1. Credit card applications

Commercial banks receive a lot of applications for credit cards. Many of them get rejected for many reasons, like high loan balances, low income levels, or too many inquiries on an individual's credit report, for example. Manually analyzing these applications is mundane, error-prone, and time-consuming (read: perfect task for ML).

Luckily, this task can be automated with the power of machine learning and pretty much every commercial bank does so nowadays. In this notebook, we will build an automatic credit card approval predictor using machine learning techniques, just like the real banks do.

We'll use the Credit Card Approval dataset from the UCI Machine Learning Repository. The structure of this notebook is as follows:

-First, we will start off by loading and viewing the dataset.

-We will see that the dataset has a mixture of both numerical and non-numerical features, that it contains values from different ranges, plus that it contains a number of missing entries.

-We have to preprocess the dataset to ensure the machine learning model we choose can make good predictions.

-Next we will do some exploratory data analysis.

-Finally, we will build a machine learning model that can predict if an individual's application for a credit card will be accepted.

First, loading and viewing the dataset. We find that since this data is confidential, the contributor of the dataset has anonymized the feature names.

In [1]:
```
# Import pandas
import pandas as pd

dr = pd.read_csv('cc_approvals.csv',header='None')

dr
```

Out[1]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	b	30.83	0.000	u	g	w	v	1.25	t	t	01	f	g	00202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	06	f	g	00043	560	+
2	a	24.50	0.500	u	g	q	h	1.50	t	f	0	f	g	00280	824	+
3	b	27.83	1.540	u	g	w	v	3.75	t	t	05	t	g	00100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	00120	0	+
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
685	b	21.08	10.085	y	p	e	h	1.25	f	f	0	f	g	00260	0	-
686	a	22.67	0.750	u	g	c	v	2.00	f	t	2	t	g	00200	394	-
687	a	25.25	13.500	y	p	ff	ff	2.00	f	t	1	t	g	00300	1	-
688	b	17.92	0.205	u	g	aa	v	0.04	f	f	0	f	g	00280	750	-
689	b	35.00	3.375	u	g	c	h	8.29	f	f	0	t	g	00000	0	-

690 rows × 16 columns

2. Inspecting the applications

The features of this dataset have been anonymized to protect the privacy. The probable features in a typical credit card application are Gender, Age, Debt, Married, BankCustomer, EducationLevel, Ethnicity, YearsEmployed, PriorDefault, Employed, CreditScore, DriversLicense, Citizen, ZipCode, Income and finally the ApprovalStatus. This gives us a pretty good starting point, and we can map these features with respect to the columns in the dataset.

As we can see from our first glance at the data, the dataset has a mixture of numerical and non-numerical features. This can be fixed with some preprocessing, but before we do that, let's learn about the dataset a bit more to see if there are other dataset issues that need to be fixed.

In [2]:
```
dr.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 690 entries, 0 to 689
Data columns (total 16 columns):
#   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       690 non-null    object 
 1   1       690 non-null    object 
 2   2       690 non-null    float64
 3   3       690 non-null    object 
 4   4       690 non-null    object 
 5   5       690 non-null    object 
 6   6       690 non-null    object 
 7   7       690 non-null    float64
 8   8       690 non-null    object 
 9   9       690 non-null    object 
 10  10      690 non-null    int64  
 11  11      690 non-null    object 
 12  12      690 non-null    object 
 13  13      690 non-null    object 
 14  14      690 non-null    int64  
 15  15      690 non-null    object 
dtypes: float64(2), int64(2), object(12)
memory usage: 86.4 KB
```

In [3]:
```
dr.describe()
```

Out[3]:

	2	7	10	14
count	690.000000	690.000000	690.000000	690.000000
mean	4.758725	2.223406	2.400000	1017.385507
std	4.978163	3.346513	4.86294	5210.102598
min	0.000000	0.000000	0.000000	0.000000
25%	1.000000	0.165000	0.000000	0.000000
50%	2.750000	1.000000	0.000000	5.000000
75%	7.207500	2.625000	3.000000	395.500000
max	28.000000	28.500000	67.000000	100000.000000

In [4]:
```
dr.tail(20)
```

Out[4]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
670	b	47.17	5.835	u	g	w	v	5.500	f	f	0	f	g	00445	150	-
671	b	25.83	12.835	u	g	cc	v	0.500	f	f	0	f	g	00000	2	-
672	a	50.25	0.835	u	g	ff	ff	2.000	f	f	0	f	g	00240	117	-
673	NaN	29.50	2.000	y	p	e	h	2.000	f	f	0	f	g	00256	17	-
674	a	37.33	2.500	u	g	i	h	0.210	f	f	0	f	g	00260	246	-
675	a	41.58	1.040	u	g	aa	v	0.665	f	f	0	f	g	00240	237	-
676	a	30.58	10.665	u	g	q	h	0.085	t	t	12	t	g	00129	3	-
677	b	19.42	7.250	u	g	m	v	0.540	f	t	1	f	g	00100	1	-
678	a	17.92	10.210	u	g	ff	ff	0.000	f	f	0	f	g	00000	50	-
679	a	20.08	1.250	u	g	c	v	0.000	f	f	0	f	g	00080	0	-
680	b	19.50	0.290	u	g	k	v	0.290	f	f	0	f	g	00280	364	-
681	b	27.83	1.000	y	p	e	h	3.000	f	f	0	f	g	00179	537	-
682	b	17.08	3.290	u	g	i	v	2.000	t	f	2	t	g	00240	0	-
683	b	36.42	0.750	y	p	d	v	0.585	f	f	0	f	g	00240	3	-
684	b	40.58	3.290	u	g	m	v	3.500	f	f	0	t	g	00400	0	-
685	b	21.08	10.085	y	p	e	h	1.250	f	f	0	f	g	00260	0	-
686	a	22.67	0.750	u	g	c	v	2.000	f	t	2	t	g	00200	394	-
687	a	25.25	13.500	y	p	ff	ff	2.000	f	t	1	t	g	00300	1	-
688	b	17.92	0.205	u	g	aa	v	0.040	f	f	0	f	g	00280	750	-
689	b	35.00	3.375	u	g	c	h	8.290	f	f	0	t	g	00000	0	-

3. Dealing with missing values

Our dataset contains both numeric and non-numeric data (specifically data that are of float64, int64 and object types).The features 2, 7, 10 and 14 contain numeric values (of types float64, float64, int64 and int64 respectively) and all the other features contain non-numeric values.

The dataset also contains values from several ranges. Some features have a value range of 0 - 28, some have a range of 2 - 67, and some have a range of 1017 - 100000. Apart from these, we can get useful statistical information (like mean, max, and min) about the features that have numerical values.

Finally, the dataset has missing values. The missing values in the dataset are labeled with '?', which can be seen in the last cell's output.

Now, let's temporarily replace these missing value question marks with NaN.

In [5]:
```
import numpy as np

# Replace the '?'s with NaN
dr = dr.replace('?',np.nan)

dr.tail(20)
```

Out[5]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
670	b	47.17	5.835	u	g	w	v	5.500	f	f	0	f	g	00445	150	-
671	b	25.83	12.835	u	g	cc	v	0.500	f	f	0	f	g	00000	2	-
672	a	50.25	0.835	u	g	ff	ff	2.000	f	f	0	f	g	00240	117	-
673	NaN	29.50	2.000	y	p	e	h	2.000	f	f	0	f	g	00256	17	-
674	a	37.33	2.500	u	g	i	h	0.210	f	f	0	f	g	00260	246	-
675	a	41.58	1.040	u	g	aa	v	0.665	f	f	0	f	g	00240	237	-
676	a	30.58	10.665	u	g	q	h	0.085	t	t	12	t	g	00129	3	-
677	b	19.42	7.250	u	g	m	v	0.540	f	t	1	f	g	00100	1	-
678	a	17.92	10.210	u	g	ff	ff	0.000	f	f	0	f	g	00000	50	-
679	a	20.08	1.250	u	g	c	v	0.000	f	f	0	f	g	00080	0	-
680	b	19.50	0.290	u	g	k	v	0.290	f	f	0	f	g	00280	364	-
681	b	27.83	1.000	y	p	e	h	3.000	f	f	0	f	g	00179	537	-
682	b	17.08	3.290	u	g	i	v	2.000	t	f	2	t	g	00240	0	-
683	b	36.42	0.750	y	p	d	v	0.585	f	f	0	f	g	00240	3	-
684	b	40.58	3.290	u	g	m	v	3.500	f	f	0	t	g	00400	0	-
685	b	21.08	10.085	y	p	e	h	1.250	f	f	0	f	g	00260	0	-
686	a	22.67	0.750	u	g	c	v	2.000	f	t	2	t	g	00200	394	-
687	a	25.25	13.500	y	p	ff	ff	2.000	f	t	1	t	g	00300	1	-
688	b	17.92	0.205	u	g	aa	v	0.040	f	f	0	f	g	00280	750	-
689	b	35.00	3.375	u	g	c	h	8.290	f	f	0	t	g	00000	0	-

We replaced all the question marks with NaNs. This will help us in the next missing value treatment that we are going to do.

Ignoring missing values can affect the performance of a machine learning model heavily. While ignoring the missing values our machine learning model may miss out on information about the dataset that may be useful for its training. Then, there are many models which cannot handle missing values implicitly.

So, to avoid this problem, we are going to impute the missing values with a strategy called mean imputation.

In [6]:
```
dr.fillna(dr.mean(),inplace=True)
dr.isnull().sum()
```

Out[6]:

```
0     12
1     12
2      0
3      6
4      6
5      9
6      9
7      0
8      9
9      9
10     0
11     0
12     9
13    13
14     0
15     0
dtype: int64
```

We have taken care of the missing values present in the numeric columns. There are still some missing values to be imputed for columns 0, 1, 3, 4, 5, 6 and 13. All of these columns contain non-numeric data and this why the mean imputation strategy would not work here. This needs a different treatment.

We are going to impute these missing values with the most frequent values as present in the respective columns.

In [7]:
```
# Iterate over each column of dataframe

for col in dr.columns:
    # Check if the column is of object type
    if dr[col].dtype == 'object':
        # Impute with the most frequent value
        dr = dr.fillna(dr[col].value_counts().index[0])

dr.isnull().sum()
```

Out[7]:

```
0     0
1     0
2     0
3     0
4     0
5     0
6     0
7     0
8     0
9     0
10    0
11    0
12    0
13    0
14    0
15    0
dtype: int64
```

The missing values are now successfully handled.

4. Preprocessing the data

There is still some minor but essential data preprocessing needed before we proceed towards building our machine learning model. Three main tasks are:

1.Convert the non-numeric data into numeric.

2.Split the data into train and test sets.

3.Scale the feature values to a uniform range.

First, we will convert all the non-numeric values into numeric ones. We do this because not only it results in a faster computation but also many machine learning models (like XGBoost) require the data to be in a strictly numeric format. We will do this by using a technique called label encoding.

In [8]:
```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

for col in dr.columns:
    # Compare if the dtype is object
    if dr[col].dtype=='object':
        # Use LabelEncoder to do the numeric transformation
        dr[col] = le.fit_transform(dr[col])
```

5. Splitting the dataset into train and test sets

We have converted all the non-numeric values to numeric ones.

Now, we split our data into train set and test set to prepare our data for two different phases of machine learning modeling: training and testing. Ideally, no information from the test data should be used to scale the training data or should be used to direct the training process of a machine learning model. Hence, we first split the data and then apply the scaling.

Features like DriversLicense and ZipCode are not as important as the other features in the dataset for predicting credit card approvals. We should drop them to design our machine learning model with the best set of features.

In [9]:
```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Drop the features 11 and 13 and convert the Dataframe to a Numpy array
dr = dr.drop([11,13],axis=1)

dr = dr.values

# Segregate features and labels into separate variables
X, y = dr[:,0:13] , dr[:,13]

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.3,random_state=0)
```

```
C:\Users\Pranav.LAPTOP-HOVCQ7L4\anaconda3\lib\importlib_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may indicate binary incompatibility. Expected 192 from C header, got 216 from PyObject
  return f(*args, **kwds)
```

In [10]:
```
# Import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler

# Instantiate MinMaxScaler and use it to rescale X_train and X_test
scaler = MinMaxScaler(feature_range=(0, 1))
rescaledX_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)
```

```
C:\Users\Pranav.LAPTOP-HOVCQ7L4\anaconda3\lib\importlib_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may indicate binary incompatibility. Expected 192 from C header, got 216 from PyObject
  return f(*args, **kwds)
```

In [11]:
```
# Import LogisticRegression
from sklearn.linear_model import LogisticRegression

# Instantiate a LogisticRegression classifier with default parameter values
logreg = LogisticRegression()

# Fit logreg to the train set
logreg.fit(X_train, y_train)
```

```
C:\Users\Pranav.LAPTOP-HOVCQ7L4\anaconda3\lib\importlib_bootstrap.py:219: RuntimeWarning: numpy.ufunc size changed, may indicate binary incompatibility. Expected 192 from C header, got 216 from PyObject
  return f(*args, **kwds)
```

Out[11]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

6. Making predictions and evaluating performance

We will now evaluate our model on the test set with respect to classification accuracy. But we will also take a look the model's confusion matrix. In the case of predicting credit card applications, it is equally important to see if our machine learning model is able to predict the approval status of the applications as denied that originally got denied.

In [12]:
```
# Import confusion_matrix
from sklearn.metrics import confusion_matrix

# Use logreg to predict instances from the test set and store it
y_pred = logreg.predict(X_test)

# Get the accuracy score of logreg model and print it
print("Accuracy of logistic regression classifier: ", logreg.score(X_test, y_test))

# Print the confusion matrix of the logreg model
confusion_matrix(y_test, y_pred)
```

```
Accuracy of logistic regression classifier:  0.8743961352657005
```

Out[12]:

```
array([[ 91,   6],
       [ 23, 105]], dtype=int64)
```

Our model was able to yield an accuracy score of almost 88%

In [ ]:
```

```