FE 520 Project Report

Team Number 14

Names:

Parth Parab - 10444835

Varsha Raghavendra - 10444838

Predicting credit rating using machine learning (Classification) The topic we chose specifically is to predict whether a credit card will be approved or not for that customer based on the entire credit rating and customer information given to us by an anonymous bank. The data we found was from Kaggle: https://www.kaggle.com/rikdifos/credit-card-approval-prediction

Motivation and Insights:

Analysis of customer history data often gives us a lot of insight into whether approving credit cards would prove to be disastrous to the bank or not. Repayment of credit and loans are how banks function and are necessary to be ensured before giving away credit cards to customers. Other than checking for income or bank balance a lot of other factors play a role. Analyzing the data we have, making them useful and building predictive models will make credit card approvals a more automatic and faster process also ensuring integrity of the customer.

Related work:

Related work would be the various classification algorithms we have worked on before from other courses. Credit Rating might be new but we have done COVID-19 detection and Breast Cancer detection both of which have similar mix of numerical, string, boolean and text data with either binary or multi labels to classify. We have taken a data mining course before that puts us in a good place to navigate through the data we have, visualize, modify, merge and build models.

Importing necessary packages

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from imblearn.over_sampling import SMOTE
import itertools

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

from xgboost import XGBClassifier
```

```
14 from sklearn import svm
15 from sklearn.ensemble import RandomForestClassifier
```

Storing dataset into pandas dataframe downloaded from - https://www.kaggle.com/rikdifos/credit-card-approval-prediction

```
1 df1 = pd.read_csv('application_record.csv')
2 df2 = pd.read_csv('credit_record.csv')
```

We have 2 datasets here, application_record.csv which has features like ID, Gender, car or realty owned, number of children, annual income, income type, education type, family status, housing type, age, days employed, whether they own a mobile, work and a home phone, email and occupation type, number of family members. credit_record.csv has ID, months of balance left and the status of payment

Displaying first 5 rows from 'application_record.csv'

```
1 df1.head()
```

	ID	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_T
0	5008804	M	Y	Y	0	427
1	5008805	M	Y	Y	0	427
2	5008806	M	Y	Y	0	112
3	5008808	F	N	Y	0	270
4	5008809	F	N	Y	0	270

Display first 5 rows from 'credit_record.csv'

```
1 df2.head()
```

	ID	MONTHS_BALANCE	STATUS
0	5001711	0	X

Data Preprocessing

Both datasets have a common column called ID on which they can be merged in order to complete our dataset and define our target variable.

```
#Let's join both datasets on ID

df = df1.merge(df2, on='ID', how='left')
df.head()
```

	ID	CODE_GENDER	FLAG_OWN_CAR	FLAG_OWN_REALTY	CNT_CHILDREN	AMT_INCOME_T
0	5008804	M	Y	Y	0	427
1	5008804	M	Y	Y	0	427
2	5008804	M	Y	Y	0	427
3	5008804	M	Y	Y	0	427
4	5008804	M	Y	Y	0	427

Next we start with data pre-processing. Given the nature of the data we thought it is best to drop the Null or NaN rows to maintain a consistent dataset without affecting the prediction output.

```
print('Total number of Rows before dropping Null/NaN values: ', (len(df)))
df.dropna()
df = df.mask(df == 'NULL').dropna()
print('Total number of Rows after dropping Null/NaN values: ', (len(df)))
```

```
Total number of Rows before dropping Null/NaN values:  1179815
Total number of Rows after dropping Null/NaN values:  537667
```

We used column ‘MONTHS_BALANCE’ and binned values into a new column ‘Risk’ more than -3 to be “no” risk and values less than -3 to be “yes” risk. If it isn't either, then risk doesn't exist and will be 'null'

```
def risk(x):
    if x >= -3:
        return 'no'
    elif x < -3:
```

```
        return 'yes'
    else:
        return 'null'

df['RISK'] = df['MONTHS_BALANCE'].apply(lambda x: risk(x))
```

```
df['RISK'].head()
```

```
31     no
32     no
33     no
34     no
35    yes
Name: RISK, dtype: object
```

For our target variable we created a column ‘Label’ where if the column STATUS has a 0, 1, 2, 3, 4, 5, it means that they are that many months late in paying their credit debt. Hence this equates to 1 (don’t approve). 'X' and 'C' are non-defaulter codes where the customer has cleared all dues or has no dues. Hence if the column STATUS has any of these two values, column 'Label' is equated to 0.

These definitions are based on the dataset descriptions given on Kaggle

```
defaulter_codes= ['0','1','2','3','4','5']
# data labelling. 1 is a defaulter 0 is not
df['Label'] = np.where(df.STATUS.isin(defaulter_codes), 1, 0)
df['Label'].head(10)
```

```
31    0
32    0
33    0
34    0
35    0
36    0
37    0
38    0
39    1
40    1
Name: Label, dtype: int64
```

We now drop the 'STATUS' column as we already derived our target variable from it

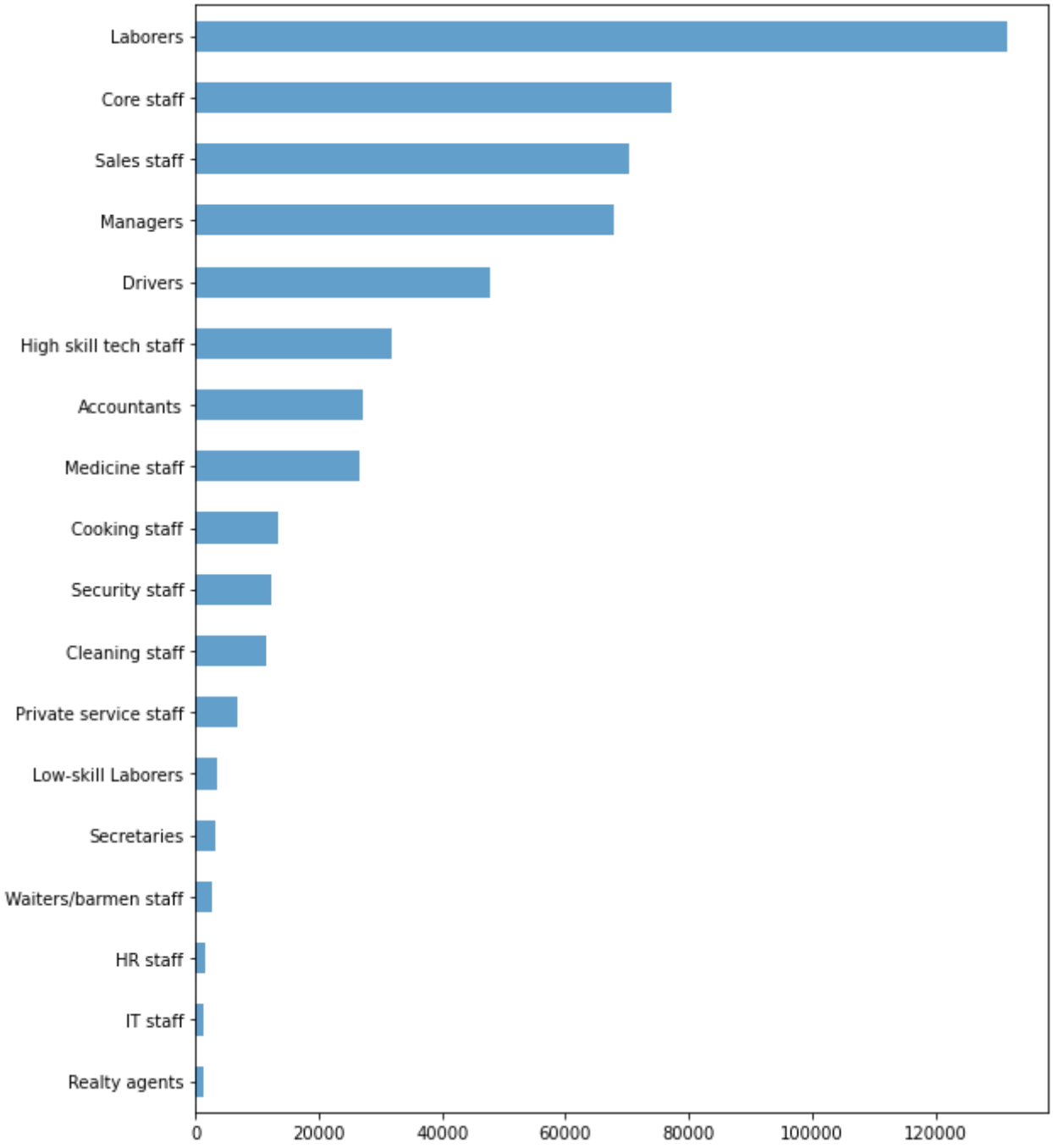
```
df.drop('STATUS', axis=1, inplace=True)
```

Data Visualization

Let’s see the various occupation types and the count of customers in each type. We see that laborers and core staff are the highest number of applicants for credit cards

```
1 df['OCCUPATION_TYPE'].value_counts().sort_values().plot(kind='barh', figsize=(9,12)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7403712518>
```



We now visualize our target variable 'Label'. Looking at the counts, we can see that there are more 0s than 1s as would be the case in real life credit card companies. There would be more people who get their credit cards approved as opposed to people who don't.

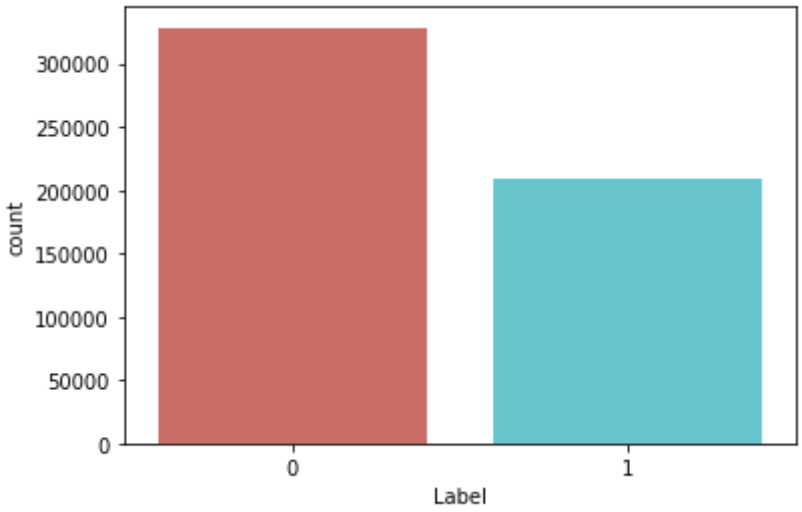
We have 328352 rows with target variable 0 and 209315 rows with target variable 1. The dataset is not extremely unbalanced and hence does not require data manipulation to fix it.

```
1 print('Total value count of target variable:\n',df['Label'].value_counts(),'\n\nEDA
```

```
2 chart=sns.countplot(x='Label', data = df, palette = 'hls')
```

```
Total value count of target variable:
 0    328352
1    209315
Name: Label, dtype: int64
```

EDA for the above:



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to Gender. We can see that there were more female applicants and in general more customers in both male and female were approved credit cards as opposed to those who weren't approved credit cards

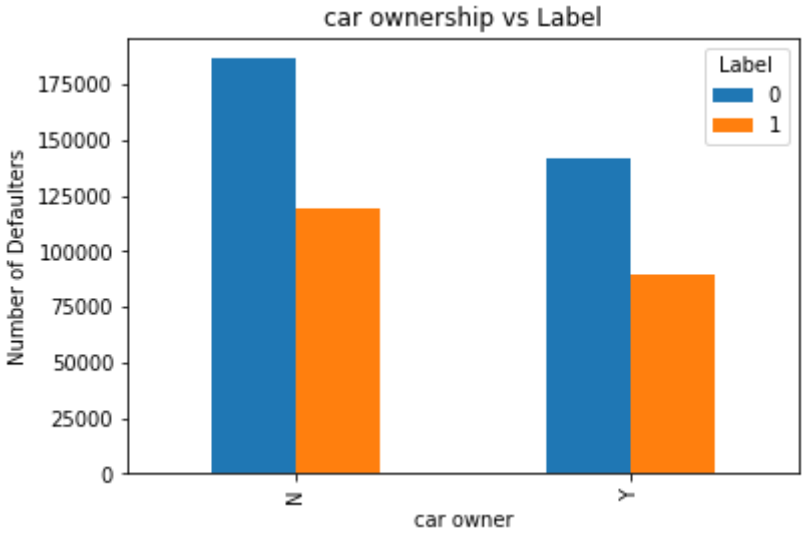
```
1 pd.crosstab(df.CODE_GENDER,df.Label).plot(kind='bar')
2 plt.title('gender vs Label')
3 plt.xlabel('gender')
4 plt.ylabel('Number of Defaulters')
5
```

```
Text(0, 0.5, 'Number of Defaulters')
```

Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to Car ownerships. We can see that there were more applicants who didn't own a car as opposed to those who did own a car.

```
1 pd.crosstab(df.FLAG_OWN_CAR,df.Label).plot(kind='bar')
2 plt.title('car ownership vs Label')
3 plt.xlabel('car owner')
4 plt.ylabel('Number of Defaulters')
```

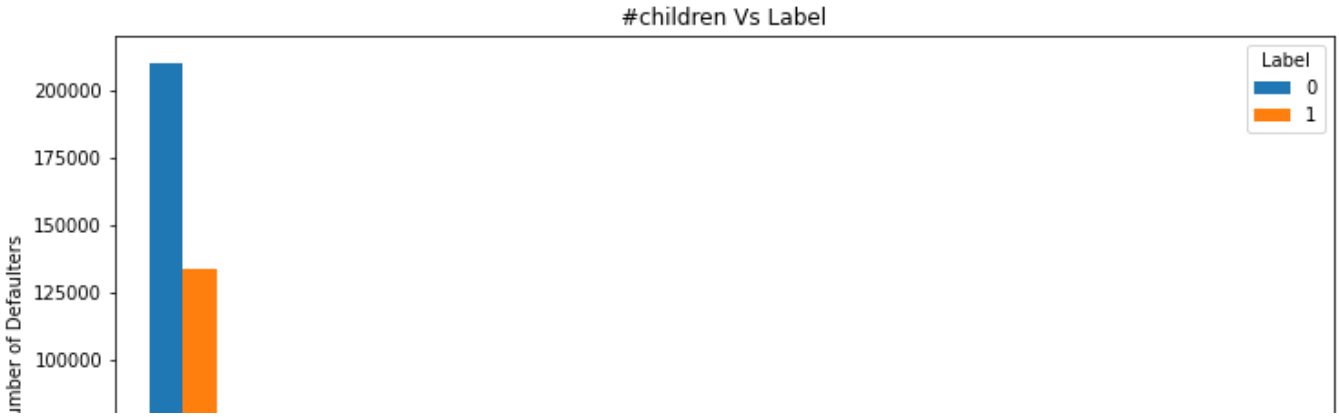
```
Text(0, 0.5, 'Number of Defaulters')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to the number of children. We can see that there were more applicants who had no children as opposed to more than 1. This might mean more people apply to credit cards when they have less liability as also observed in the graph of car ownership.

```
1 pd.crosstab(df.CNT_CHILDREN, df.Label).plot(kind='bar', figsize=(12,6))
2 plt.title('#children Vs Label')
3 plt.xlabel('number of children')
4 plt.ylabel('Number of Defaulters')
```

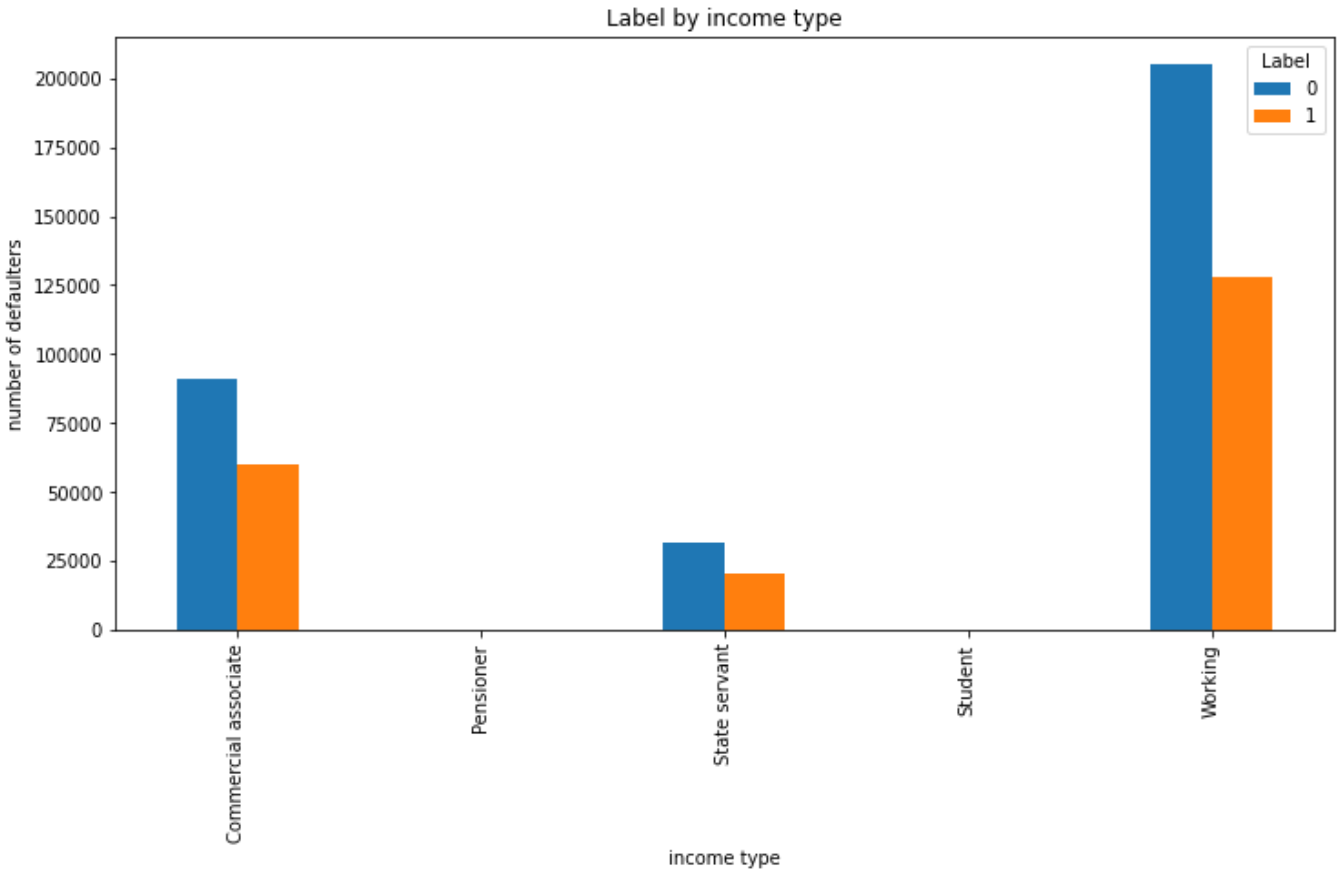
```
Text(0, 0.5, 'Number of Defaulters')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to Income type. We can see that there were more working professionals applied for a credit card because they have the capital required to use one.

```
1 pd.crosstab(df.NAME_INCOME_TYPE, df.Label).plot(kind='bar', figsize=(12,6))
2 plt.title('Label by income type')
3 plt.xlabel('income type')
4 plt.ylabel('number of defaulters')
```

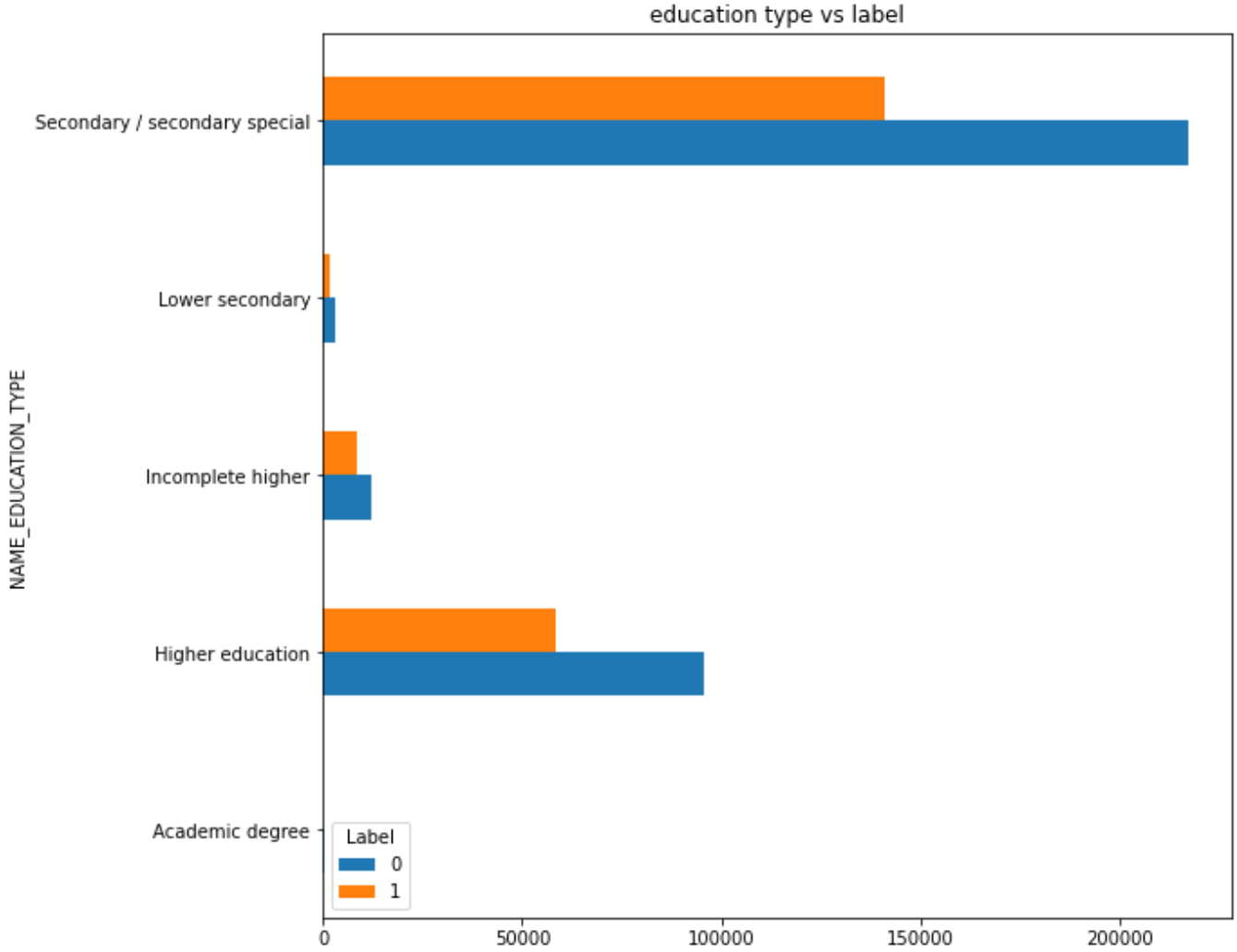
```
Text(0, 0.5, 'number of defaulters')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to the education type. We can see that there were more customers with secondary and higher education that applied for credit cards as opposed to just those with lower secondary or incomplete education because with higher levels of education comes a better understanding of credit and the ability to take responsibility of finances

```
1 pd.crosstab(df.NAME_EDUCATION_TYPE, df.Label).plot(kind='barh', figsize=(9,9))
2 plt.title('education type vs label')
```

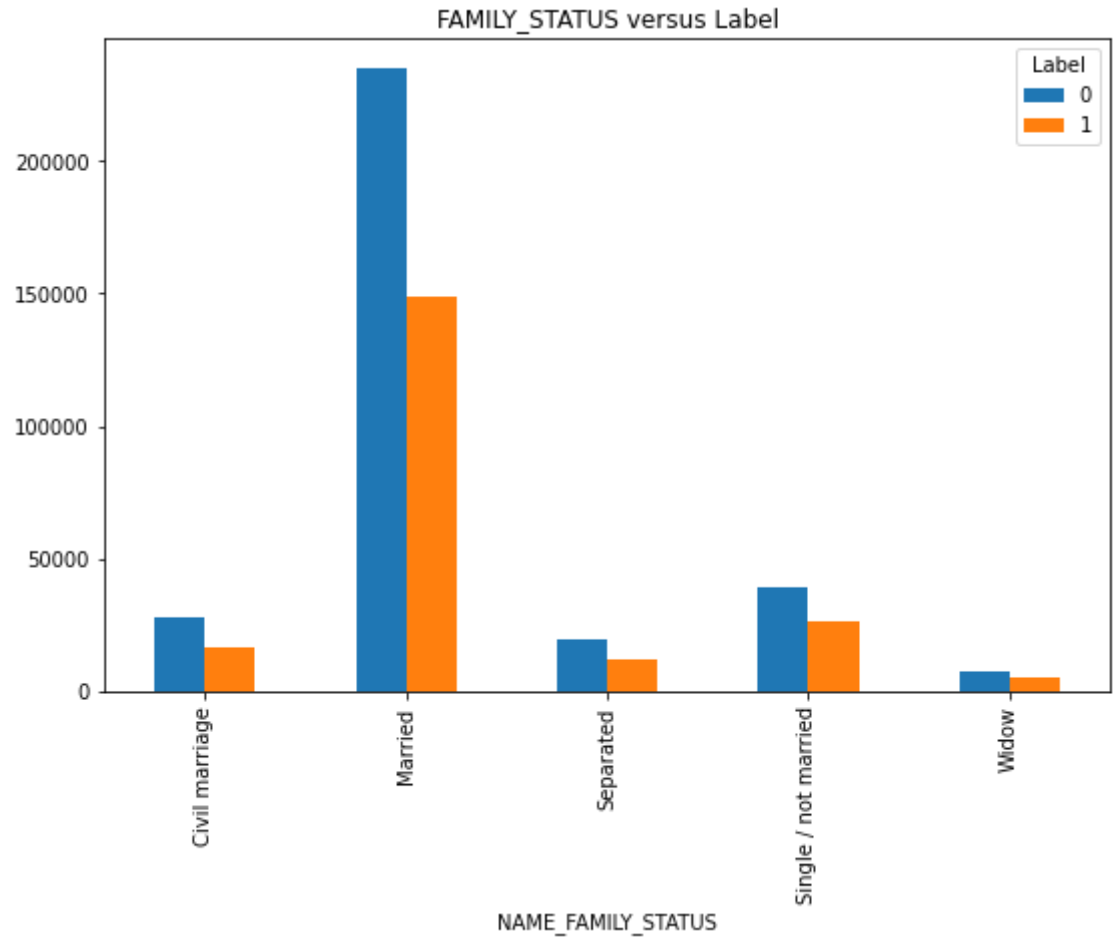
```
Text(0.5, 1.0, 'education type vs label')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to the family status. We can see that there were majority of customers that applied for credit cards were married as more income in the household gives a better security to be able to repay debts.

```
1 pd.crosstab(df.NAME_FAMILY_STATUS, df.Label).plot(kind='bar', figsize=(9,6))
2 plt.title('FAMILY_STATUS versus Label')
```

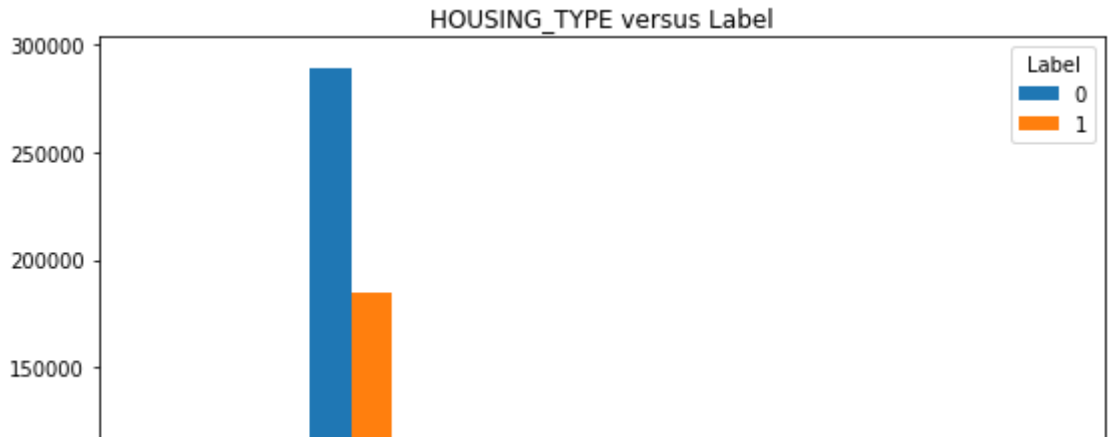
```
Text(0.5, 1.0, 'FAMILY_STATUS versus Label')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to the housing type. We can see that there were majority of customers that lived in a single house or apartment that applied for credit cards because owning a house or apartment suggests a better ability to handle finances

```
1 pd.crosstab(df.NAME_HOUSING_TYPE, df.Label).plot(kind='bar', figsize=(9,6))
2 plt.title('HOUSING_TYPE versus Label')
```

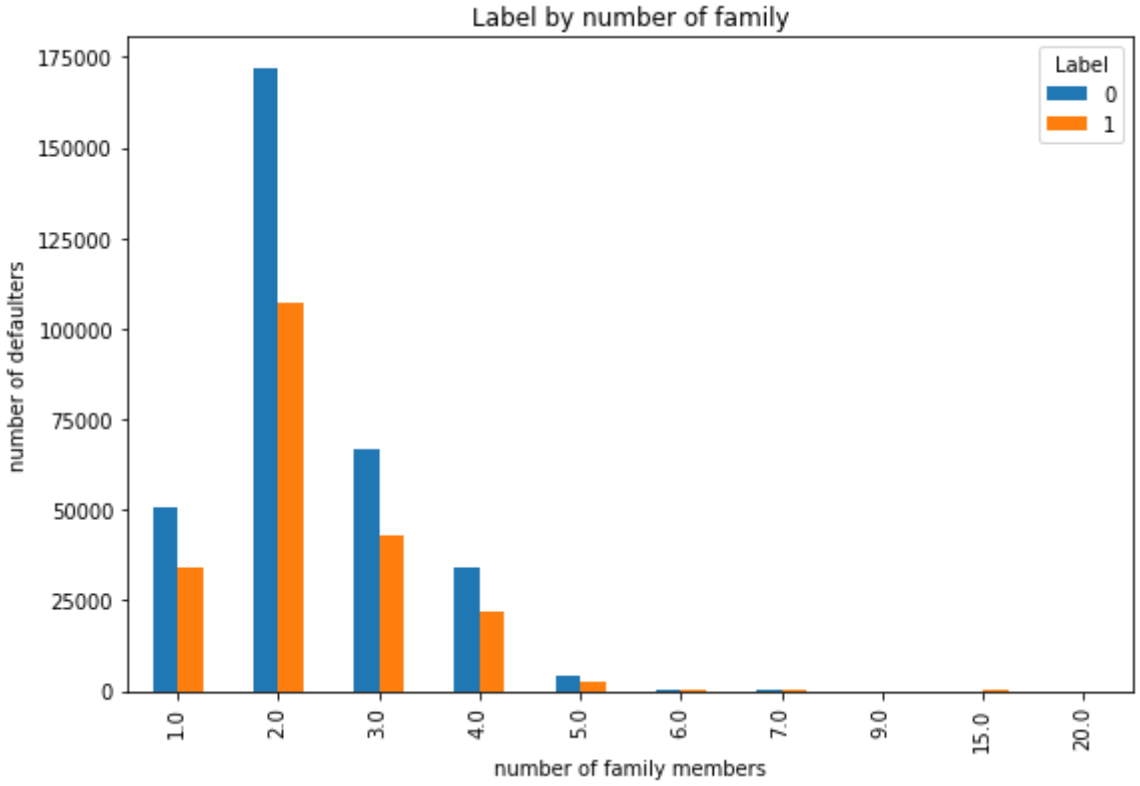
```
Text(0.5, 1.0, 'HOUSING_TYPE versus Label')
```



Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to the number of family members in the household. We can see that there were majority of customers that lived with 2 people that applied for credit cards. This complements the graph where we see mostly married customers with no children applying for credit cards in majority.


```
1 pd.crosstab(df.CNT_FAM_MEMBERS,df.Label).plot(kind='bar', figsize=(9,6))
2
3 plt.title('Label by number of family')
4 plt.xlabel('number of family members')
5 plt.ylabel('number of defaulters')
```

```
Text(0, 0.5, 'number of defaulters')
```

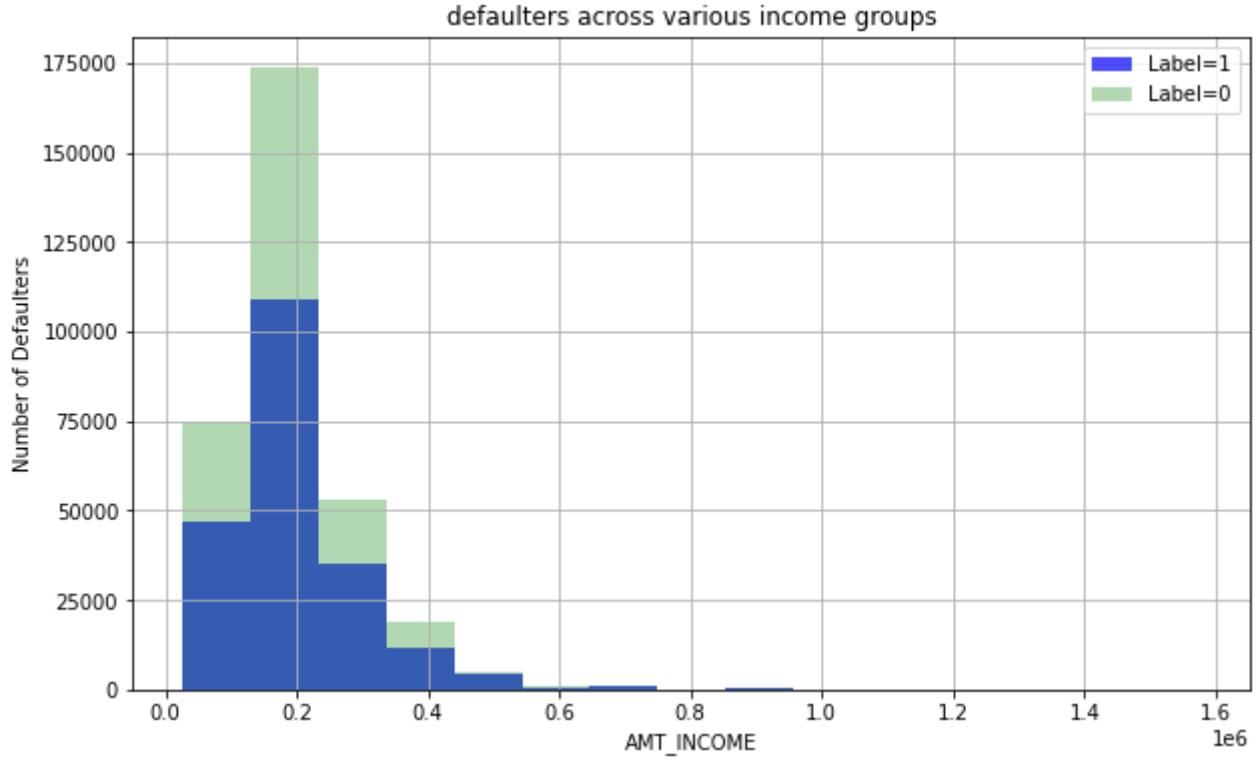


Let's see the count of the number of defaulters as opposed to the ones that didn't with respect to annual income. We can see that there were majority of customers that applied for credit cards had approximate annual incomes of $200000 as this is a good figure to be able to afford to repay credit debt

```
plt.figure(figsize=(10,6))
df[df['Label']==1]['AMT_INCOME_TOTAL'].hist(alpha=0.7,color='blue',
                                            bins=15,label='Label=1')
df[df['Label']==0]['AMT_INCOME_TOTAL'].hist(alpha=0.3,color='green',
                                              bins=15,label='Label=0')

plt.title('defaulters across various income groups')
plt.legend()
plt.xlabel('AMT_INCOME')
plt.ylabel('Number of Defaulters')
```

```
Text(0, 0.5, 'Number of Defaulters')
```



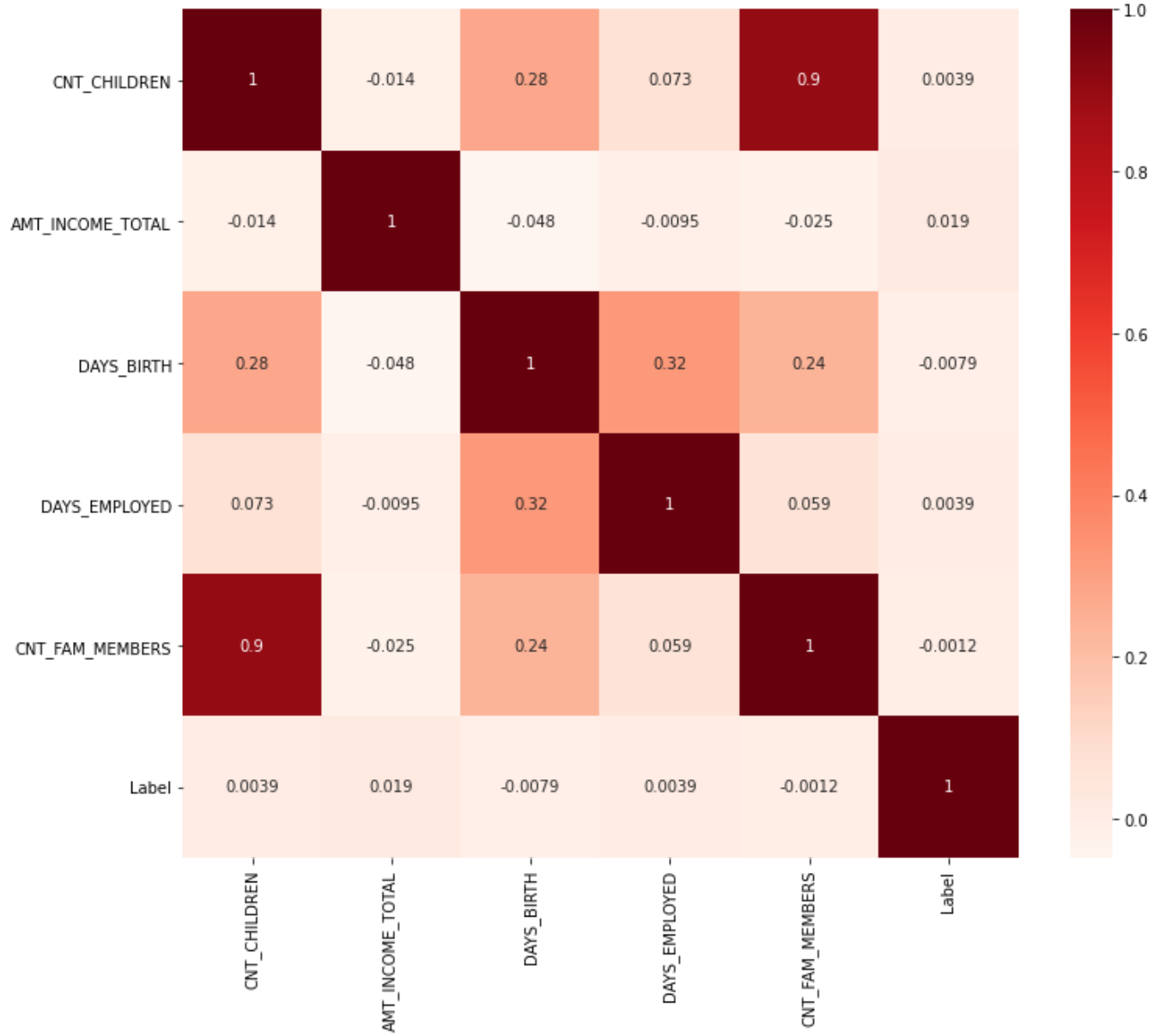
Let's tabulate a correlation matrix to understand our features and which ones are dependent on each other. The highest correlation was seen between number of children and number of family members. Furthermore, we see that days since birth and days employed as well as days since birth and number of children are also slightly correlated.

```
#Using Pearson Correlation
plt.figure(figsize=(12,10))

corr = df[['CNT_CHILDREN', 'AMT_INCOME_TOTAL', 'DAYS_BIRTH', 'DAYS_EMPLOYED', 'CNT_
sns.heatmap(corr,
            xticklabels=corr.columns,
```

```
7         yticklabels=corr.columns,
8         annot=True, cmap=plt.cm.Reds)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f74028fd9e8>
```



Classification Algorithms

Before we build our classification algorithms to predict credit card approval based on our features, we need to convert string variables to labels. For example, CODE_GENDER has two types of variables, F and M. Our model does not understand string inputs like these. The Label Encoder function converts these input variables to labels or classes. F and M hence become 0 and 1. We run this function and convert 9 such variables to labels.

```
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
df['CODE_GENDER']= label_encoder.fit_transform(df['CODE_GENDER'])
df['FLAG_OWN_CAR']= label_encoder.fit_transform(df['FLAG_OWN_CAR'])
df['FLAG_OWN_REALTY']= label_encoder.fit_transform(df['FLAG_OWN_REALTY'])
df['NAME_INCOME_TYPE']= label_encoder.fit_transform(df['NAME_INCOME_TYPE'])
df['NAME_EDUCATION_TYPE']= label_encoder.fit_transform(df['NAME_EDUCATION_TYPE'])
df['NAME_FAMILY_STATUS']= label_encoder.fit_transform(df['NAME_FAMILY_STATUS'])
df['NAME_HOUSING_TYPE']= label_encoder.fit_transform(df['NAME_HOUSING_TYPE'])
df['OCCUPATION_TYPE']= label_encoder.fit_transform(df['OCCUPATION_TYPE'])
df['RISK']= label_encoder.fit_transform(df['RISK'])
```

We now use train_test_split from sklearn to split our data into training set on which our models learn and train from and a testing set that will be used to evaluate our model built. We define our features which are all variables other than the 'ID' which is just an identifier and 'Label' because that is our target variable. We randomly split our dataset in the ratio 80:20.

```
from sklearn.model_selection import train_test_split

# Features - exclude ID and Label columns
X = df.drop(columns=['ID', 'Label'])
# Label - select only label column
y = df['Label']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Gaussian Naive Bayes Classifier

Gaussian Naive Bayes is an algorithm having a Probabilistic Approach. It involves prior and posterior probability calculation of the classes in the dataset and the test data given a class respectively. Prior probabilities of all the classes are calculated using the same formula

We fit our X_train and y_train on this model

```
from sklearn.naive_bayes import GaussianNB
nb = GaussianNB()
nb.fit(X_train,y_train)
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

We now use the built model to predict the target variables for X_test and print an accuracy score after comparing predcited values to actual y_test values

```
y_pred = nb.predict(X_test)

```

```
3 accuracy_score(y_pred, y_test)
```

```
0.6229843584354716
```

We also display the classification report that has the f1 score, precision and recall values with the accuracy score. We also have printed the confusion matrix.

```
1 from sklearn.metrics import classification_report
2 from sklearn.metrics import confusion_matrix
3 print(classification_report(y_test, y_pred))
4 print(confusion_matrix(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.63      0.90      0.74     65656
           1       0.55      0.19      0.28     41878

    accuracy                           0.62    107534
   macro avg       0.59      0.54      0.51    107534
weighted avg       0.60      0.62      0.56    107534

[[59155  6501]
 [34041  7837]]
```

We saw an accuracy of 62% and hence decided to implement 2 more popular algorithms to improve our accuracy. We also similarly computed classification reports and confusion matrix for the next two algorithms.

Random Forest Classifier

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes or mean/average prediction of the individual trees.

```
1 rf = RandomForestClassifier(n_estimators=100, max_depth=10, random_state=42)
2 rf.fit(X_train,y_train)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=10, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

```
1 y_pred = rf.predict(X_test)
```

```
y_pred = rf.predict(X_test)

accuracy_score(y_pred, y_test)
```

```
0.6608020005358578
```

```
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

```
                precision    recall  f1-score   support

             0       0.65      0.94      0.77     20447
             1       0.71      0.22      0.34     13144

      accuracy                           0.66     33591
     macro avg       0.68      0.58      0.56     33591
  weighted avg       0.68      0.66      0.60     33591

[[19252  1195]
 [10199  2945]]
```

We see an accuracy of 66% which is much more improved as compared to Naive Bayes

XGBoost Classifier

XGBoost is termed as Extreme Gradient Boosting Algorithm which is again an ensemble method that works by boosting trees. XGboost makes use of a gradient descent algorithm which is the reason that it is called Gradient Boosting.

```
creditxgb = XGBClassifier(
learning_rate =0.1,
n_estimators=100,
max_depth=5,
min_child_weight=50,
gamma=0,
subsample=0.8,
colsample_bytree=0.8,
objective= 'binary:logistic'
)
```

```
creditxgb.fit(X_train,y_train)
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.8, gamma=0,
              learning_rate=0.1, max_delta_step=0, max_depth=5,
              min_child_weight=50, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=0.8, verbosity=1)
```

```
y_pred = creditxgb.predict(X_test)

accuracy_score(y_pred, y_test)
```

```
0.6754785508022982
```

```
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

```
                precision    recall  f1-score   support

             0       0.67      0.92      0.77     20447
             1       0.70      0.30      0.42     13144

      accuracy                           0.68     33591
     macro avg       0.68      0.61      0.60     33591
  weighted avg       0.68      0.68      0.64     33591

[[18742  1705]
 [ 9196  3948]]
```

XGBoost gives us the highest accuracy of 68%

A few reasons why XGBoost is a powerful algorithm:

1. Regularization: XGBoost has in-built L1 (Lasso Regression) and L2 (Ridge Regression) regularization which prevents the model from overfitting. That is why, XGBoost is also called regularized form of GBM (Gradient Boosting Machine). While using Scikit Learn libarary, we pass two hyper-parameters (alpha and lambda) to XGBoost related to regularization. alpha is used for L1 regularization and lambda is used for L2 regularization.
2. Cross Validation: XGBoost allows user to run a cross-validation at each iteration of the boosting process and thus it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike GBM where we have to run a grid-search and only a limited values can be tested.
3. Effective Tree Pruning: A GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a greedy algorithm. XGBoost on the other hand make splits upto the max_depth specified and then start pruning the tree backwards and remove splits beyond which there is no positive gain.

We now perform cross validation on all three algorithms in consideration so we can compare their performances.

```
from sklearn.model_selection import cross_val_score, KFold
```

```
models = []
models.append(('NB', GaussianNB()))
models.append(('RF', RandomForestClassifier(n_estimators=100, max_depth=10, random_
models.append(('XGB', XGBClassifier(
learning_rate =0.1,
 n_estimators=100,
 max_depth=5,
 min_child_weight=50,
 gamma=0,
 subsample=0.8,
 colsample_bytree=0.8,
 objective= 'binary:logistic'
)))

results = []
names = []
scoring = 'accuracy'
for name, model in models:
  cv_results = cross_val_score(model, X_train, y_train, cv=2, scoring=scoring)
  results.append(cv_results)
  names.append(name)
  msg = "%s: %.2f (%f)" % (name, cv_results.mean(), cv_results.std())
  print(msg)
```

```
NB: 0.62 (0.000405)
RF: 0.65 (0.001779)
XGB: 0.65 (0.000438)
```

We see that XGBoost and Random Forest both have equal accuracy scores but XGBoost has a lower standard devaition thus making it a better choice to tune and use for credit card approval.

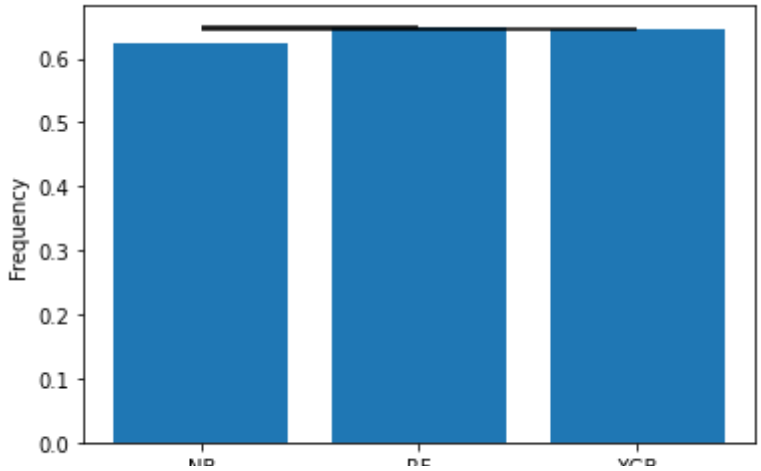
From the graph below, we can clearly see how both these algorithms perform and make the final decision.

```
import matplotlib.pyplot as plt

x = names
y = [results[i].mean() for i in range(len(results))]

plt.bar(x,y,align='center') # A bar chart
plt.xlabel('Bins')
plt.ylabel('Frequency')
for i in range(len(y)):
    plt.hlines(y[i],0,x[i]) # Here you are drawing the horizontal lines
plt.show()

```



We choose XGBoost to hyperparameter tune our model to improve its performance. We use a parameter grid as specified below and run a RandomizedSearchCV algorithm to find out the best parameters that give the best accuracy.

RandomizedSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings. In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

```
#Hyperparameter tuning for XGBoost
import time
param_grid = {
        'silent': [False],
        'max_depth': [6, 10, 15, 20],
        'learning_rate': [0.001, 0.01, 0.1, 0.2, 0,3],
        'subsample': [0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'colsample_bytree': [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'colsample_bylevel': [0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        'min_child_weight': [0.5, 1.0, 3.0, 5.0, 7.0, 10.0],
        'gamma': [0, 0.25, 0.5, 1.0],
        'reg_lambda': [0.1, 1.0, 5.0, 10.0, 50.0, 100.0],
        'n_estimators': [100]}

fit_params = {'eval_metric': 'mlogloss',
              'early_stopping_rounds': 10,
              'eval_set': [(X_test, y_test)]}
xgb=XGBClassifier(random_state=42, subsample = 1.0, silent = False, reg_lambda = 1(

rs_clf = RandomizedSearchCV(xgb, param_grid, n_iter=20,
                            n_jobs=1, verbose=2, cv=2,
                            scoring='neg_log_loss', refit=False, random_state=42)
print("Randomized search..")
search_time_start = time.time()
rs_clf.fit(X_train, y_train)
```

```
rs_clf.fit(X_train, y_train)
print("Randomized search time:", time.time() - search_time_start)

best_score = rs_clf.best_score_
best_params = rs_clf.best_params_
```

```
Randomized search..
Fitting 2 folds for each of 20 candidates, totalling 40 fits
[CV] subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child_
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers
[CV]  subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child
[CV] subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child_
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    6.2s remaining:    0.0s
[CV]  subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child
[CV] subsample=0.6, silent=False, reg_lambda=1.0, n_estimators=100, min_child_w
[CV]  subsample=0.6, silent=False, reg_lambda=1.0, n_estimators=100, min_child_
[CV] subsample=0.6, silent=False, reg_lambda=1.0, n_estimators=100, min_child_w
[CV]  subsample=0.6, silent=False, reg_lambda=1.0, n_estimators=100, min_child_
[CV] subsample=0.9, silent=False, reg_lambda=50.0, n_estimators=100, min_child_
[CV]  subsample=0.9, silent=False, reg_lambda=50.0, n_estimators=100, min_child
[CV] subsample=0.9, silent=False, reg_lambda=50.0, n_estimators=100, min_child_
[CV]  subsample=0.9, silent=False, reg_lambda=50.0, n_estimators=100, min_child
[CV] subsample=0.5, silent=False, reg_lambda=1.0, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=1.0, n_estimators=100, min_child_
[CV] subsample=0.5, silent=False, reg_lambda=1.0, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=1.0, n_estimators=100, min_child_
[CV] subsample=0.7, silent=False, reg_lambda=5.0, n_estimators=100, min_child_w
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
[CV]  subsample=0.7, silent=False, reg_lambda=5.0, n_estimators=100, min_child_
[CV] subsample=0.7, silent=False, reg_lambda=5.0, n_estimators=100, min_child_w
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
[CV]  subsample=0.7, silent=False, reg_lambda=5.0, n_estimators=100, min_child_
[CV] subsample=0.6, silent=False, reg_lambda=10.0, n_estimators=100, min_child_
[CV]  subsample=0.6, silent=False, reg_lambda=10.0, n_estimators=100, min_child
[CV] subsample=0.6, silent=False, reg_lambda=10.0, n_estimators=100, min_child_
[CV]  subsample=0.6, silent=False, reg_lambda=10.0, n_estimators=100, min_child
[CV] subsample=0.5, silent=False, reg_lambda=0.1, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=0.1, n_estimators=100, min_child_
[CV] subsample=0.5, silent=False, reg_lambda=0.1, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=0.1, n_estimators=100, min_child_
[CV] subsample=0.5, silent=False, reg_lambda=5.0, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=5.0, n_estimators=100, min_child_
[CV] subsample=0.5, silent=False, reg_lambda=5.0, n_estimators=100, min_child_w
[CV]  subsample=0.5, silent=False, reg_lambda=5.0, n_estimators=100, min_child_
[CV] subsample=0.9, silent=False, reg_lambda=10.0, n_estimators=100, min_child_
[CV]  subsample=0.9, silent=False, reg_lambda=10.0, n_estimators=100, min_child
[CV] subsample=0.9, silent=False, reg_lambda=10.0, n_estimators=100, min_child_
[CV]  subsample=0.9, silent=False, reg_lambda=10.0, n_estimators=100, min_child
[CV] subsample=0.8, silent=False, reg_lambda=100.0, n_estimators=100, min_child
[CV]  subsample=0.8, silent=False, reg_lambda=100.0, n_estimators=100, min_chil
[CV] subsample=0.8, silent=False, reg_lambda=100.0, n_estimators=100, min_child
```

```
[CV]  subsample=0.8, silent=False, reg_lambda=100.0, n_estimators=100, min_chil
[CV] subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child_
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:2295:
  loss = -(transformed_labels * np.log(y_pred)).sum(axis=1)
[CV]  subsample=0.7, silent=False, reg_lambda=50.0, n_estimators=100, min_child
```

Here are the best parameters as computed by our RandomizedSearchCV algorithm

```
1 print(best_params)
```

```
{'subsample': 0.9, 'silent': False, 'reg_lambda': 10.0, 'n_estimators': 100, 'mi
```

We now fit these parameters into a new XGBoost model and train on our training dataset again.

```
1 xgb = XGBClassifier(**rs_clf.best_params_)
2 xgb.fit(X_train, y_train)
```

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=0.4,
              colsample_bynode=1, colsample_bytree=0.6, gamma=0.5,
              learning_rate=0.2, max_delta_step=0, max_depth=15,
              min_child_weight=0.5, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=0,
              reg_alpha=0, reg_lambda=10.0, scale_pos_weight=1, seed=None,
              silent=False, subsample=0.9, verbosity=1)
```

We compute the predicted variables on our testing set and compare the values by computing an accuracy score.

```
1 y_pred = xgb.predict(X_test)
2
3 accuracy_score(y_pred, y_test)
```

```
0.7705635438063767
```

We now see a highly improved accuracy score of 77% which is a 13% performance increase compared to our previous untuned XGBoost Classifier. We can now calculate a classification report and confusion matrix to verify this upgrade

```
1 print(classification_report(y_test, y_pred))
2 print(confusion_matrix(y_test, y_pred))
```

```
              precision    recall  f1-score   support
```

```
              0       0.79      0.86      0.82     20447
              1       0.74      0.64      0.69     13144

       accuracy                           0.77     33591
      macro avg       0.76      0.75      0.75     33591
   weighted avg       0.77      0.77      0.77     33591

   [[17493  2954]
    [ 4753  8391]]
```

Hence we can now use XGBoost with its tuned parameter to assess applicants and decide if credit cards should be approved or not

We also plot the feature importances using XGBoost to look at the variables that have the maximum weightage in determining our result. The age, number of days employed and the annual income are crucial features as seen from the graph

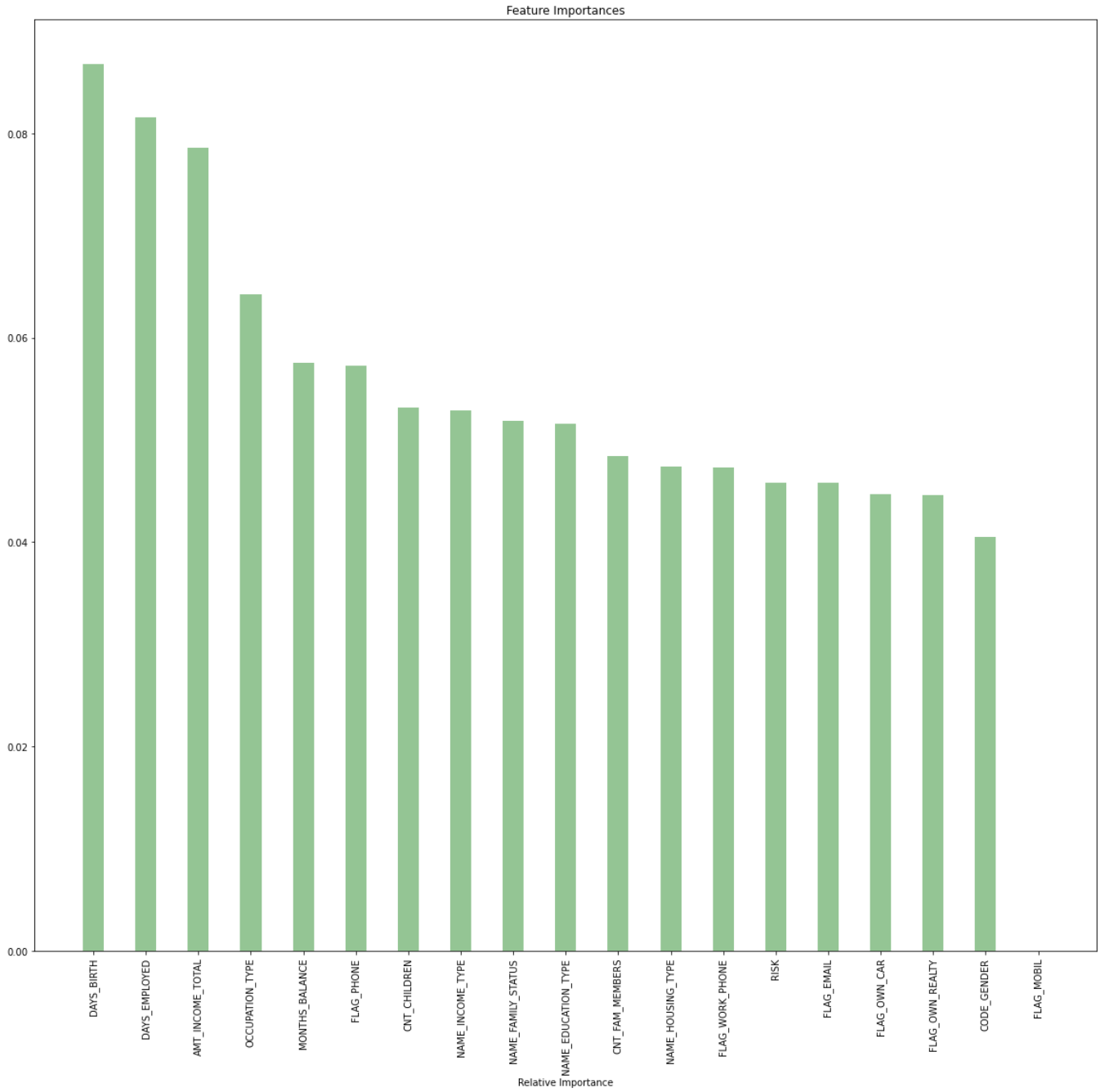
```
#Plot the importances

important_features_dict = {}
for x,i in enumerate(xgb.feature_importances_):
    important_features_dict[x]=i


important_features_list = sorted(important_features_dict,
                                 key=important_features_dict.get,
                                 reverse=True)

#Features from Rank 1 to end
cols = list(X_test.columns)
final_features = []
for i in important_features_list:
  final_features.append(cols[i])

importances = xgb.feature_importances_
indices = np.argsort(importances)
indices = indices[::-1]
cols = list(X.columns)
plt.figure(figsize=(20,18))
plt.title('Feature Importances')
plt.bar(range(len(indices)), importances[indices], color='#94c594', align='center',
plt.xticks(range(len(indices)), final_features, rotation ='vertical')
plt.xlabel('Relative Importance')
plt.show()
```


Feature Importances
0.08
0.06
0.04
0.02
0.00
DAYS_BIRTH
DAYS_EMPLOYED
AMT_INCOME_TOTAL
OCCUPATION_TYPE
MONTHS_BALANCE
FLAG_PHONE
CNT_CHILDREN
NAME_INCOME_TYPE
NAME_FAMILY_STATUS
NAME_EDUCATION_TYPE
CNT_FAM_MEMBERS
NAME_HOUSING_TYPE
FLAG_WORK_PHONE
RISK
FLAG_EMAIL
FLAG_OWN_CAR
FLAG_OWN_REALTY
CODE_GENDER
FLAG_MOBIL
Relative Importance

Conclusion

In conclusion,

- With less liability and higher education, applicants tend to get their credit cards approved
- More married couples with no children tend to apply for credit cards
- Using XGBoost with RandomizedSearchCV's best parameters gives us an outstanding accuracy of 77% as compared to other algorithms
- When we assessed the most important features that help determine our outcome, whether a credit card should be approved or not, we saw that age, days since employed and annual income played a crucial role
- We believe that the model we built will be able to perform well in determining an applicant's credit rating, risk factor and a decision on whether to approve credit card or not given the applicant's history