Sanjay Mahto

Final Report:
Bitcoin Address Analysis

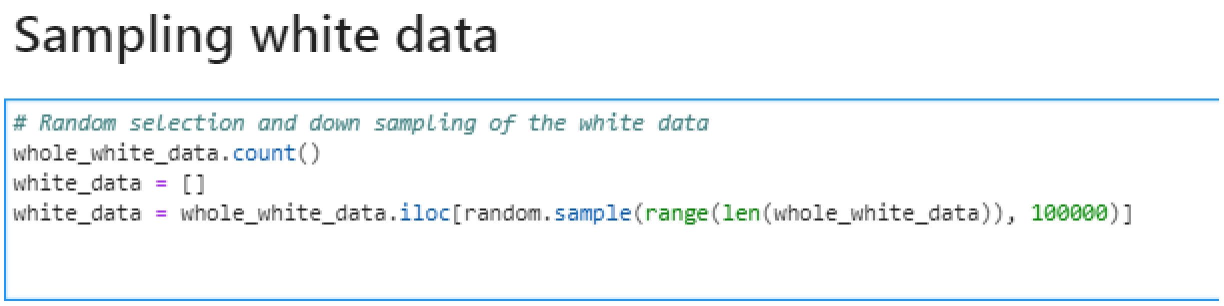
Problem Statement

In the recent world the use of crypto currency has increased significantly and with that increased the scams and cyber-attacks using these crypto currency as a medium. In this project I have taken the most popular crypto currency and tried to predict if the bitcoin address is suspicious or if it could be the part of ransomware attacks.

The data was taken from UCI Machine learning website which is openly available for the general public for the analysis. This would be really helpful for the users who are using bitcoin for the transactions. They would be able to verify if the bitcoin address is suspicious or not.

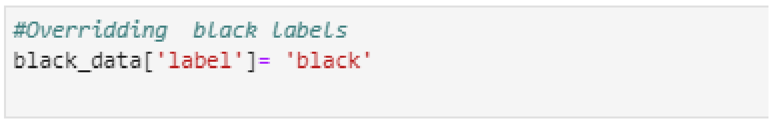
Data Wrangling

The acquired dataset consists over 10 Million data points and 8 distinguish features (address, year, length, weight, count, looped, neighbors, income, label) of the bitcoin address. In the raw dataset we had only 30k suspected data (black data) and over 10 Million clean (white data). To normalize the ratio between black data and white data we have only considered 100k white data in random and all of black data to reduce the future bias on the model


Sampling white data

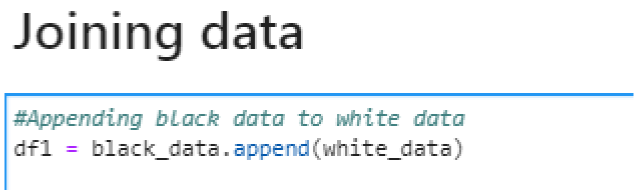
```
# Random selection and down sampling of the white data
whole_white_data.count()
white_data = []
white_data = whole_white_data.iloc[random.sample(range(len(whole_white_data)), 100000)]
```

Additionally the suspected bitcoin address were divided into the various labels which specified the ransomware attacks name in which the bitcoin address were used. Since we are making a tool to provide basic information to the user we have mapped all the label to black. So we shall consider all the address used in ransomware attacks as black irrespective of attacks name.



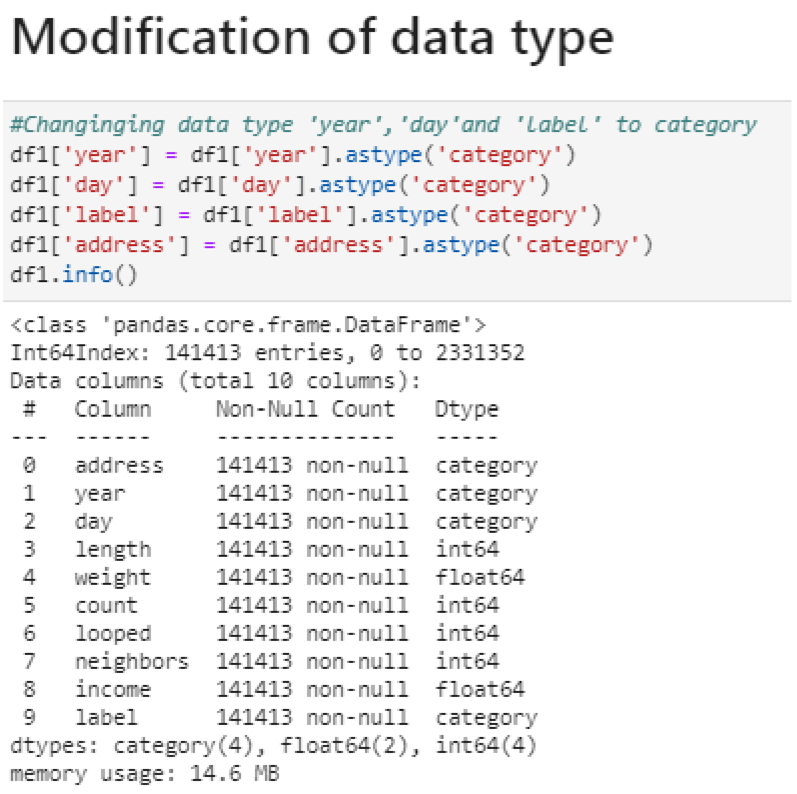
```
#Overridding  black labels
black_data['label']= 'black'
```

After that we have to append our white and black data to get the dataset to analyses and model upon.


Joining data

```
#Appending black data to white data
df1 = black_data.append(white_data)
```

After we have our final dataset we should prepare it for the data analysis. The data types of the columns has been modified to desired type and at last the dataset is ready for the analysis



Modification of data type

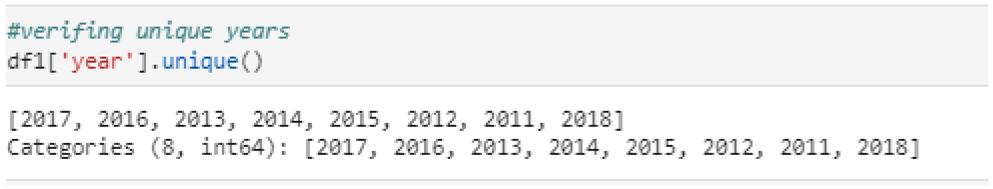
```
#Changinging data type 'year','day'and 'label' to category
df1['year'] = df1['year'].astype('category')
df1['day'] = df1['day'].astype('category')
df1['label'] = df1['label'].astype('category')
df1['address'] = df1['address'].astype('category')
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 141413 entries, 0 to 2331352
Data columns (total 10 columns):
 #   Column     Non-Null Count   Dtype
---  ------     --------------   -----
 0   address    141413 non-null  category
 1   year       141413 non-null  category
 2   day        141413 non-null  category
 3   length     141413 non-null  int64
 4   weight     141413 non-null  float64
 5   count      141413 non-null  int64
 6   looped     141413 non-null  int64
 7   neighbors  141413 non-null  int64
 8   income     141413 non-null  float64
 9   label      141413 non-null  category
dtypes: category(4), float64(2), int64(4)
memory usage: 14.6 MB
```

Exploratory Data Analysis

Exploratory Data Analysis (EDA is one of the crucial step in machine learning. It helps us to understand the data in deeper level and provide the insight on the data points of the dataset and the relation between them.

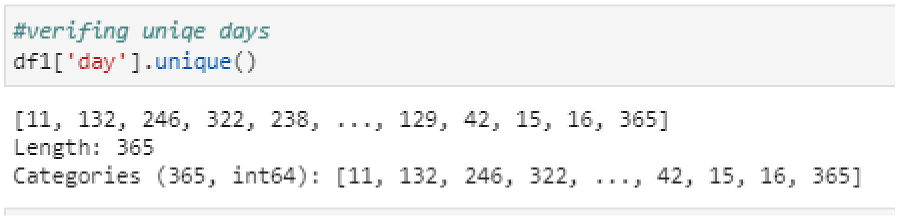
In our case we have started our analysis to check the range of years for which the data exist. We got to know that the data is from 2013 to 2018



```
#verifing unique years
df1['year'].unique()
```

```
[2017, 2016, 2013, 2014, 2015, 2012, 2011, 2018]
Categories (8, int64): [2017, 2016, 2013, 2014, 2015, 2012, 2011, 2018]
```

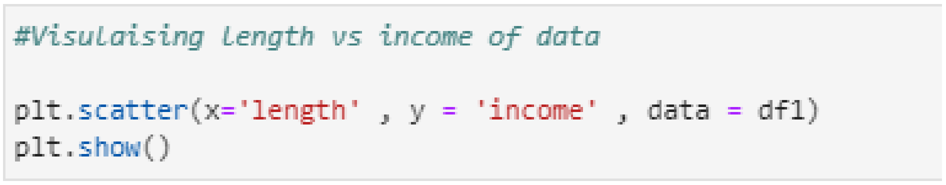
We also had to verify the anomaly in the day's numbers to see if number of days are between 1 to 365. After verifying we can confirm we have 365 unique value from 1 to 365.



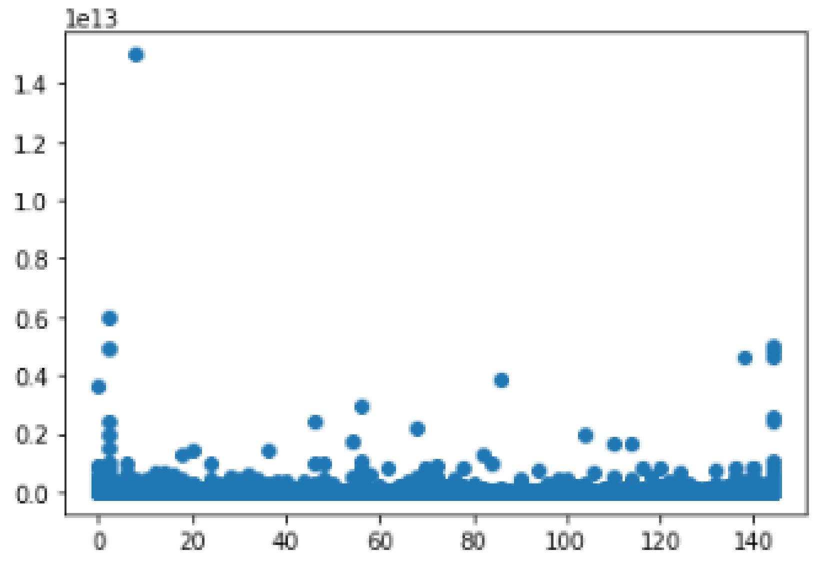
```
#verifing uniqe days
df1['day'].unique()
```

```
[11, 132, 246, 322, 238, ..., 129, 42, 15, 16, 365]
Length: 365
Categories (365, int64): [11, 132, 246, 322, ..., 42, 15, 16, 365]
```

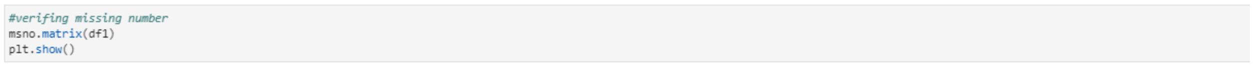
Next we have visualized the relation between length of the address and the income of the address. We can very well say that there is no apparent relation between these two columns as the income has been spread evenly for the all length of bitcoin address.



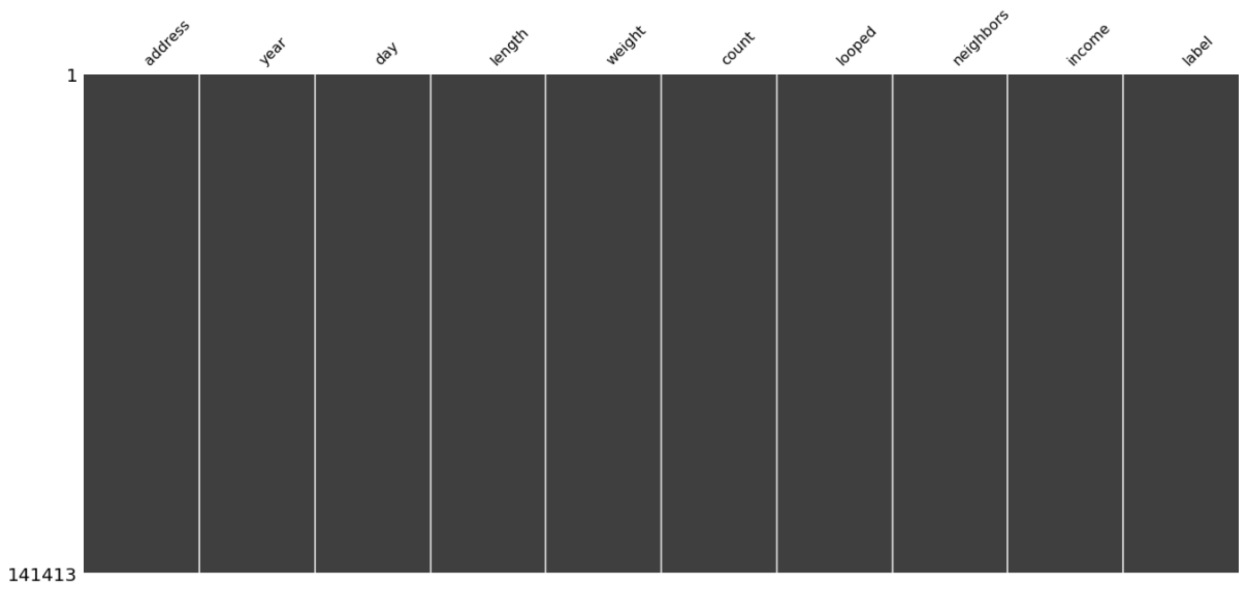
```
#Visulaising length vs income of data

plt.scatter(x='length' , y = 'income' , data = df1)
plt.show()
```



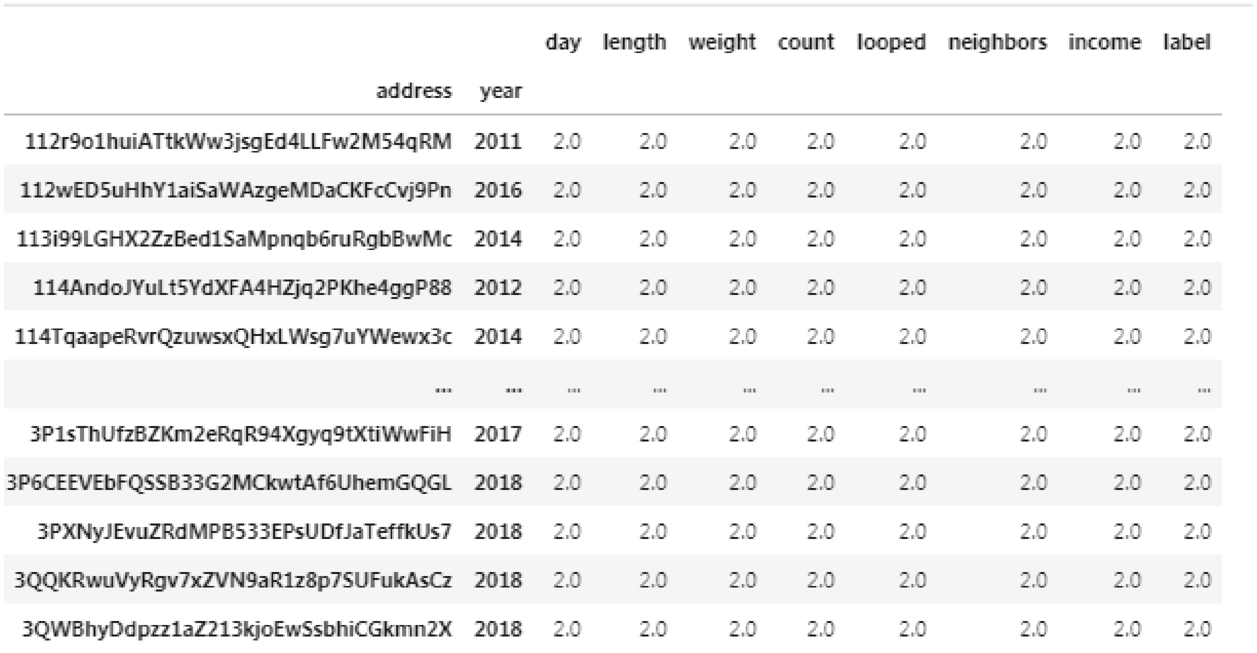
Next we will visualize if we have any null data among the columns or not. We can clearly see there is no null data in the dataset.



```
#verifing missing number
msno.matrix(df1)
plt.show()
```

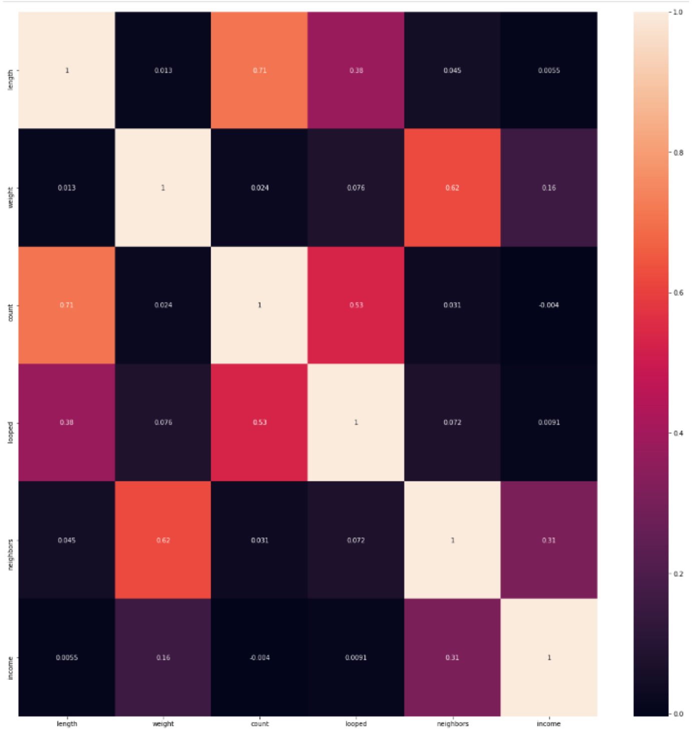


Next we will visualize the duplicate data in our dataset with respect to address and year. That provided us the intuition that same address has been repeated in the data but it is mostly on two different transections so we can use this data for analysis.

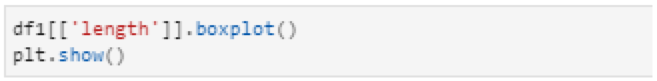


		day	length	weight	count	looped	neighbors	income	label
address	year								
112r9o1huiATtkWw3jsgEd4LLFw2M54qRM	2011	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
112wED5uHhY1aiSaWAzgeMDaCKFcCvj9Pn	2016	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
113i99LGHX2ZzBed1SaMpnqb6ruRgbBwMc	2014	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
114AndoJYuLt5YdXFA4HZjq2PKhe4ggP88	2012	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
114TqaapeRvrQzuwsxQHxLWsg7uYWewx3c	2014	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
...	...	...	...	...	...	...	...	...	...
3P1sThUfzBZKm2eRqR94Xgyq9tXtiWwFiH	2017	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
3P6CEEVEbFQSSB33G2MCkwtAf6UhemGQGL	2018	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
3PXNyJEvuZRdMPB533EPsUDfJaTeffkUs7	2018	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
3QQKRwuVyRgv7xZVN9aR1z8p7SUFukAsCz	2018	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
3QWBhyDdpzz1aZ213kjoEwSsbhiCGkmn2X	2018	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0

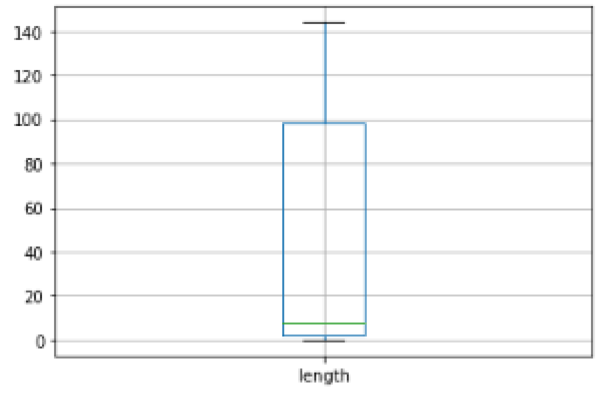
By looking at the below heat map of the correlation matrix of the columns we clearly see that length and count are closely related to each other but others are not



The below boxplot showed us that the length of the bitcoin address varied from 2 to 140+ while the mean is around 10



```
df1[['length']].boxplot()
plt.show()
```



Now we have the deeper understanding of the data and we would be using these understanding in the model selection and preprocessing of the data in further steps.

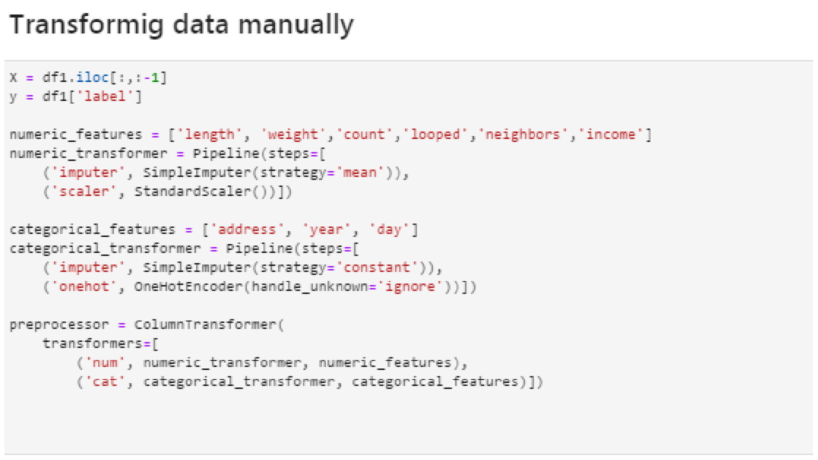
Feature Engineering

We have used two different ways for the feature engineering in this project.

- Manual Feature Engineering
- Automated Feature Engineering

Manual feature Engineering

In the manual feature engineering, I have specified the Numerical features and the categorical features in a list and by using the pipeline numerical features has been imputed with mean strategy and normalized using standard scalar, and the categorical features has been imputed with constant strategy with onehotencodr which would give us the sparse matrix.

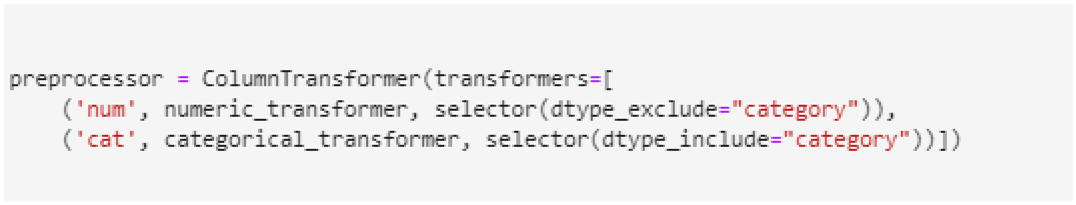


Transformig data manually

```
X = df1.iloc[:,:-1]
y = df1['label']

numeric_features = ['length', 'weight','count','looped','neighbors','income']
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())])

categorical_features = ['address', 'year', 'day']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])
```

Automatic Feature engineering

In the automatic feature engineering the preprocessing has been done with the function “ColumnTransformer” which does not require any numerical or categorical features to be specified separately. It can I dandify the column types and pass it to the data to the pipeline as we have done it in manual feature engineering step



```
preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, selector(dtype_exclude="category")),
    ('cat', categorical_transformer, selector(dtype_include="category"))])
```

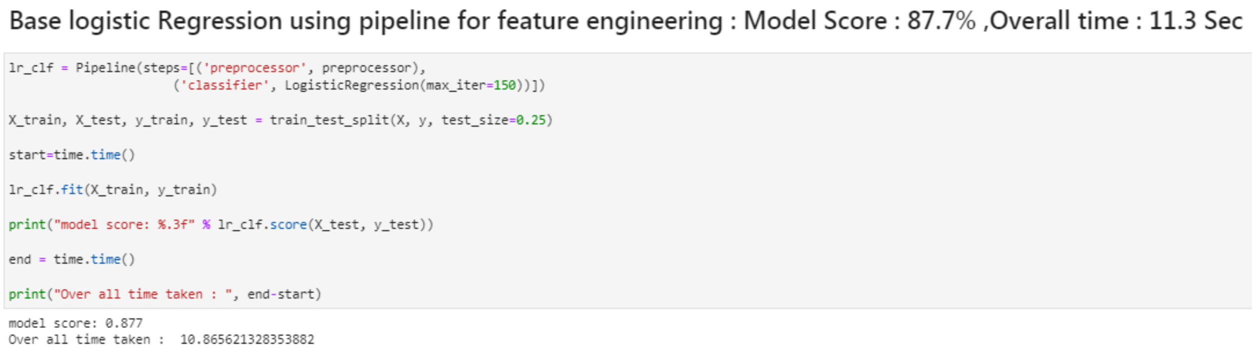
Model Selection and hyper tuning

The last stage of the project is model selection and hyper parameter tuning of the model to get the best results out of the model. In this project we have used Logistic regression (with and without grid search), Random forest (With and without random search) and Support Vector classifier (SVC) (with and without random search) for training and predictions.

1) Logistic regression with default values

We started off or model selection with logistic regression with default value and

It provided the accuracy of 87%. It was quiet effective as out of the box model. In order to explore more we have run a grid search hyper parameter tuning on this model.



Base logistic Regression using pipeline for feature engineering : Model Score : 87.7% ,Overall time : 11.3 Sec

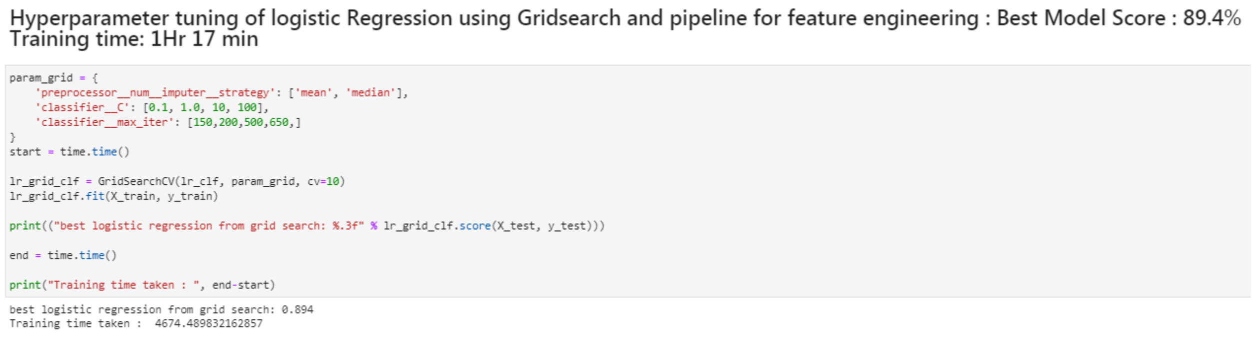
```
lr_clf = Pipeline(steps=[('preprocessor', preprocessor),
                         ('classifier', LogisticRegression(max_iter=150))])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)

start=time.time()

lr_clf.fit(X_train, y_train)

print("model score: %.3f" % lr_clf.score(X_test, y_test))

end = time.time()

print("Over all time taken : ", end-start)
```

```
model score: 0.877
Over all time taken :  10.865621328353882
```

2) Logistic regression with Grid search hyper parameter tuning

In the grid search hyper parameter tuning we have selected a list of parameter as a range and passed it to grid search function to run the model for each combination and provide the model and parameter with the best result to us.

The model ran for 1 hour and 17 min and provided the accuracy of 89.4%. It is better than out of the box model.

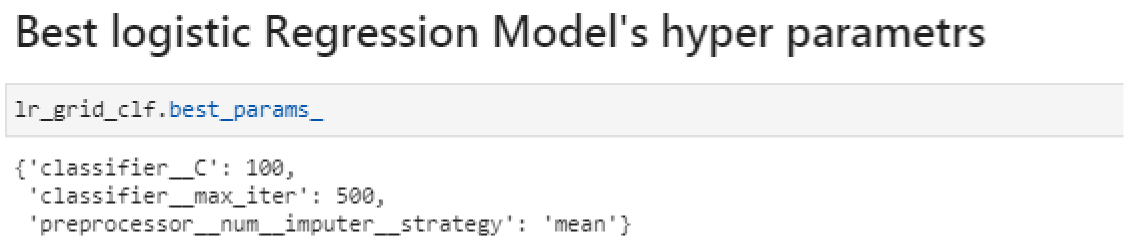


Hyperparameter tuning of logistic Regression using Gridsearch and pipeline for feature engineering : Best Model Score : 89.4%
Training time: 1Hr 17 min

```
param_grid = {
    'preprocessor__num__imputer__strategy': ['mean', 'median'],
    'classifier__C': [0.1, 1.0, 10, 100],
    'classifier__max_iter': [150,200,500,650,]
}
start = time.time()

lr_grid_clf = GridSearchCV(lr_clf, param_grid, cv=10)
lr_grid_clf.fit(X_train, y_train)

print(("best logistic regression from grid search: %.3f" % lr_grid_clf.score(X_test, y_test)))

end = time.time()

print("Training time taken : ", end-start)
```

```
best logistic regression from grid search: 0.894
Training time taken :  4674.489832162857
```

The best parameter of the logistic regression model after the grid search is given below.



Best logistic Regression Model's hyper parametrs

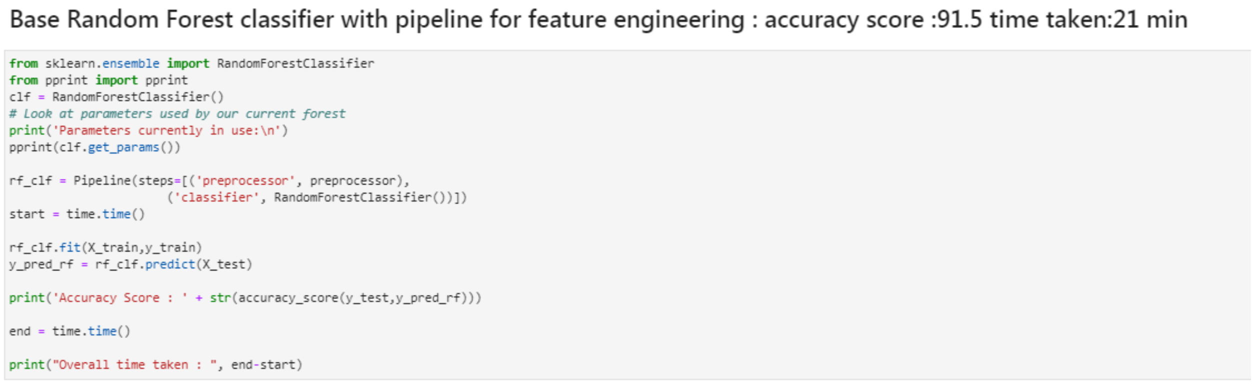
```
lr_grid_clf.best_params_
```

```
{'classifier__C': 100,
 'classifier__max_iter': 500,
 'preprocessor__num__imputer__strategy': 'mean'}
```

3) Random Forest Classifier with default parameter

The next model which we tried was random forest. Random forest is known for the accuracy in the classification problems. So initially we have used random forest with the default parameters. The base random forest classifier ran for 21 min and provided the accuracy of 19.5 which was better than the best model of logistic regression with grid search.

The random forest has truly shown the potential of better classifier in this case. We will develop is further for the better results

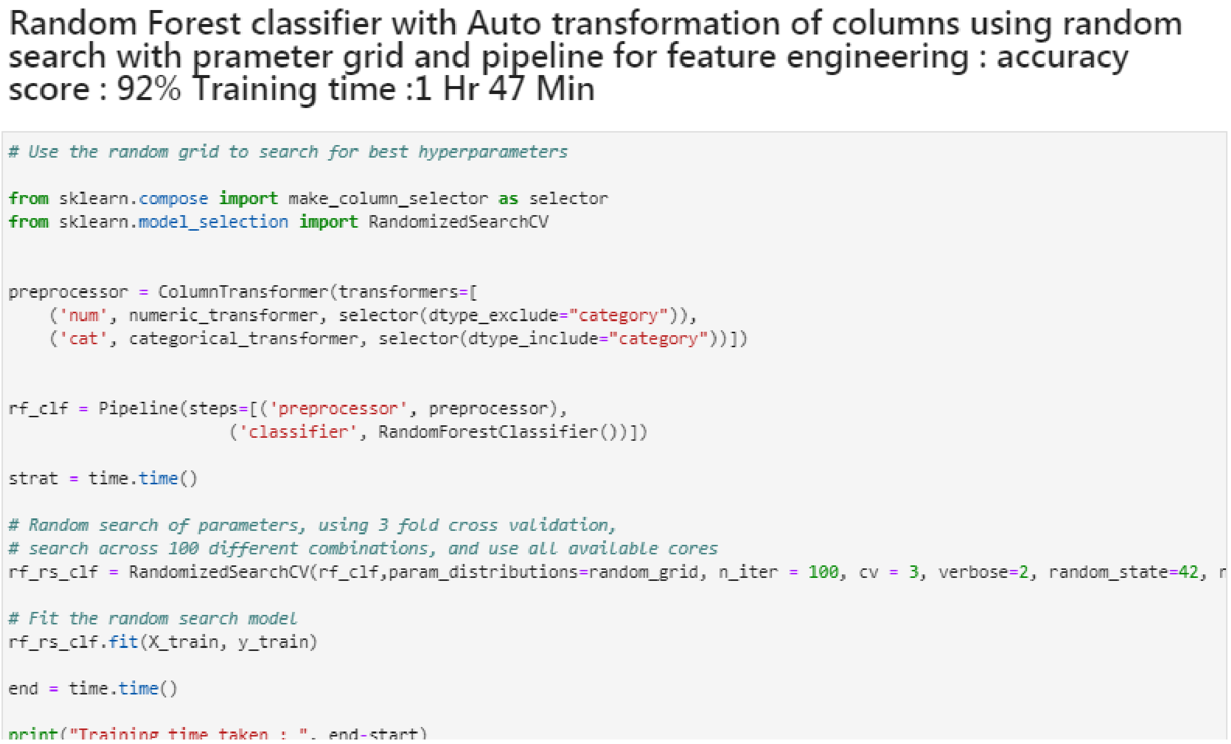


Base Random Forest classifier with pipeline for feature engineering : accuracy score :91.5 time taken:21 min

```
from sklearn.ensemble import RandomForestClassifier
from pprint import pprint
clf = RandomForestClassifier()
# Look at parameters used by our current forest
print('Parameters currently in use:\n')
pprint(clf.get_params())

rf_clf = Pipeline(steps=[('preprocessor', preprocessor),
                        ('classifier', RandomForestClassifier())])
start = time.time()

rf_clf.fit(X_train,y_train)
y_pred_rf = rf_clf.predict(X_test)

print('Accuracy Score : ' + str(accuracy_score(y_test,y_pred_rf)))

end = time.time()

print("Overall time taken : ", end-start)
```

4) Random forest with random search

Next up is the hyper parameter tuning of the random forest classifier with random search technique. The training of the model took 1 hr. and 47 min and gave the accuracy of 92% which is bit better than the out of the box classifier.

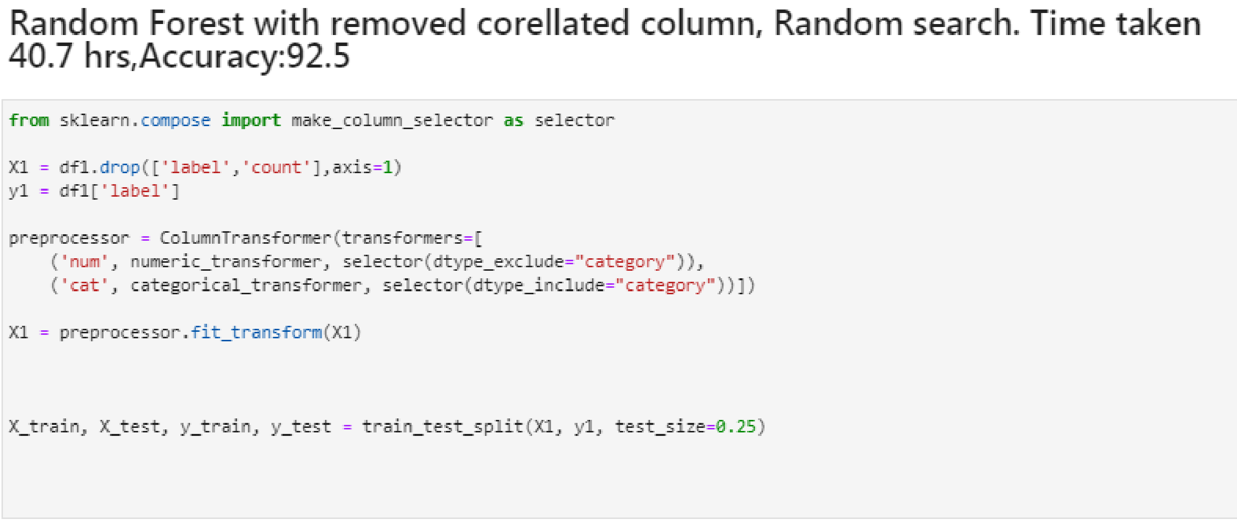


Random Forest classifier with Auto transformation of columns using random search with prameter grid and pipeline for feature engineering : accuracy score : 92% Training time :1 Hr 47 Min

```
# Use the random grid to search for best hyperparameters

from sklearn.compose import make_column_selector as selector
from sklearn.model_selection import RandomizedSearchCV


preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, selector(dtype_exclude="category")),
    ('cat', categorical_transformer, selector(dtype_include="category"))])


rf_clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', RandomForestClassifier())])

strat = time.time()

# Random search of parameters, using 3 fold cross validation,
# search across 100 different combinations, and use all available cores
rf_rs_clf = RandomizedSearchCV(rf_clf,param_distributions=random_grid, n_iter = 100, cv = 3, verbose=2, random_state=42, r

# Fit the random search model
rf_rs_clf.fit(X_train, y_train)

end = time.time()

print("Training time taken : ", end-start)
```

5) Random forest with random search with removed correlated columns

In this model we have removed the count column as it was closely related to length column as we have seen earlier. We have again train the random classifier and it gave the accuracy of 92.5%. Turns out removing the correlated columns do increase the accuracy.



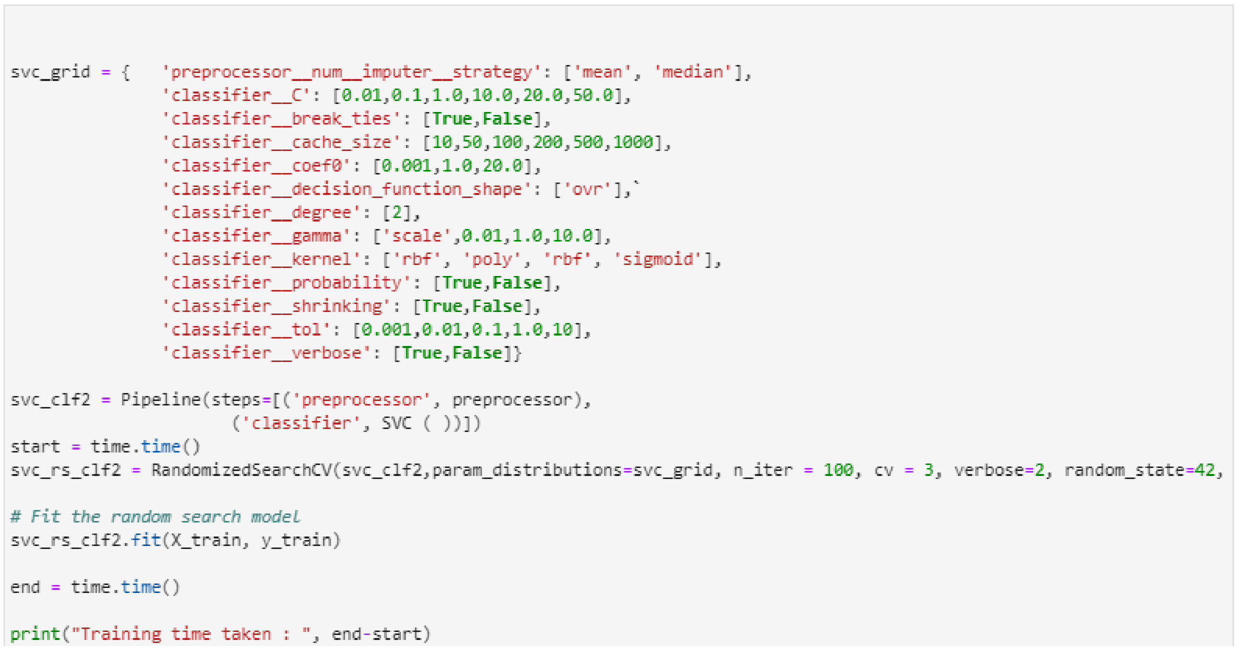
Random Forest with removed corellated column, Random search. Time taken 40.7 hrs,Accuracy:92.5

```
from sklearn.compose import make_column_selector as selector

X1 = df1.drop(['label','count'],axis=1)
y1 = df1['label']

preprocessor = ColumnTransformer(transformers=[
    ('num', numeric_transformer, selector(dtype_exclude="category")),
    ('cat', categorical_transformer, selector(dtype_include="category"))])

X1 = preprocessor.fit_transform(X1)



X_train, X_test, y_train, y_test = train_test_split(X1, y1, test_size=0.25)
```

6) **SVC Support vector classifier with random search**

The last model we trained on this data was SVC model with random search for the hyper parameter tuning. It trained for most time 71.2hrs and gave the accuracy of 90.6% which is even less the out of the box random classifier. Turns out waiting for 70 hours for the result was not worth it after all

SVC model Random search: time take 71.2 hrs, accuracy: 90.6

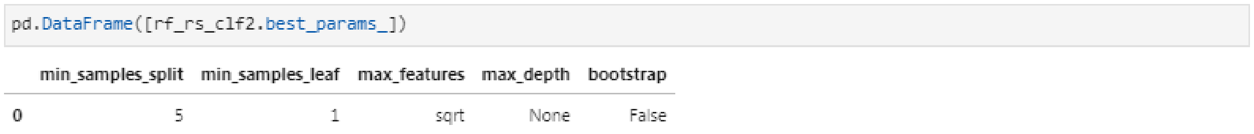


```
svc_grid = {   'preprocessor__num__imputer__strategy': ['mean', 'median'],
               'classifier__C': [0.01,0.1,1.0,10.0,20.0,50.0],
               'classifier__break_ties': [True,False],
               'classifier__cache_size': [10,50,100,200,500,1000],
               'classifier__coef0': [0.001,1.0,20.0],
               'classifier__decision_function_shape': ['ovr'],`
               'classifier__degree': [2],
               'classifier__gamma': ['scale',0.01,1.0,10.0],
               'classifier__kernel': ['rbf', 'poly', 'rbf', 'sigmoid'],
               'classifier__probability': [True,False],
               'classifier__shrinking': [True,False],
               'classifier__tol': [0.001,0.01,0.1,1.0,10],
               'classifier__verbose': [True,False]}

svc_clf2 = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', SVC ( ))])
start = time.time()
svc_rs_clf2 = RandomizedSearchCV(svc_clf2,param_distributions=svc_grid, n_iter = 100, cv = 3, verbose=2, random_state=42,

# Fit the random search model
svc_rs_clf2.fit(X_train, y_train)

end = time.time()

print("Training time taken : ", end-start)
```

Conclusion

SO the final conclusion was the random forest classifier with removed correlated parameters was able to achieve the accuracy of 92.5%. Therefore for this data we would prefer the random forest with below parameters.

Best model has been acquried using the random searh on random forest clasifer with removed corelated columns with 92.5% accuracy with below parameters



```
pd.DataFrame([rf_rs_clf2.best_params_])
```

	min_samples_split	min_samples_leaf	max_features	max_depth	bootstrap
0	5	1	sqrt	None	False

Future Research

Future research on this data could be driving more features based on the current ones and try other machine learning classifier and neural net to classify the bitcoin address. It can also be deployed as a website to be used as bitcoin address verification tool before performing any transaction using bitcoin.