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 (and time is money!). Luckily, this task can be automated with the power of machine learning and pretty much every commercial bank does so nowadays. In this notebook, an automatic credit card approval predictor is built using machine learning techniques, just like the real banks do.

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

First, we start off by loading and viewing the dataset. We 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 preprocess the dataset to ensure the machine learning model we choose can make good predictions. After our data is in good shape, we do some exploratory data analysis to build our intuitions. Finally, we 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. The data can be accessed at 'https://archive.ics.uci.edu/ml/datasets/credit+approval'

```
# Importing the data
import os
import urllib.request
urllib.request.urlretrieve('http://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/crx.data', 'cc_approvals.data')
```

```
('cc_approvals.data', <http.client.HTTPMessage at 0x7fe887e233d0>)
```

```
# Import pandas
import pandas as pd

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

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

```
   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 the applications

The features of this dataset have been anonymized to protect the privacy, but from 'https://medium.datadriveninvestor.com/predicting-credit-card-approvals-using-ml-techniques-9cd8eaeb5b8c', 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.

```
# 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
print(cc_apps.tail(17))
```

```
                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
```

```
    0      1       2  3  4   5   6      7  8  9   10 11 12     13   14 15
673  ?  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  -
```

3. Handling the missing values (Part I)

The 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. The missing values in the dataset are labeled with '?'. Temporarily, we can 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))

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

# Inspect the missing values again
```

```
print(cc_apps.tail(17))
```

```
     0      1       2  3  4   5   6      7  8  9   10 11 12     13   14 15
673  ?  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  -
       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 in the next missing value treatment.

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 such as LDA.

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

```
# 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 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])

# 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 the data (Part I)

The missing values are now successfully handled.

There is still some minor but essential data preprocessing needed before we proceed towards building our machine learning model. We are going to divide these remaining preprocessing steps into three main tasks:

Convert the non-numeric data into numeric. Split the data into train and test sets. 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.values:
    # 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

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

Now, we will 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.

Also, 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.

```
# 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.values

# 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=0.33,random_state=42)
```

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.

```
# 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.fit_transform(X_test)
```

9. Fitting a logistic regression model to the train set

Essentially, predicting if a credit card application will be approved or not is a classification task. 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.

This gives us a benchmark. A good machine learning model should be able to accurately predict the status of the applications with respect to these statistics.

Let's start our machine learning modeling with a Logistic Regression model (a generalized linear model).

```
# 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)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_logistic.py:940: ConvergenceWarning: lbfgs failed to converge (statu
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
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)
```

10. 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. If our model is not performing well in this aspect, then it might end up

```
# 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(rescaledX_test)

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

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

```
    Accuracy of logistic regression classifier:  0.8289473684210527
    [[ 84  19]
     [ 20 105]]
```

11. Grid searching and making the model perform better

Our model was pretty good! It was able to yield an accuracy score of almost 83%.

For the confusion matrix, the first element of the of the first row of the confusion matrix denotes the true negatives meaning the number of negative instances (denied applications) predicted by the model correctly. And the last element of the second row of the confusion matrix denotes the true positives meaning the number of positive instances (approved applications) predicted by the model correctly.

We can perform 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

```
# 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 are corresponding 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 converted them into a single dictionary format which GridSearchCV() expects as one of its parameters. Now, we will begin the grid search to see which values perform best.

We will instantiate GridSearchCV() with our earlier logreg model with all the data we have. Instead of passing train and test sets separately, we will supply X (scaled version) and y. We will also instruct GridSearchCV() to perform a cross-validation of five folds.

```
# Instantiate GridSearchCV with the required parameters
grid_model = GridSearchCV(estimator=logreg, param_grid=param_grid, cv=5)

# Use scaler to rescale X and assign it to rescaledX
rescaledX = scaler.fit_transform(X)

# Fit data to grid_model
grid_model_result = grid_model.fit(rescaledX,y)

# Summarize results
best_score, best_params = grid_model_result.best_score_, grid_model_result.best_params_
print("Best: %f using %s" % (best_score,best_params))
```

```
Best: 0.850725 using {'max_iter': 100, 'tol': 0.01}
```

13. Another classifier model - Random Forest

The best logistic regression model gives an accuracy of 85%, and this is corresponding to the hyperparameters 'max_iter' equal to 100 and 'tol' equal to 0.01.

Random Forest is another classifier that is very useful for handling complex data. This algorithm usually performs better than decision trees and can be used for real time prediction. We can test the performance of this classifer for our credit card dataset.

```
# Import Random Forest Classifier
from sklearn.ensemble import RandomForestClassifier

# Instantiate a Random Forest classifier with default parameter values
rf = RandomForestClassifier()

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

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, 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=None,
                       verbose=0, warm_start=False)
```

14. Making predictions and evaluating performance for Random Forest

The output shows the classifier along with its default hyperparameters. Using similar code as earlier, this classifier can be tested for its prediction capabilities and a confusion matrix can be constructed.

```
# Use rf to predict instances from the test set and store it
y_pred = rf.predict(rescaledX_test)

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

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

```
Accuracy of logistic regression classifier:  0.7324561403508771
[[82 21]
 [40 85]]
```

15. The best of both models

We can see that Random Forest Classifier only provides an accuracy of 73%. This does not imply the algorithm is necessarily worse than Logistic Regression. Random Forest has been shown to perform better with cross validation, and hyperparameter tuning can go a long way towards improving the accuracy, just like it did during grid search for Logistic Regression. However, for default parameters, Logistic Regression performs better than Random Forest for our dataset.

We end the notebook by storing the best-achieved score and the respective best parameters. Some improvements over this work include feature selection, ensemble methods and using techniques like stacking or boosting.