Predicting **Credit Card Approval**

Your three-digit credit scores are not the only thing that card issuers evaluate when deciding whether to approve your application for a new credit card.

```
Credit scores.
Number of delinquencies.
Hard inquiries.
Credit card utilization rate.
Income.
Credit history.
```

Credit score cards are a common risk control method in the financial industry. It uses personal information and data submitted by credit card applicants to predict the probability of future defaults and credit card borrowings. The bank is able to decide whether to issue a credit card to the applicant. Credit scores can objectively quantify the magnitude of risk.

```
# Import pandas
import pandas as pd

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

# Inspect data
print(cc_apps.head(5))
```

```
   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  +
```

The output may appear a bit confusing at its first sight, but let's try to figure out the most important features of a credit card application. The features of this dataset have been anonymized to protect the privacy, but this blog gives us a pretty good 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 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.

```
# 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
# ... YOUR CODE FOR TASK 2 ...
```

```
                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
```

```
12    690 non-null object
13    690 non-null object
14    690 non-null int64
15    690 non-null object
dtypes: float64(2), int64(2), object(12)
memory usage: 86.3+ KB
None
```

3. Handling the missing values (part i)

We've uncovered some issues that will affect the performance of our machine learning model(s) if they go unchanged:

- Our dataset contains both numeric and non-numeric data (specifically data that are of `float64`, `int64` and `object` types). Specifically, 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, which we'll take care of in this task. 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.

```
# Import numpy
import numpy as np

# Inspect missing values in the dataset
print(cc_apps.tail(17))
```

```
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  00200    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  -
      0      1       2  3  4   5   6      7  8  9   10 11 12     13   14 15
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  f  t  12  t  g  00129    3  -
677    b  19.42   7.250  u  g   m   v  0.040  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  00000    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   d   h  3.000  f  f   0  f  g  00176  537  -
682    b  17.08   3.290  u  g   i   v  0.335  f  f   0  t  g  00140    2  -
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  s  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  00200    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  -
```

4. Handling the missing values (part ii)

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

An important question that gets raised here is *why are we giving so much importance to missing values*? Can't they be just ignored? Ignoring missing values can affect the performance of a machine learning model heavily. While ignoring the missing values our machine learning model

```
13    13
14     0
15     0
dtype: int64
```

5. Handling the missing values (part iii)

We have successfully 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. This is good practice when it comes to imputing missing values for categorical data in general.

```
# 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])
```

2. Split the data into train and test sets.
3. Scale the feature values to a uniform range.

First, we will be converting 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) (and especially the ones developed using scikit-learn) require the data to be in a strictly numeric format. We will do this by using a technique called label encoding.

```
# 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
```

8. Preprocessing the data (part ii)

The data is now split into two separate sets - train and test sets respectively. We are only left with one final preprocessing step of scaling before we can fit a machine learning model to the data.

Now, let's try to understand what these scaled values mean in the real world. Let's use `CreditScore` as an example. The credit score of a person is their creditworthiness based on their credit history. The higher this number, the more financially trustworthy a person is

```
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)
```

10. Making predictions and evaluating performance