1. Intro

With a lot of applications, came a lot of rejection, with various reasons; high loan, low income levels, or too many inquiries on an individual's credit report, for example. Analyzing these applications one by one is mundane, prone to making mistakes, and time-consuming. This task can be automated with machine learning and pretty much every commercial bank does so nowadays. In this notebook, we will build an credit card approval predictor using machine learning. We'll use the Credit Card Approval dataset from the UCI Machine Learning Repository.

Plans

First, start by taking a look at the dataset. We will see that the dataset has a mix numerical and non-numerical features, it contains values with different ranges, plus that it contains a number of missing entries. We will have to preprocess the dataset to ensure the machine learning model we choose can make good predictions. After our data is in good shape, we will do exploratory data analysis gain initial insights and build intuititions. Finally, we will build a machine learning model that can predict if an application for a credit card will be accepted.

First, loading and viewing the dataset. Since this data is confidential, the contributor of the dataset has anonymized the feature names.

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

# Load dataset
cc_apps = pd.read_csv('datasets/cc_approvals.data', header=None)

# Inspect data
cc_apps.head()
```

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	1	f	g	00202	0	+
1	a	58.67	4.460	u	g	q	h	3.04	t	t	6	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	5	t	g	00100	3	+
4	b	20.17	5.625	u	g	w	v	1.71	t	f	0	f	s	00120	0	+

2. Inspecting data

The dataset may appear a bit confusing, let's try to figure out what's important features of a credit card application. The features of this dataset have been anonymized to protect the privacy, but this blog gives us an overview of the probable features. 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 output.

As we can see from our data, the dataset has a mixture of numerical and non-numerical features, it needs to be changed into numerical data before we fit it into our machine learning model. This can be fixed with some preprocessing. Before we do that, let's do some EDA to learn about the data and to see if theres any issues to be fixed.

In [2]:
```
# Print summary statistics
cc_apps_description = cc_apps.describe()
print(cc_apps_description)

print("\n")

# Print DataFrame information
cc_apps_info = cc_apps.info()
print(cc_apps_info)

print("\n")

# Inspect missing values in the dataset
cc_apps.tail()
```

```
                2           7          10             14
count  690.000000  690.000000  690.00000     690.000000
mean     4.758725    2.223406    2.40000    1017.385507
std      4.978163    3.346513    4.86294    5210.102598
min      0.000000    0.000000    0.00000       0.000000
25%      1.000000    0.165000    0.00000       0.000000
50%      2.750000    1.000000    0.00000       5.000000
75%      7.207500    2.625000    3.00000     395.500000
max     28.000000   28.500000   67.00000  100000.000000


<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
None
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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	00200	1	-
688	b	17.92	0.205	u	g	aa	v	0.04	f	f	0	f	g	00280	750	-

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
689	b	35.00	3.375	u	g	c	h	8.29	f	f	0	t	g	00000	0	-

3. Handling the missing values (part i)

Before any process can be done to the data, we need to see if there are any issues:

- Our dataset contains both numeric and non-numeric data (`float64` , `int64` and `object`). All non-numeric must be encoded.
- The dataset also contains values from several ranges. Later we will scale it.
- The dataset has missing values, labeled with '?'.

Now, temporarily replace missing values question marks with NaN.

In [3]:
```
# Import numpy
import numpy as np

# Inspect missing values in the dataset
print(cc_apps.tail())

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

# Inspect the missing values again
cc_apps.tail()
```

```
     0      1       2  3  4   5   6     7  8  9   10 11 12     13   14 15
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  00200    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  -
```

Out[3]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
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	00200	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	-

4. Handling the missing values (part ii)

Replacing missing values from '?' to NaN will help us treating this missing value issue.

To treat missing values, theoritically we can just exclude missing values from our dataset, but that means we are losing other valuable information contained in other features. So, unless theres specific situation where data cannot be syntethically generated, removing uncomplete observation should be a last resort decision.

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

In [4]:
```
# Impute the missing values with mean imputation
cc_apps.fillna(cc_apps.mean(), inplace=True)

# Count the number of NaNs in the dataset to verify
print(cc_apps.isnull().sum())
```

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

5. Handling the missing values (part iii)

We have imputed the missing values in the numeric columns. There are still missing values present in non-numeric values. We couldnt calculate the mean of a letter, right?. well you couldnt, for this issue, we are going to impute the missing values using the most frequent value in that column.

In [5]:
```
# Iterate over each column of cc_apps
for col in cc_apps.columns:
    # Check if the column is of object type
    if cc_apps[col].dtypes == 'object':
        # Impute with the most frequent value
        cc_apps = cc_apps.fillna(cc_apps[col].value_counts().index[0])

# Count the number of NaNs in the dataset and print the counts to verify
print(cc_apps.isnull().sum())
```

```
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
```

6. Preprocessing data (part i)

All missing values is now handled.

There's still some essential steps to do, most machine learning model (even machine learning that accept categorical data will train faster with encoding) only accept numeric values, so it is important to convert the non numeric data into a numeric data. Not only that, we have to split the data into two sets; training set and test set. Also, dont forget to scale the data so we have uniform ranges across all of our data.

In [6]:
```
# Import LabelEncoder
from sklearn.preprocessing import LabelEncoder

# Instantiate LabelEncoder
le = LabelEncoder()

# Iterate over all the values of each column and extract their dtypes
for col in cc_apps.columns:
    # Compare if the dtype is object
    if cc_apps[col].dtypes=='object':
    # Use LabelEncoder to do the numeric transformation
        cc_apps[col]=le.fit_transform(cc_apps[col])
```

7. Splitting the dataset into train and test sets

Our data is now in good shape. Now, we split our data into train set and test set for two different phases of machine learning modeling: training and testing. The idea is that the model we going to train is not supposedly to 'know' what it will predict after the training. Also applies in scaling process.

Feature selection

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 [7]:
```
# 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
cc_apps = cc_apps.drop([11, 13], axis=1)
cc_apps = cc_apps.to_numpy()

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

# Split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X,
                                    y,
                                    test_size=.33,
                                    random_state=42)
```

8. Preprocessing data (part ii)

Our data is not splitted into two sets. final preprocessing step is scaling.

Why do we do this? lets simplify our problem to predict credit card approval from 13 features to 2 features, for example we only using `CreditScore` and `Income` to predict Approval, `CreditScore` usually come with up to 2 (3 if you are wilding) digit numbers, compared to

`Income` that have (in this data) 4 digits up to 5 digits. If we fit the model using the data without scaling it first, the model will heavily biased towards income.

Thats why scaling is important, here we are going to transform every features so it will have range between 0 to 1.

In [8]:
```
# 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)
rescaledX_test = scaler.transform(X_test)
```

9. Model Fitting

This is a classification task, a binary one, we only predict if an individuals is accepted or not. According to UCI, our dataset contains more instances that correspond to "Denied" status than instances corresponding to "Approved" status. Specifically, out of 690 instances, there are 383 (55.5%) applications that got denied and 307 (44.5%) applications that got approved.

We will use this information as a benchmark. A good machine learning model should be able to closely predict the status of the applications with respect to these statistics.

Which model should we pick? If we are to assume that the features we used are correlated to the target variable, simply, we can use Logistic Regression model.

In [9]:
```
# 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(rescaledX_train, y_train)
```

Out[9]:
```
LogisticRegression()
```

10. Making predictions and evaluations

How well does our model perform?

Evaluate our model on the test set with respect to classification accuracy. 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 model is able to predict the approval status of the applications also denied status that originally got denied. If our model is not performing well in this aspect, then it might end up approving the application that should have been approved.

Also add classification so we dont have to manually calculate the recall and precision.

In [10]:
```
# Import confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

# Use logreg to predict instances from the test set and store it
```

```
y_pred = logreg.predict(rescaledX_test)

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

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

# Classification Report
print(classification_report(y_test, y_pred))
```

```
Accuracy of logistic regression classifier:  0.8421052631578947
[[94  9]
 [27 98]]
              precision    recall  f1-score   support

         0.0       0.78      0.91      0.84       103
         1.0       0.92      0.78      0.84       125

    accuracy                           0.84       228
   macro avg       0.85      0.85      0.84       228
weighted avg       0.85      0.84      0.84       228
```

11. Tuning Model

The model is decent, it yields 84% accuracy.

For confusion matrix, the two elements on our first row are True Negatives and False Negatives repectively, and in the second row are our False Positve and True Positive respectively, so the model predict 'Denied' 94 correct and 9 false, on the 'Approved' status, the model predict 27 false, and 98 correct predictions.

To improve the model further. We can tune the model using a grid search of the model parameters to improve the model's ability to predict credit card approvals.

scikit-learn's implementation of logistic regression consists of different hyperparameters but we will grid search over the following two:

- tol
- max_iter

In [11]:
```
# Import GridSearchCV
from sklearn.model_selection import GridSearchCV

# Define the grid of values for tol and max_iter
tol = [0.01, 0.001, 0.0001]
max_iter = [100, 150, 200]

# Create a dictionary where tol and max_iter are keys and the lists of their values
param_grid = dict(tol=tol, max_iter=max_iter)
```

12. Finding the best performing model

We have defined the grid of hyperparameter values and put them into a single dictionary format which `GridSearchCV()` expects as its parameters. Now see if the model is improving.

Like any other sklearn classes, instantiate `GridSearchCV()` with `logreg` model as its estimator.

We will also instruct `GridSearchCV()` to perform a cross-validation of five folds.

In [12]:
```
# Instantiate GridSearchCV with the required parameters
grid_model = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=5)

# Fit grid_model to the data
grid_model.fit(rescaledX_train, y_train)

#predict using tuned model
grid_model_pred = grid_model.predict(rescaledX_test)

# Get the accuracy score of tuned logreg model and print it
print("Accuracy of tuned logistic regression classifier: ", grid_model.score(rescale

# Print the confusion matrix of the logreg model
print(confusion_matrix(y_test, grid_model_pred))

# Classification Report
print(classification_report(y_test, grid_model_pred))

# Summarize results
best_params =grid_model.best_params_
print("Best using %s" % (best_params))
```

```
Accuracy of tuned logistic regression classifier:  0.8421052631578947
[[94  9]
 [27 98]]
              precision    recall  f1-score   support

         0.0       0.78      0.91      0.84       103
         1.0       0.92      0.78      0.84       125

    accuracy                           0.84       228
   macro avg       0.85      0.85      0.84       228
weighted avg       0.85      0.84      0.84       228

Best using {'max_iter': 100, 'tol': 0.01}
```

In [13]:
```
hyperparam = ['tol', 'max_iter']
for i in hyperparam:
    print(logreg.get_params().get(i))
```

```
0.0001
100
```

After performing grid search the model decided to use 0.0001 as its best 'tol' parameter and the model doesnt seems to improve compared the default logistic regression.

13. Closing

While building this credit card predictor, we covered some of the most widely-known preprocessing steps such as **scaling**, **encoding**, and **imputation**. We finished with some **machine learning** model to predict if an individual's application would get approved or not given some information about that person.