: converting words into numbers

2.1.3 Applying HMM :

Hidden Markov Models can be trained by using the Baum-Welch algorithm. However input of the training is just dataset (Words). We cannot map back the states to the POS tag.That's why we have to calculate the model parameters for hmmlearn.hmm.MultinomialHMM manually by calculating startprob_,transmat_,emissionprob_ .

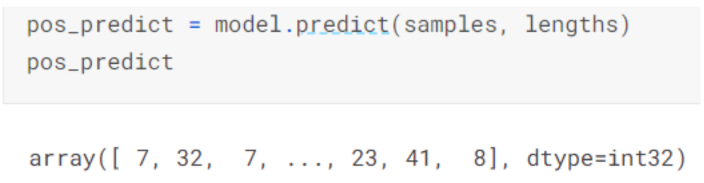


```
model = hmm.MultinomialHMM(n_components=len(tags), algorithm='viterbi', random_state=42)
model.startprob_ = mystartprob
model.transmat_ = mytransmat
model.emissionprob_ = myemissionprob
```

Figure 6: intialising HMM

As some words may never appear in the training set, we need to transform them into UNKNOWN first. Then we split data_test into samples & lengths and send them to HMM.

Often, when dealing with iterators, we also get need to keep a count of iterations. Python eases the programmers' task by providing a built-in function enumerate() for this task. Enumerate() method adds a counter to an iterable and returns it in a form of enumerating object. This enumerated object can then be used directly for loops or converted into a list of tuples using the list() method. we are now using enumerate for the samples to append all the word2id ,using for loop.

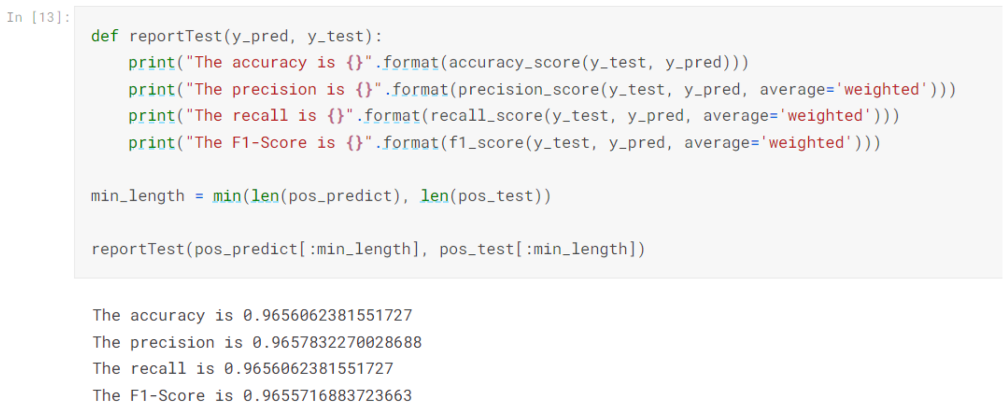


```
pos_predict = model.predict(samples, lengths)
pos_predict
```

```
array([ 7, 32,  7, ..., 23, 41,  8], dtype=int32)
```

Figure 7: predicting the pos model

3 Conclusion :



```
In [13]:
def reportTest(y_pred, y_test):
    print("The accuracy is {}".format(accuracy_score(y_test, y_pred)))
    print("The precision is {}".format(precision_score(y_test, y_pred, average='weighted')))
    print("The recall is {}".format(recall_score(y_test, y_pred, average='weighted')))
    print("The F1-Score is {}".format(f1_score(y_test, y_pred, average='weighted')))

min_length = min(len(pos_predict), len(pos_test))

reportTest(pos_predict[:min_length], pos_test[:min_length])
```

```
The accuracy is 0.9656062381551727
The precision is 0.9657832270028688
The recall is 0.9656062381551727
The F1-Score is 0.9655716883723663
```

Figure 8: Model accuracy and results

0.9656062381551727 is the accuracy we achieved .When we can not observe the state themselves but only the result of some probability function(observation) of the states we utilize HMM. HMM is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. Learning in HMMs involves estimating the state transition probabilities A and the output emission probabilities B that make an observed sequence most likely. Expectation-Maximization algorithms are used for this purpose. An algorithm is known as Baum-Welch algorithm, that falls under this category and uses the forward algorithm, is widely used .

4 References

[1] Prof. Changyou Chen. Lecture Slides

[2] https://github.com/cchangyou/hmmlearn/tree/main/examples

[3] https://web.stanford.edu/~jurafsky/slp3/A.pdf

[4] https://towardsdatascience.com/markov-and-hidden-markov-model-3eec42298d75#:~:text=Markov%20and%20Hidden%20Markov%20models,several%20(hidden)%20internal%20states.

[5] https://en.wikipedia.org/wiki/Hidden_Markov_model

[6] https://www.nature.com/articles/nbt1004-1315

[7] https://medium.com/@postsanjay/hidden-markov-models-simplified-c3f58728caab

Random Forest Report

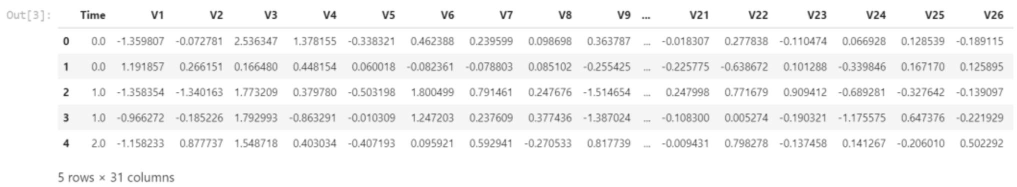
1 Random Forest

Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression.One of the most important features of the Random Forest Algorithm is that it can handle the data set containing continuous variables as in the case of regression and categorical variables as in the case of classification. It performs better results for classification problems.Random forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned. Random decision forests correct for decision trees' habit of overfitting to their training set. Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance.

2 Experiment

2.1 Dataset

For this task, we used the creditcard.csv dataset from Kaggle(
https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download).
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. The dataset 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-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

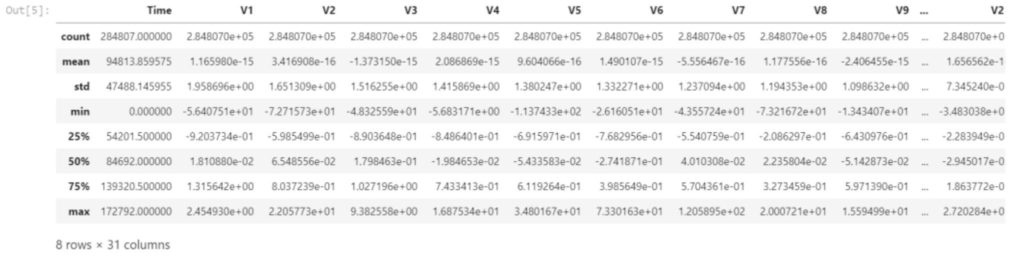


Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292

5 rows × 31 columns

Figure 1: Head of dataset



Out[5]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V2
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	...	2.848070e+0
mean	94813.859575	1.165980e-15	3.416908e-16	-1.373150e-15	2.086869e-15	9.604066e-16	1.490107e-15	-5.556467e-16	1.177556e-16	-2.406455e-15	...	1.656562e-1
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+00	...	7.345240e-0
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+01	...	-3.483038e+0
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-01	...	-2.283949e-0
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-02	...	-2.945017e-0
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-01	...	1.863772e-0
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+01	...	2.720284e+0

8 rows × 31 columns

Figure 2: Describe dataset

2.1.1 Data Engineering

For proceeding further, first we need to analyse the data. For this, we dealt with some missing values and null values. now, we need to replace the incomes with an integer that denotes its class . plotting some visualisations such as histogram of numerical columns to understand the data better and to check how the different columns correlate.

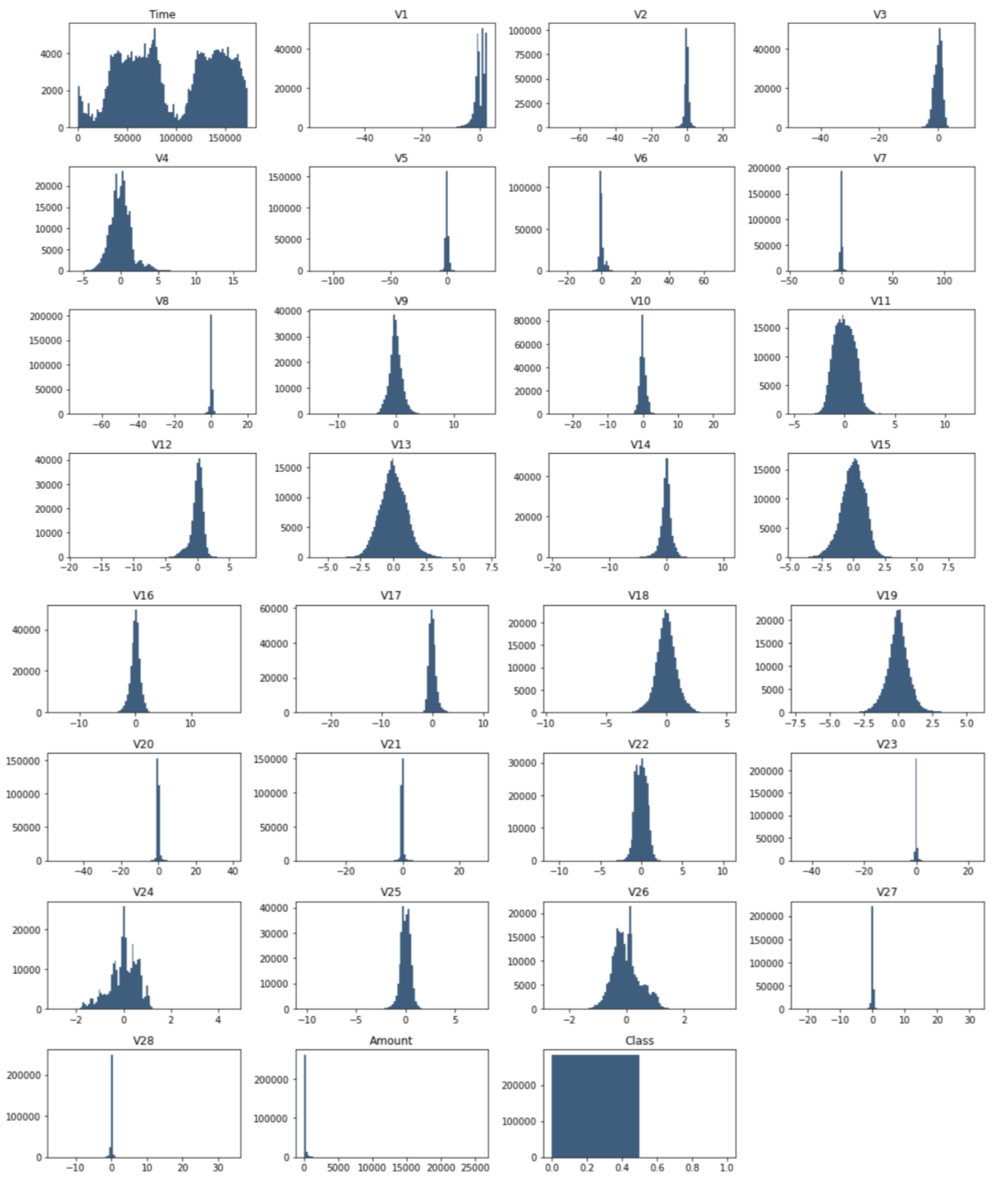
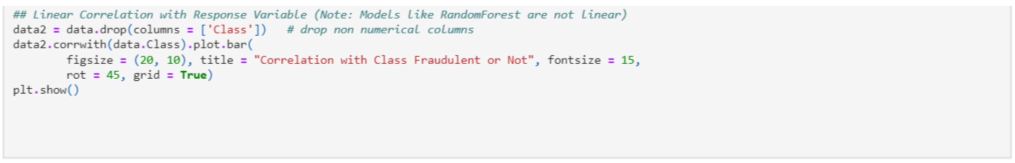


Figure 3: Histograms

Now, we can observe a large prevalence of Class 0 (non fraudulent).
checking for outliers in the data using the matplotlib boxplot function. We observed some outliers and after treating the outliers, we can see there are no outliers. Box plot after outlier treatment is shown in figure 3 below.



```
## Linear Correlation with Response Variable (Note: Models like RandomForest are not linear)
data2 = data.drop(columns = ['Class'])   # drop non numerical columns
data2.corrwith(data.Class).plot.bar(
        figsize = (20, 10), title = "Correlation with Class Fraudulent or Not", fontsize = 15,
        rot = 45, grid = True)
plt.show()
```

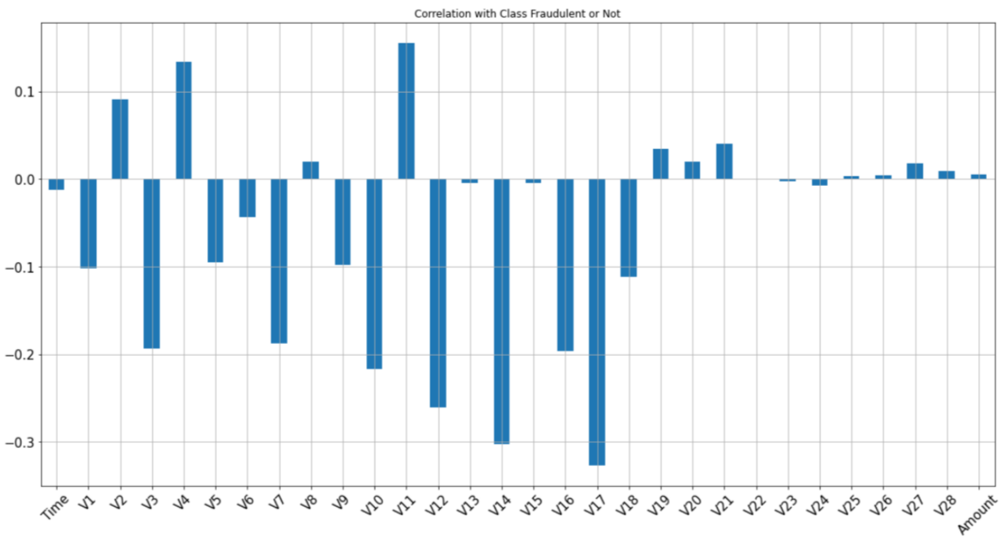
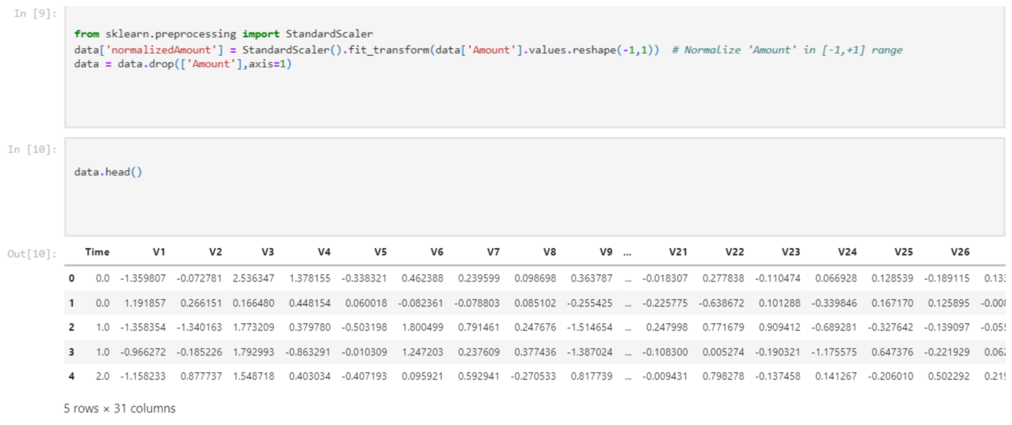


Figure 4: Checking for correlation



```
In [9]:

from sklearn.preprocessing import StandardScaler
data['normalizedAmount'] = StandardScaler().fit_transform(data['Amount'].values.reshape(-1,1))  # Normalize 'Amount' in [-1,+1] range
data = data.drop(['Amount'],axis=1)
```

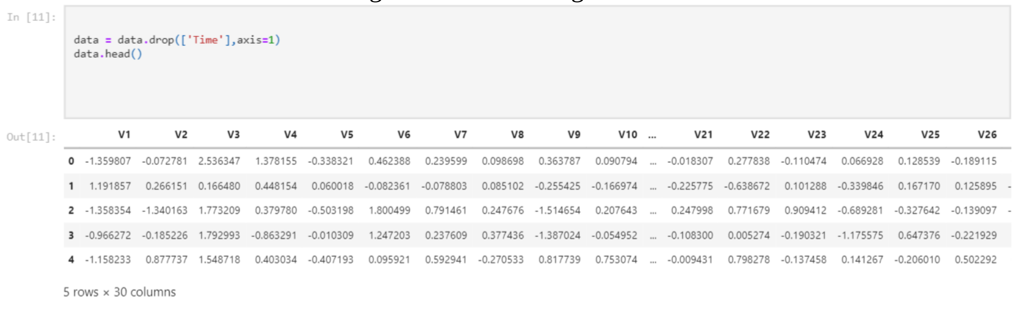
```
In [10]:

data.head()
```

Out[10]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.13
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.00
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.05
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.06
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.21

5 rows × 31 columns

Figure 5: Normalising the data



```
In [11]:

data = data.drop(['Time'],axis=1)
data.head()
```

Out[11]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V21	V22	V23	V24	V25	V26	
0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	0.090794	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	
1	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-
2	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-
3	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	-0.054952	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	
4	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	0.753074	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	

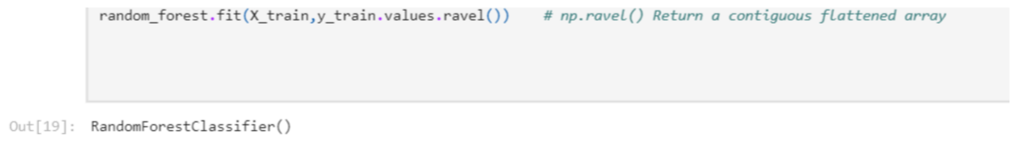
5 rows × 30 columns

Figure 6: Dropping Time variable

The next step before training the dataset is to perform feature scaling using standard scalar fit transform function. Now that the data is scaled, the dataset is split into training data and testing data(with test_size = 0.3, random_state=0).

2.1.2 Applying Randomforest Classifier :

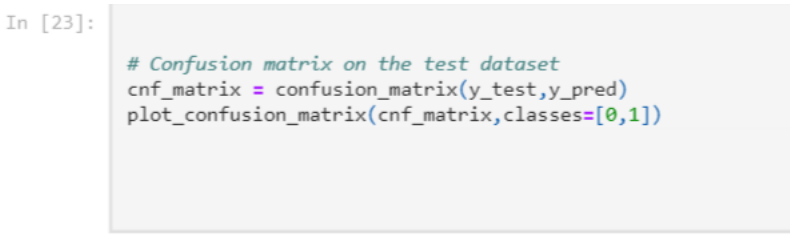
From sklearn.ensemble import RandomForestClassifier and created RandomForestClassifier with n_estimators=100.



```
random_forest.fit(X_train,y_train.values.ravel())    # np.ravel() Return a contiguous flattened array

Out[19]:  RandomForestClassifier()
```

Figure 7: Fitting RandomForestClassifier

Creating Confusion matrix on the test dataset created earlier.



```
In [23]:

# Confusion matrix on the test dataset
cnf_matrix = confusion_matrix(y_test,y_pred)
plot_confusion_matrix(cnf_matrix,classes=[0,1])
```

```
Confusion matrix, without normalization
[[85290     6]
 [   33   114]]
```

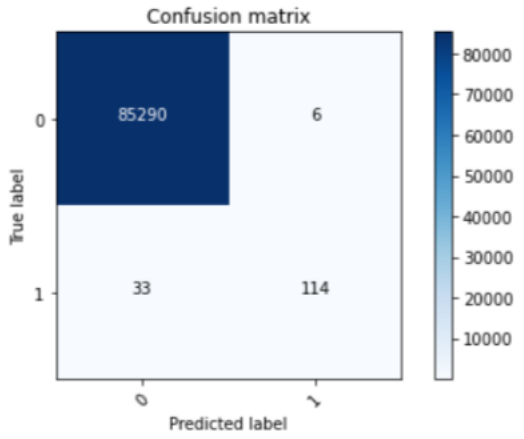
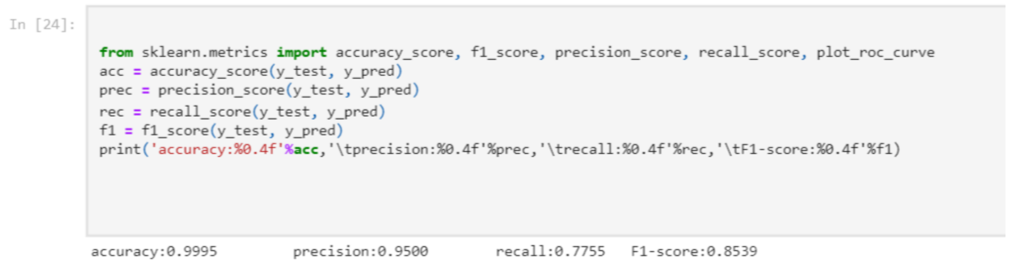


Figure 8: Confusion matrix on the test dataset

From the above matrix created, we can say that while only 6 regular transactions are wrongly predicted as fraudulent, the model only detects 78% of the fraudulent transactions. As a consequence 33 fraudulent transactions are not detected (False Negatives).



```
In [24]:

from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score, plot_roc_curve
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
print('accuracy:%0.4f'%acc,'\tprecision:%0.4f'%prec,'\trecall:%0.4f'%rec,'\tF1-score:%0.4f'%f1)

accuracy:0.9995         precision:0.9500        recall:0.7755    F1-score:0.8539
```

Figure 9: accuracy obtained on the test dataset

```
In [26]:

ROC_RF = plot_roc_curve(random_forest, X_test, y_test)
plt.show()
```

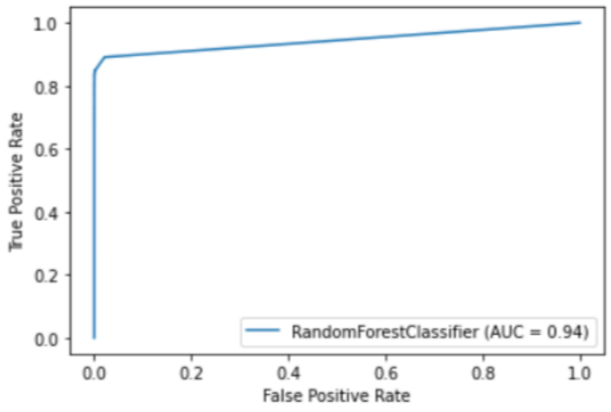


Figure 10: Roc Curve on test data

```
y_pred = random_forest.predict(X)
cnf_matrix = confusion_matrix(y,y_pred.round())
plot_confusion_matrix(cnf_matrix,classes=[0,1])
```

```
Confusion matrix, without normalization
[[284309      6]
 [    34    458]]
```

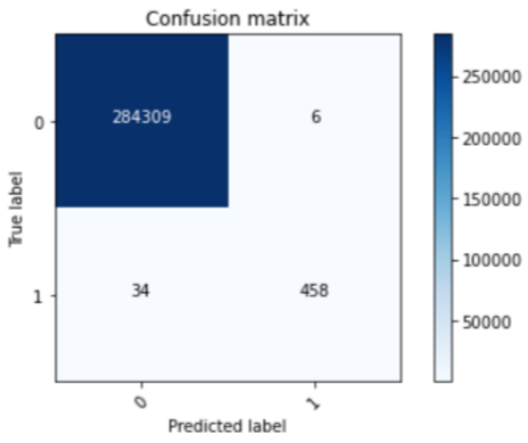


Figure 11: Confusion matrix on the full dataset

```
In [28]:

acc = accuracy_score(y, y_pred)
prec = precision_score(y, y_pred)
rec = recall_score(y, y_pred)
f1 = f1_score(y, y_pred)
print('accuracy:%0.4f'%acc,'\tprecision:%0.4f'%prec,'\trecall:%0.4f'%rec,'\tF1-score:%0.4f'%f1)

accuracy:0.9999         precision:0.9871         recall:0.9309    F1-score:0.9582
```

Figure 12: accuracy obtained on full dataset

3 Conclusion

The model of creditcard csv patients was developed successfully using Randomforest classifier. Also, the Accuracy for the test data set achieved is 0.9995 and for fulldataset achieved was 0.9999 .Now, we can conclude that Random Forest is one of the best techniques with high performance which is widely used in various industries for its efficiency. It can handle binary, continuous, and categorical data.Random forest is a great choice if anyone wants to build the model fast and efficiently as one of the best things about the random forest is it can handle missing values.

4 References

[1] Prof. Changyou Chen. Lecture Slides

[2] https://github.com/cchangyou/100-Days-Of-ML-Code/blob/master/Code/Day%2034%20Random_Forest.md

[3] https://builtin.com/data-science/random-forest-algorithm

[4] https://en.wikipedia.org/wiki/Random_forest

[5] https://www.ibm.com/cloud/learn/random-forest

[6] https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[7] https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download

AdaBoost Classifier Report

1 AdaBoost Classifier

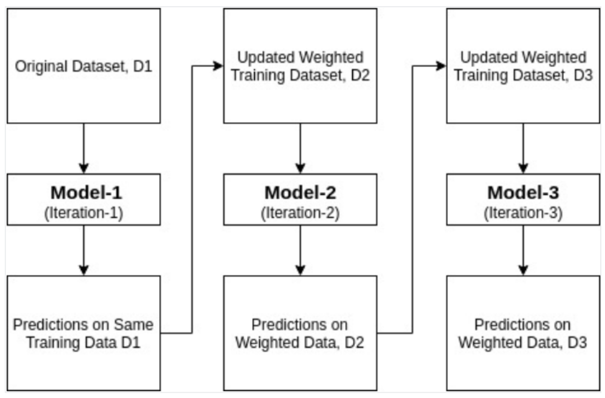
An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

Adaptive Boosting is one of ensemble boosting classifier proposed by Yoav Freund and Robert Schapire in 1996. It combines multiple classifiers to increase the accuracy of classifiers. AdaBoost is an iterative ensemble method. AdaBoost classifier builds a strong classifier by combining multiple poorly performing classifiers so that you will get high accuracy strong classifier. The basic concept behind Adaboost is to set the weights of classifiers and training the data sample in each iteration such that it ensures the accurate predictions of unusual observations. Any machine learning algorithm can be used as base classifier if it accepts weights on the training set. Adaboost should meet two conditions:

1. The classifier should be trained interactively on various weighed training examples.
2. In each iteration, it tries to provide an excellent fit for these examples by minimizing training error.

It works in the following steps:

1. Initially, Adaboost selects a training subset randomly.
2. It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.
3. It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.
4. Also, It assigns the weight to the trained classifier in each iteration according to the accuracy of the classifier. The more accurate classifier will get high weight.
5. This process iterate until the complete training data fits without any error or until reached to the specified maximum number of estimators.
6. To classify, perform a "vote" across all of the learning algorithms you built.

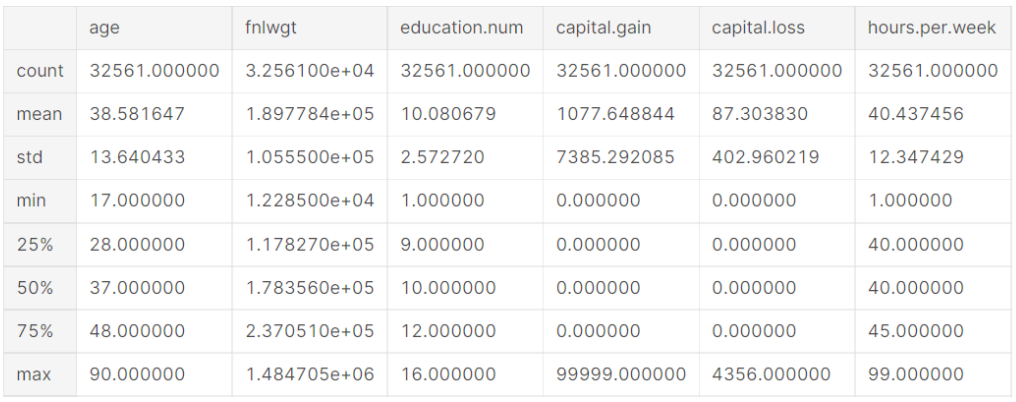

Original Dataset, D1
Updated Weighted Training Dataset, D2
Updated Weighted Training Dataset, D3
Model-1 (Iteration-1)
Model-2 (Iteration-2)
Model-3 (Iteration-3)
Predictions on Same Training Data D1
Predictions on Weighted Data, D2
Predictions on Weighted Data, D3

AdaBoost is easy to implement. It iteratively corrects the mistakes of the weak classifier and improves accuracy by combining weak learners. You can use many base classifiers with AdaBoost. AdaBoost is not prone to overfitting. This can be found out via experiment results, but there is no concrete reason available.

2 Experiment

2.1 Dataset

This data was extracted from the 1994 Census bureau database by Ronny Kohavi and Barry Becker (Data Mining and Visualization, Silicon Graphics). A set of reasonably clean records was extracted using the following conditions: ((AAGE>16) && (AGI>100) && AFNLWGT>1) && (HRSWK>0)). The prediction task is to determine whether a person makes over $50K a year. It has 15 columns in which 9 are string and 6 are integer . age ,workclass ,fnlwgt ,education ,education.num ,marital.status ,occupation ,relationship ,race ,sex ,capital.gain ,capital.loss ,hours.per.week ,native.country ,income are the columns of dataset.
This dataset is taken from Kaggle - https://www.kaggle.com/datasets/uciml/adult-census-income.



	age	fnlwgt	education.num	capital.gain	capital.loss	hours.per.week
count	32561.000000	3.256100e+04	32561.000000	32561.000000	32561.000000	32561.000000
mean	38.581647	1.897784e+05	10.080679	1077.648844	87.303830	40.437456
std	13.640433	1.055500e+05	2.572720	7385.292085	402.960219	12.347429
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.178270e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.783560e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.370510e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.484705e+06	16.000000	99999.000000	4356.000000	99.000000

Figure 1: dataset

2.1.1 Data Engineering

For proceeding further, first we need to analyse the data. For this, we dealt with some missing values and null values. now, we need to replace the incomes with an integer that denotes its class . plotting some visualisations to understand the data better and to check how the different columns correlate with the income .

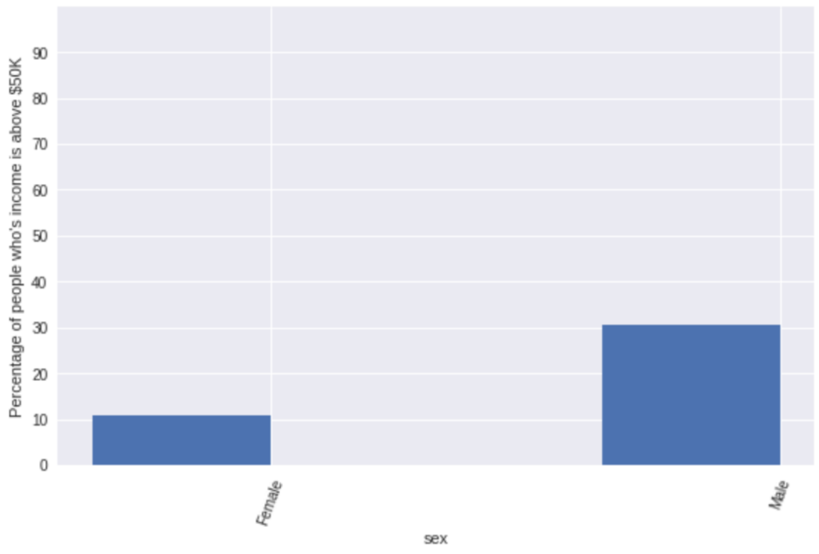


Figure 2: income vs sex

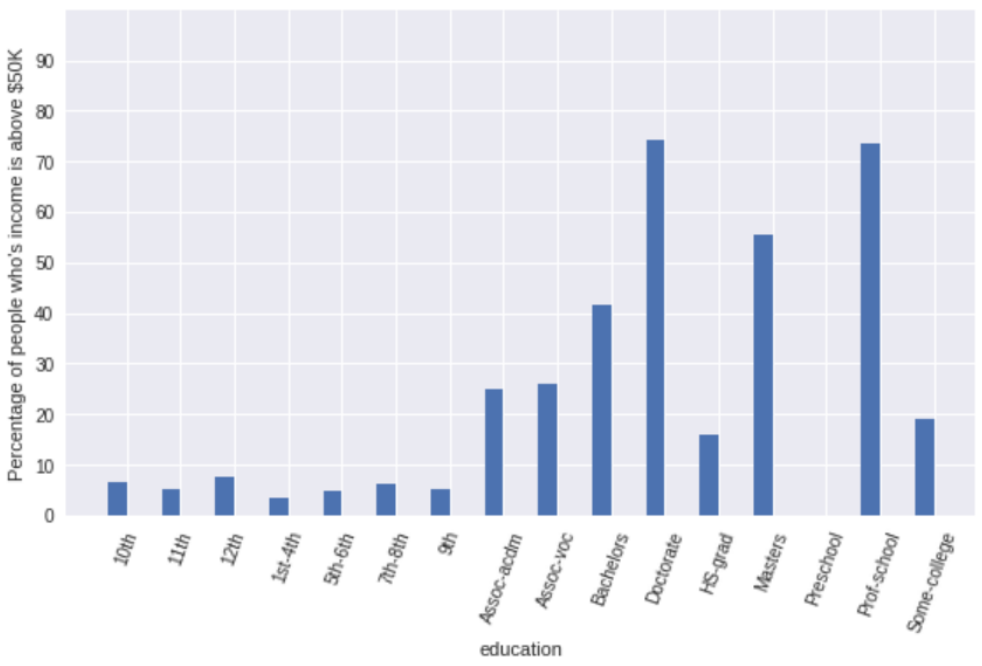


Figure 3: income vs education

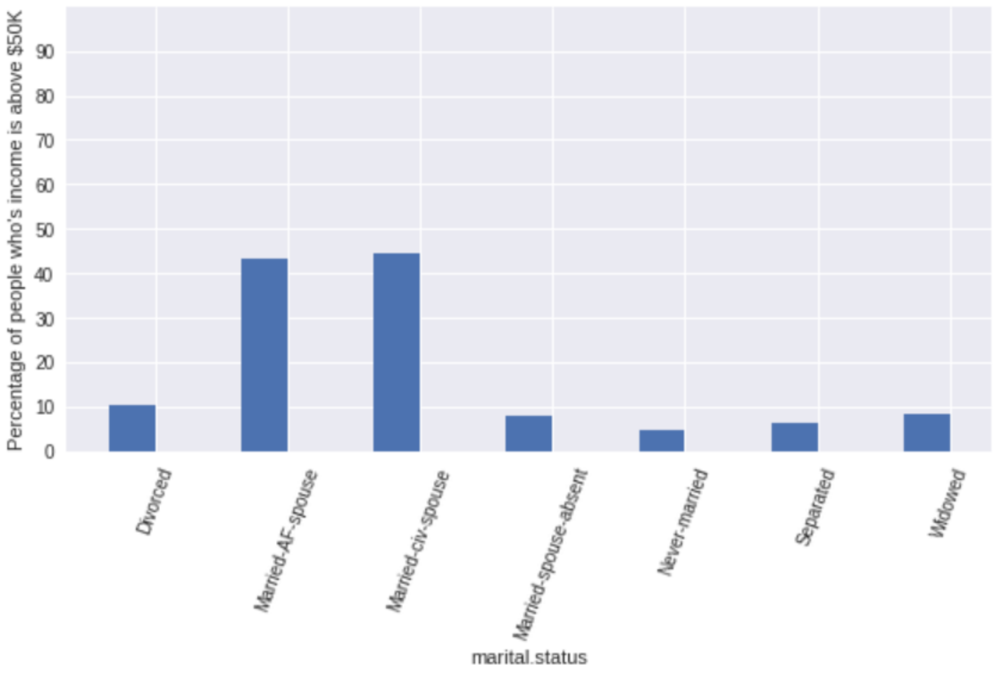


Figure 4: income vs maritial status

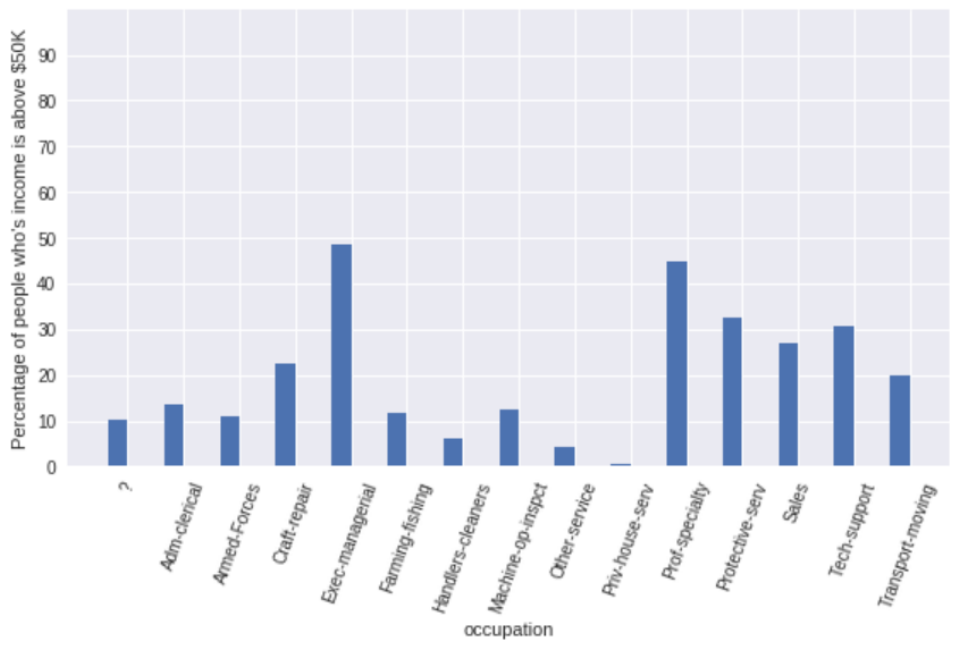
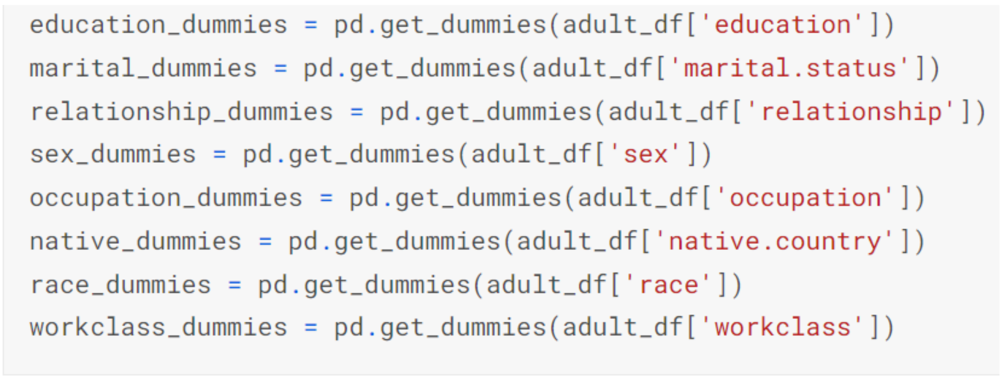


Figure 5: income vs occupation

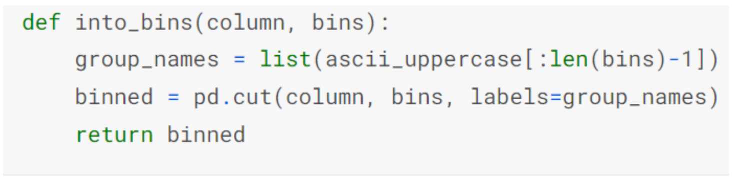
2.1.2 Implementation and Evaluation

The data in the dataset has no null values or Na's, and it ranges from 0-10, so to normalise the features of the data, feature scaling is done using min-max normalisation and this rescaled data is present in the range 0-1. This rescaled data is used for Adaptive boosting for the classification task. 1.Initially, Adaboost selects a training subset randomly . It iteratively trains the AdaBoost machine learning model by selecting the training set based on the accurate prediction of the last training.It assigns the higher weight to wrong classified observations so that in the next iteration these observations will get the high probability for classification.Also, It assigns the weight to the trained classifier in each iteration according to the accuracy of the classifier. The more accurate classifier will get high weight.This process iterate until the complete training data fits without any error or until reached to the specified maximum number of estimators.To classify, perform a "vote" across all of the learning algorithms you built.



```
education_dummies = pd.get_dummies(adult_df['education'])
marital_dummies = pd.get_dummies(adult_df['marital.status'])
relationship_dummies = pd.get_dummies(adult_df['relationship'])
sex_dummies = pd.get_dummies(adult_df['sex'])
occupation_dummies = pd.get_dummies(adult_df['occupation'])
native_dummies = pd.get_dummies(adult_df['native.country'])
race_dummies = pd.get_dummies(adult_df['race'])
workclass_dummies = pd.get_dummies(adult_df['workclass'])
```

Figure 6: Creating dummies for each variable



```
def into_bins(column, bins):
    group_names = list(ascii_uppercase[:len(bins)-1])
    binned = pd.cut(column, bins, labels=group_names)
    return binned
```

Figure 7: Converting continuous values into bins

```
unique = sorted(adult_df['capital.loss'].unique())
plt.scatter(range(len(unique)), unique)
plt.ylabel('Capital Loss')
plt.tick_params(axis='x', which='both', labelbottom='off', bottom='off') # disable x ticks
plt.show()
```

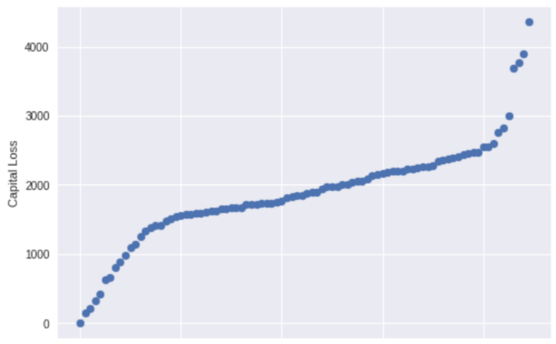


Figure 8: creating a scatter plot of all the unique values in capital.loss which will be helpful in visualizing how to assign bins to this feature

```
loss_bins = into_bins(adult_df['capital.loss'], list(range(-1, 4500, 500)))
loss_dummies = pd.get_dummies(loss_bins)
```

Figure 9: created bins from -1 to 4500, with 500 values in each bin

```
unique = sorted(adult_df['capital.gain'].unique())
plt.scatter(range(len(unique)), unique)
plt.ylabel('Capital Gain')
plt.tick_params(axis='x', which='both', labelbottom='off', bottom='off') # disable x ticks
plt.show()
```

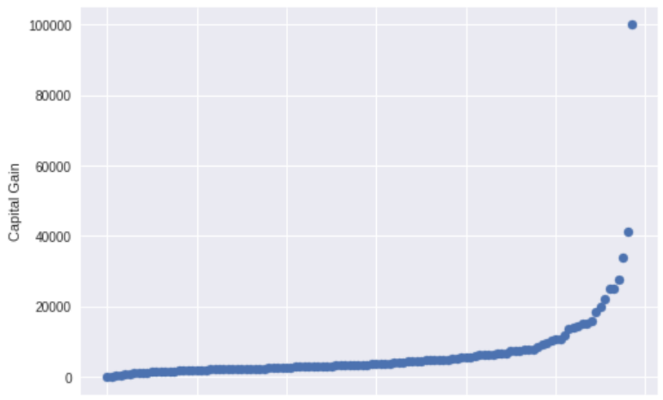
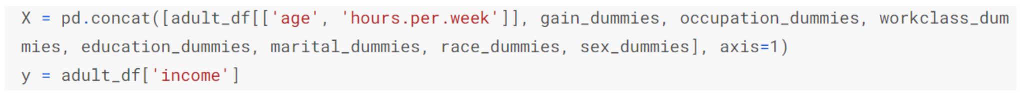


Figure 10: creating a scatter plot of all the unique values in capital.gain which will be helpful in visualizing how to assign bins to this feature

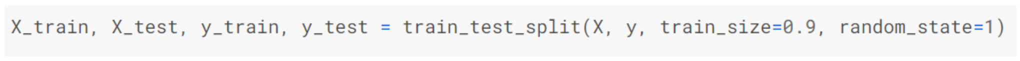
```
gain_bins = into_bins(adult_df['capital.gain'], list(range(-1, 42000, 5000)) + [100000])
gain_dummies = pd.get_dummies(gain_bins)
```

Figure 11: created bins from -1 to 42000, with 5000 values in each bin and an extra one for outlier



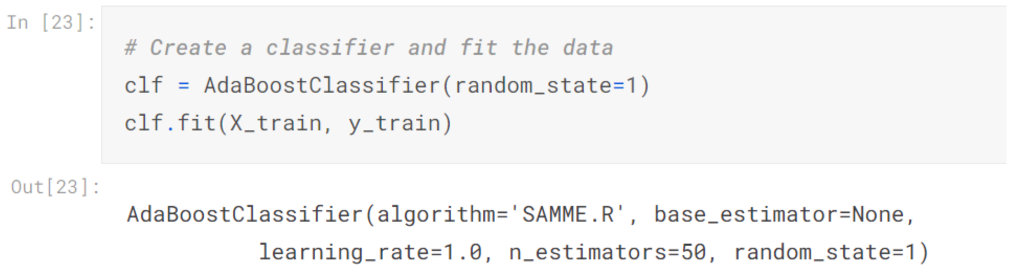
```
X = pd.concat([adult_df[['age', 'hours.per.week']], gain_dummies, occupation_dummies, workclass_dum
mies, education_dummies, marital_dummies, race_dummies, sex_dummies], axis=1)
y = adult_df['income']
```

Figure 12: concatenated all the columns we need and the ones we generated by binning and creating dummies



```
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.9, random_state=1)
```

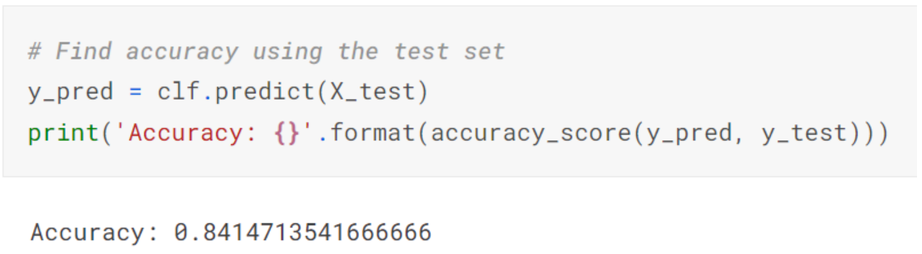
Figure 13: Created test and train sets



```
In [23]:  # Create a classifier and fit the data
          clf = AdaBoostClassifier(random_state=1)
          clf.fit(X_train, y_train)

Out[23]:  AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
                    learning_rate=1.0, n_estimators=50, random_state=1)
```

Figure 14: creating Adaboost classifier and fitting the data

3 Conclusion



```
# Find accuracy using the test set
y_pred = clf.predict(X_test)
print('Accuracy: {}'.format(accuracy_score(y_pred, y_test)))

Accuracy: 0.8414713541666666
```

Figure 15: Results and Accuracy

0.8414713541666666 is the accuracy we achieved .we can conclude that Achieves higher performance than bagging when hyper-parameters tuned properly. Can be used for classification and regression equally well , Can use "robust" loss functions that make the model resistant to outliers. Remember that AdaBoost fits a set of weak classifiers. To see what this means in practice, let us visualise the error rates for each of the stumps we trained above

4 References

[1] Prof. Changyou Chen. Lecture Slides
[2] https://github.com/cchangyou/adaboost-implementation
[3] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html
[4] Y. Freund, R. Schapire, “A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting”, 1995.
[5] Zhu, H. Zou, S. Rosset, T. Hastie, “Multi-class AdaBoost”, 2009
[6] https://www.kaggle.com/datasets/uciml/adult-census-income
[7] https://towardsdatascience.com/adaboost-from-scratch-37a936da3d50

Autoencoder Report

1 Autoencoder

An autoencoder is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning). The encoding is validated and refined by attempting to regenerate the input from the encoding. The autoencoder learns a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore insignificant data “noise”. Autoencoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (ie. correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck.

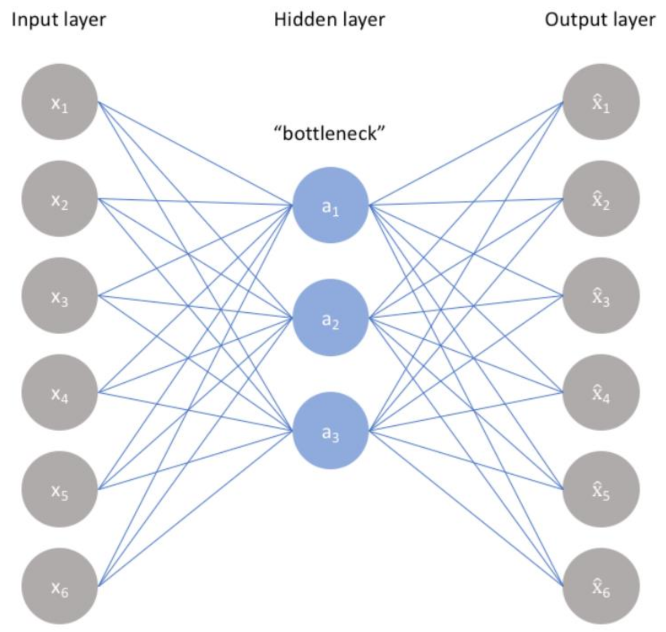


Figure 1: Autoencoder1

As visualized above, we can take an unlabeled dataset and frame it as a supervised learning problem tasked with outputting x^, a reconstruction of the original input x. This network can be trained by minimizing the reconstruction error, L(x,x^), which measures the differences between our original input and the consequent reconstruction. The bottleneck is a key attribute of our network design; without the presence of an information bottleneck, our network could easily learn to simply memorize the input values by passing these values along through the network (visualized below)

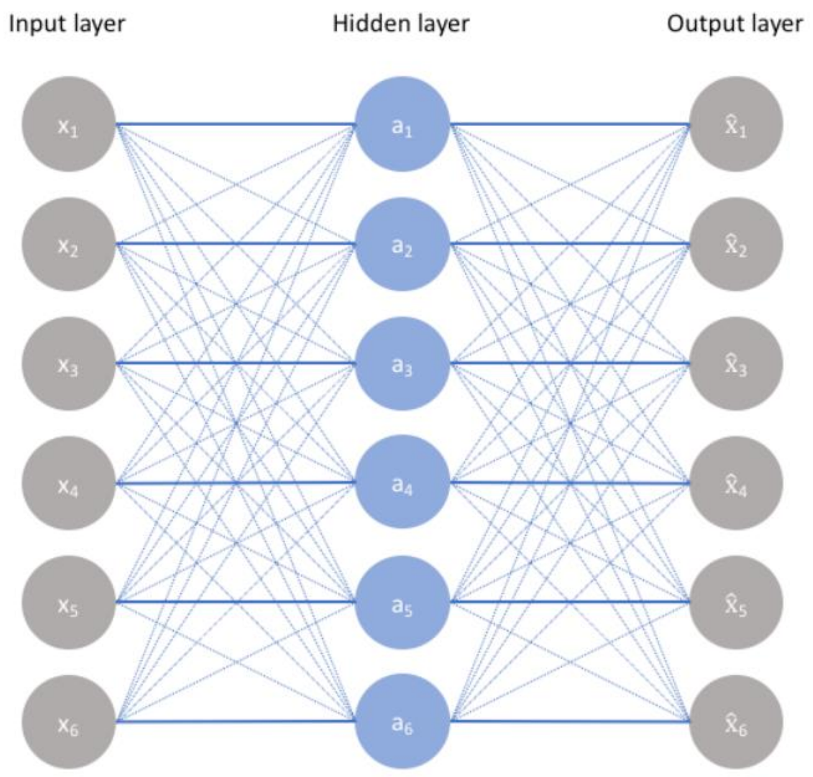


Figure 2: Autoencoder2

A bottleneck constrains the amount of information that can traverse the full network, forcing a learned compression of the input data.

Autoencoder Components:

Autoencoders consists of 4 main parts:
1- Encoder: In which the model learns how to reduce the input dimensions and compress the input data into an encoded representation.
2- Bottleneck: which is the layer that contains the compressed representation of the input data. This is the lowest possible dimensions of the input data.
3- Decoder: In which the model learns how to reconstruct the data from the encoded representation to be as close to the original input as possible.
4- Reconstruction Loss: This is the method that measures measure how well the decoder is performing and how close the output is to the original input.

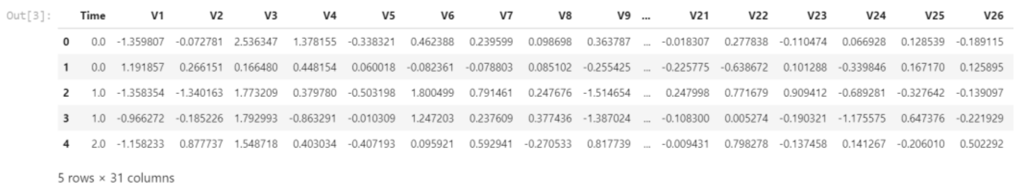
The training then involves using back propagation in order to minimize the network’s reconstruction loss.

2 Experiment

2.1 Dataset

For this task, we used the creditcard.csv dataset from Kaggle(
https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download).
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. The dataset contains transactions made by credit cards in September 2013 by European cardholders.

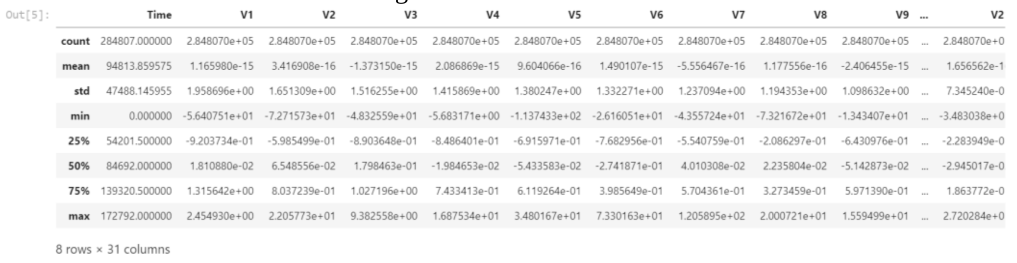
This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. It contains only numerical input variables which are the result of a PCA transformation. 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-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.


Out[3]:

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292

5 rows × 31 columns

Figure 3: Head of dataset


Out[5]:

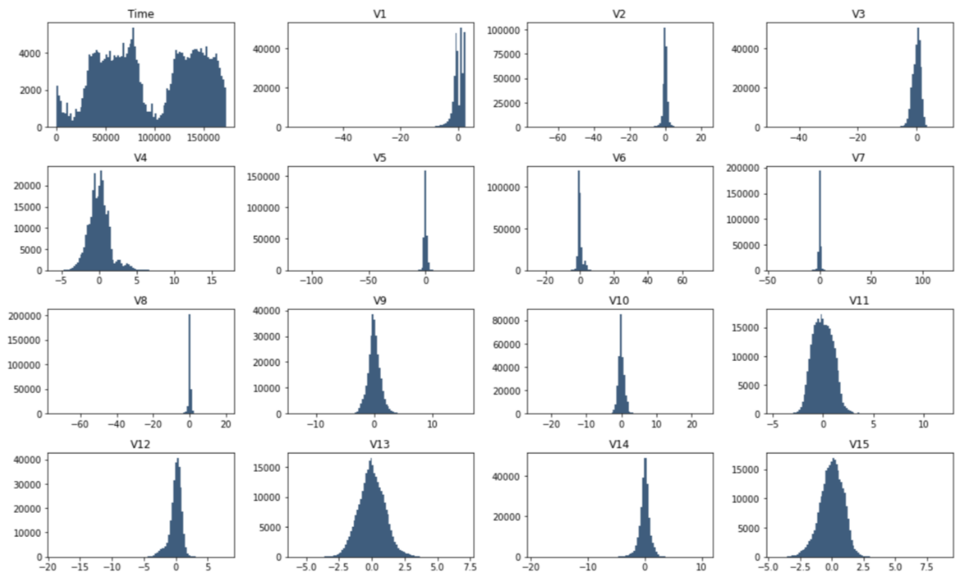
	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V2
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	...	2.848070e+0
mean	94813.859575	1.165980e-15	3.416908e-16	-1.373150e-15	2.086869e-15	9.604066e-16	1.490107e-15	-5.556467e-16	1.177556e-16	-2.406455e-15	...	1.656562e-1
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00	1.098632e+00	...	7.345240e-0
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01	-1.343407e+01	...	-3.483038e+0
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01	-6.430976e-01	...	-2.283949e-0
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02	-5.142873e-02	...	-2.945017e-0
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01	5.971390e-01	...	1.863772e-0
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01	1.559499e+01	...	2.720284e+0

8 rows × 31 columns

Figure 4: Describe dataset

2.1.1 Data Engineering

For proceeding further, first we need to analyse the data. For this, we dealt with some missing values and null values. now, we need to replace the incomes with an integer that denotes its class . plotting some visualisations such as histogram of numerical columns to understand the data better and to check how the different columns correlate.



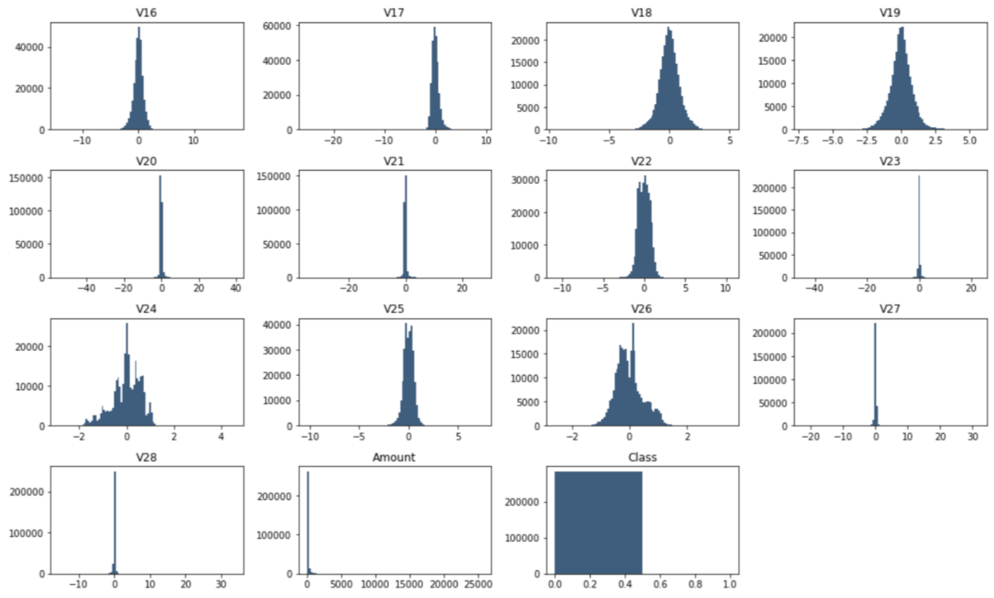
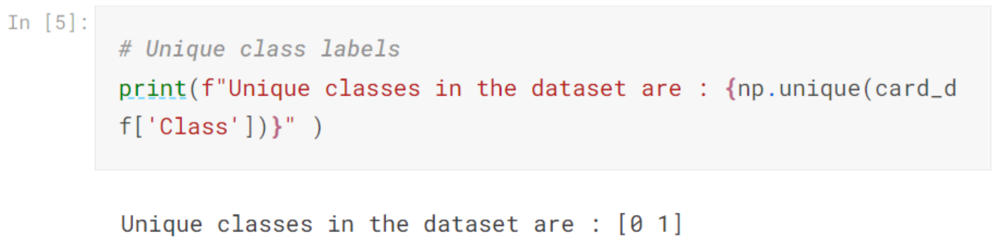


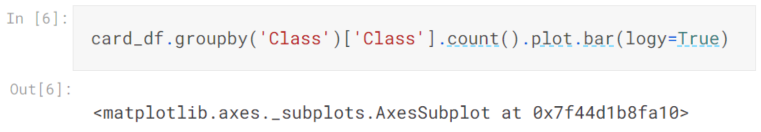
Figure 6: Histograms

Now, we can observe a large prevalence of Class 0 (non fraudulent).
checking for outliers in the data using the matplotlib boxplot function. We observed some outliers and after treating the outliers, we can see there are no outliers. Box plot after outlier treatment is shown in figure 3 below.



```
In [5]:
# Unique class labels
print(f"Unique classes in the dataset are : {np.unique(card_d
f['Class'])}" )

Unique classes in the dataset are : [0 1]
```

Figure 7: Checking for Unique class labels



```
In [6]: card_df.groupby('Class')['Class'].count().plot.bar(logy=True)

Out[6]:
<matplotlib.axes._subplots.AxesSubplot at 0x7f44d1b8fa10>
```

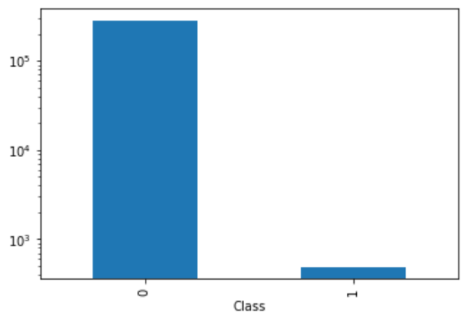
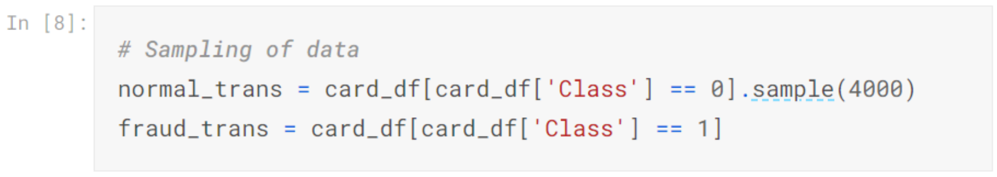


Figure 8: Plotting unique class labels

Data transformation is one of the steps in data processing. We need to transform certain attributes value so that it makes sense in the further analysis. we have performed few basic transformations like changing the time attribute in day.



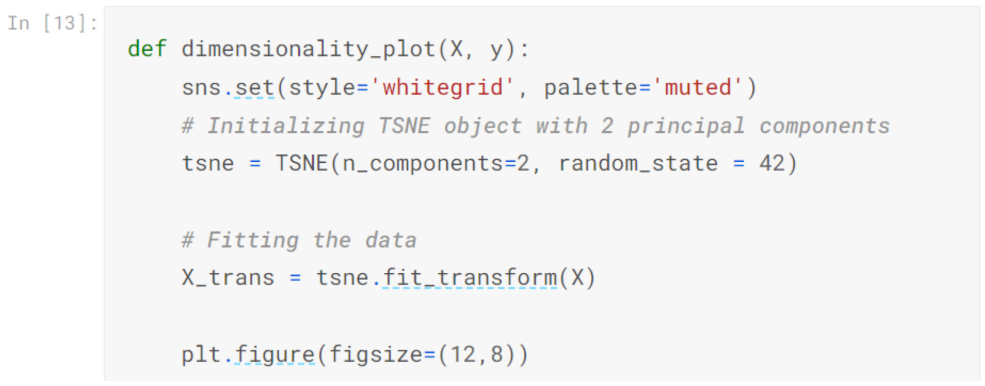
```
In [8]:
# Sampling of data
normal_trans = card_df[card_df['Class'] == 0].sample(4000)
fraud_trans = card_df[card_df['Class'] == 1]
```

Figure 9: Performing Sampling

The next step before training the dataset is to perform feature scaling using standard scalar fit transform function. Now that the data is scaled, the dataset is split into training data and testing data(with test_size = 0.3, random_state=0).

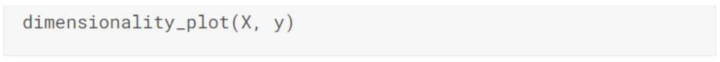
Visualising the data with t-SNE :

TNSE(t-distributed Stochastic Neighbor Embedding) is one of the dimensionality reduction method other than PCA and SVD. This will supress some noise and speed up the computation of pairwise distance between samples.



```
In [13]:
def dimensionality_plot(X, y):
    sns.set(style='whitegrid', palette='muted')
    # Initializing TSNE object with 2 principal components
    tsne = TSNE(n_components=2, random_state = 42)

    # Fitting the data
    X_trans = tsne.fit_transform(X)

    plt.figure(figsize=(12,8))
```

Figure 10: Fitting the data



```
dimensionality_plot(X, y)
```

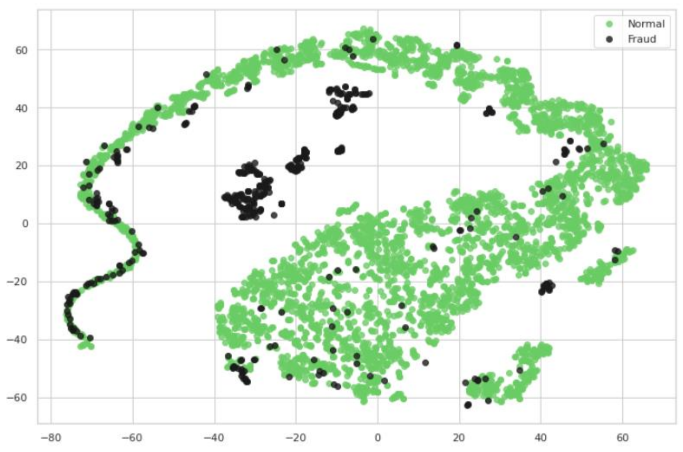
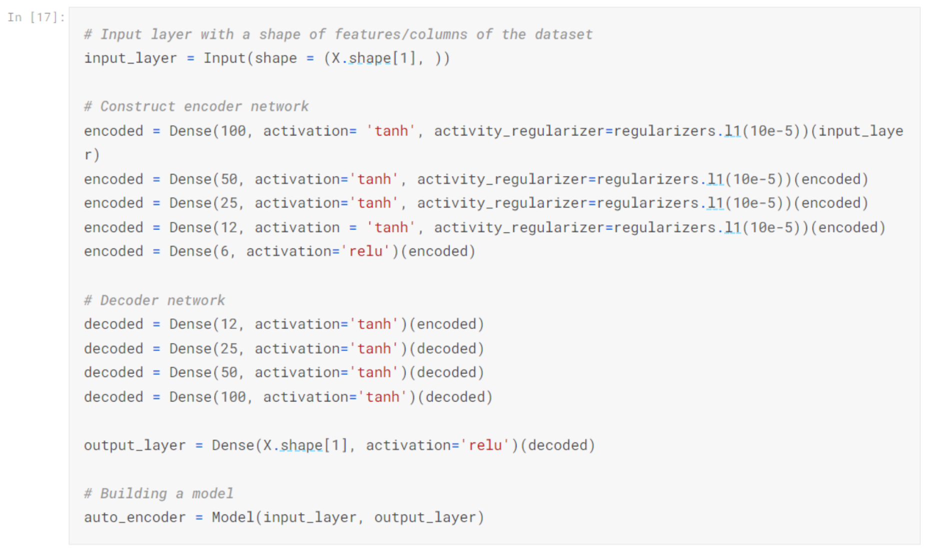


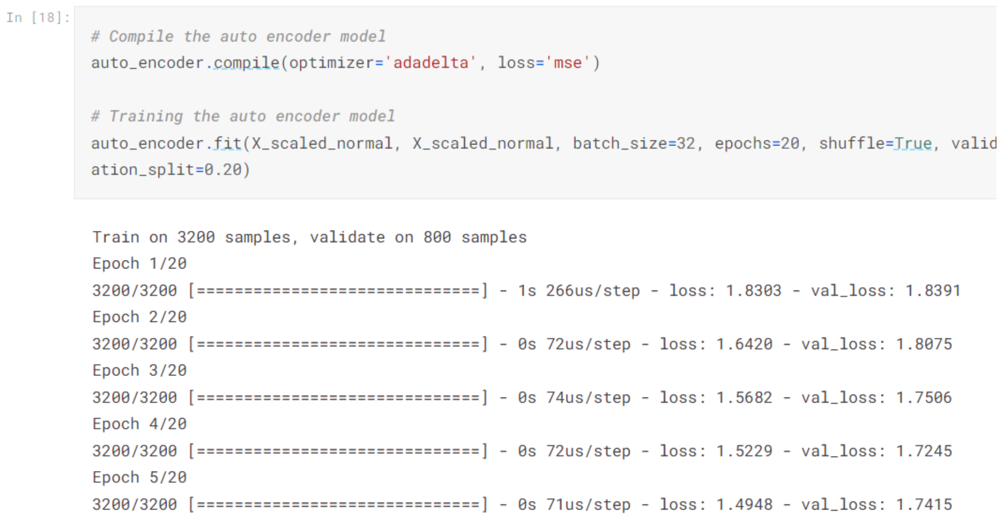
Figure 11: dimensionality plot

2.1.2 Applying Autoencoder :



```
In [17]:
# Input layer with a shape of features/columns of the dataset
input_layer = Input(shape = (X.shape[1], ))

# Construct encoder network
encoded = Dense(100, activation= 'tanh', activity_regularizer=regularizers.l1(10e-5))(input_laye
r)
encoded = Dense(50, activation='tanh', activity_regularizer=regularizers.l1(10e-5))(encoded)
encoded = Dense(25, activation='tanh', activity_regularizer=regularizers.l1(10e-5))(encoded)
encoded = Dense(12, activation = 'tanh', activity_regularizer=regularizers.l1(10e-5))(encoded)
encoded = Dense(6, activation='relu')(encoded)

# Decoder network
decoded = Dense(12, activation='tanh')(encoded)
decoded = Dense(25, activation='tanh')(decoded)
decoded = Dense(50, activation='tanh')(decoded)
decoded = Dense(100, activation='tanh')(decoded)

output_layer = Dense(X.shape[1], activation='relu')(decoded)

# Building a model
auto_encoder = Model(input_layer, output_layer)
```

Figure 12: Construct encoder network

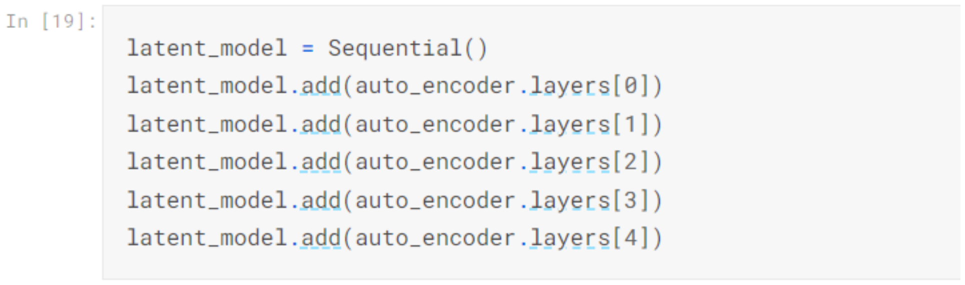


```
In [18]:
# Compile the auto encoder model
auto_encoder.compile(optimizer='adadelta', loss='mse')

# Training the auto encoder model
auto_encoder.fit(X_scaled_normal, X_scaled_normal, batch_size=32, epochs=20, shuffle=True, valid
ation_split=0.20)
```

```
Train on 3200 samples, validate on 800 samples
Epoch 1/20
3200/3200 [==============================] - 1s 266us/step - loss: 1.8303 - val_loss: 1.8391
Epoch 2/20
3200/3200 [==============================] - 0s 72us/step - loss: 1.6420 - val_loss: 1.8075
Epoch 3/20
3200/3200 [==============================] - 0s 74us/step - loss: 1.5682 - val_loss: 1.7506
Epoch 4/20
3200/3200 [==============================] - 0s 72us/step - loss: 1.5229 - val_loss: 1.7245
Epoch 5/20
3200/3200 [==============================] - 0s 71us/step - loss: 1.4948 - val_loss: 1.7415
```

Figure 13: Training auto encoder model

Denoising or noise reduction is the process of removing noise from a signal. This can be an image, audio or a document. You can train an Autoencoder network to learn how to remove noise from pictures. In order to try out this use case, let’s re-use the famous MNIST dataset and let’s create some synthetic noise in the dataset.



```
In [19]:
latent_model = Sequential()
latent_model.add(auto_encoder.layers[0])
latent_model.add(auto_encoder.layers[1])
latent_model.add(auto_encoder.layers[2])
latent_model.add(auto_encoder.layers[3])
latent_model.add(auto_encoder.layers[4])
```

Figure 14: Using Autoencode to encode data

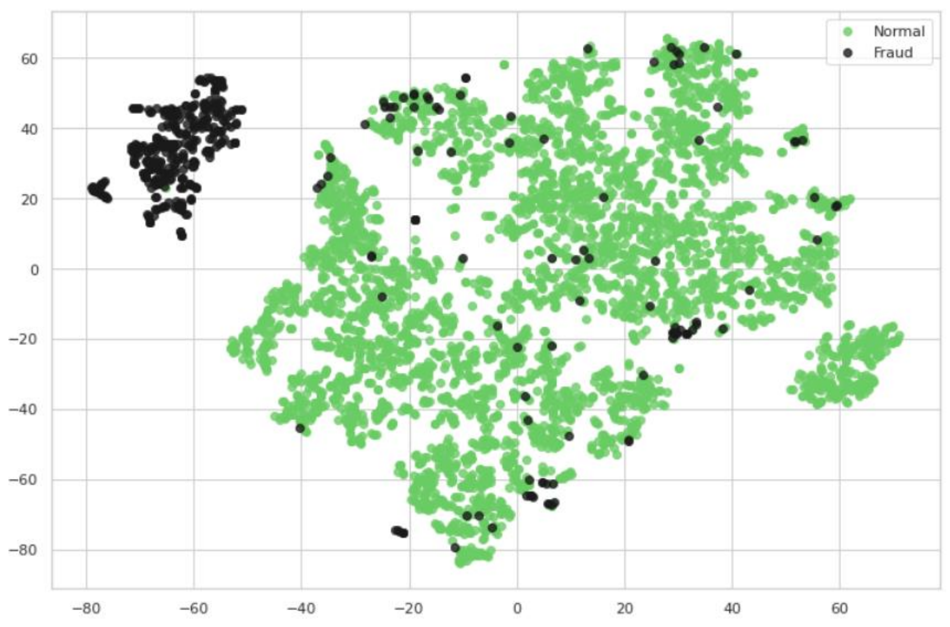


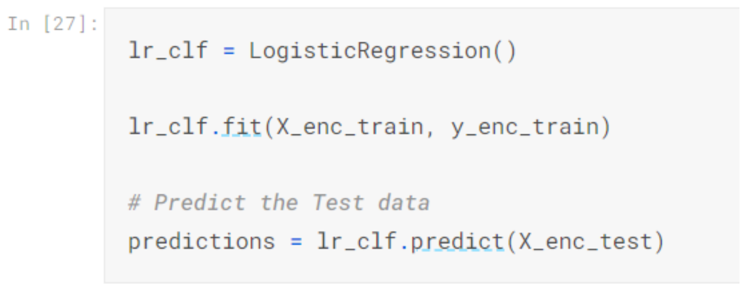
Figure 15: TSNE plot function using encoded data

We can observe that the encoded fraud data points have been moved towards one cluster, whereas there are only few fraud transaction datapoints are there among the normal transaction data points.

```
Classification report
              precision    recall  f1-score   support

           0       0.92      1.00      0.96      1201
           1       1.00      0.31      0.48       147

    accuracy                           0.93      1348
   macro avg       0.96      0.66      0.72      1348
weighted avg       0.93      0.93      0.91      1348
```

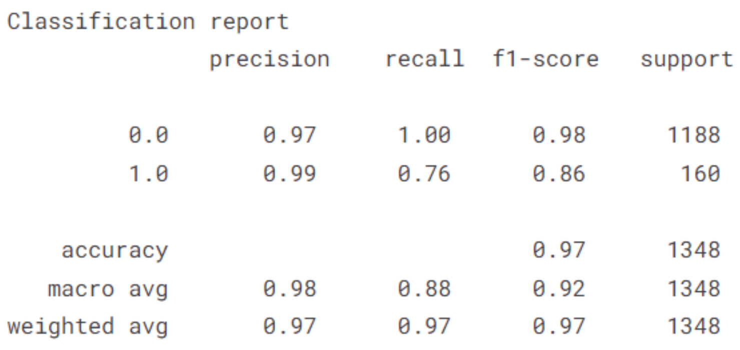
Figure 17: Non-linear Classifier Report

Now let's apply linear classifier to classify the data and observe the result. We will use Logistic Regression to build the model



```
In [27]:
    lr_clf = LogisticRegression()

    lr_clf.fit(X_enc_train, y_enc_train)

    # Predict the Test data
    predictions = lr_clf.predict(X_enc_test)
```

Figure 18: Linear Classifier



```
Classification report
              precision    recall  f1-score   support

         0.0       0.97      1.00      0.98      1188
         1.0       0.99      0.76      0.86       160

    accuracy                           0.97      1348
   macro avg       0.98      0.88      0.92      1348
weighted avg       0.97      0.97      0.97      1348
```

Figure 19: Linear Classifier Report

3 Conclusion

The model of creditcard csv patients was developed successfully using Autoencoders. Also, the Accuracy for the nonlinear classifier obtained is 0.93 and for Linear Classifier is 0.97 . Autoencoder networks teach themselves how to compress data from the input layer into a shorter code, and then uncompress that code into whatever format best matches the original input. This process sometimes involves multiple autoencoders, such as stacked sparse autoencoder layers used in image processing.

4 References

[1] Prof. Changyou Chen. Lecture Slides
[2] https://github.com/udacity/deep-learning-v2-pytorch/tree/master/autoencoder
[3] https://www.tensorflow.org/tutorials/generative/autoencoder
[4] https://www.sciencedirect.com/topics/engineering/autoencoder
[5] https://towardsdatascience.com/auto-encoder-what-is-it-and-what-is-it-used-for-part-1-3e5c6f017726
[6] https://en.wikipedia.org/wiki/Autoencoder
[7] https://www.jeremyjordan.me/autoencoders/