**Predicting Credit Card Approvals**

**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, 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](http://archive.ics.uci.edu/ml/datasets/credit+approval) 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 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 some exploratory data analysis to build our intuitions.
* 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.

Load and look at the dataset.

* Import the pandas library under the alias pd.
* Load the dataset, "datasets/cc\_approvals.data", into a pandas DataFrame called cc\_apps. Set the header argument to None.
* Print the first 5 rows of cc\_apps using the head() method.

### Good to know

For this project, it is recommended that you know basic Python programming, the pandas and numpy packages, some data preprocessing, and a little bit of machine learning. Here are some resources that may be helpful throughout the project:

* For a quick introduction to Python:
  + [DataCamp's Intro to Python for Data Science course](https://www.datacamp.com/courses/intro-to-python-for-data-science)
* For learning the basics of the pandas and numpy packages:
  + [Data Manipulation with pandas](https://www.datacamp.com/courses/data-manipulation-with-pandas)
  + [pandas Cheatsheet](https://www.datacamp.com/community/blog/python-pandas-cheat-sheet)
  + [NumPy Cheat Sheet](https://www.datacamp.com/community/blog/python-numpy-cheat-sheet)
* For data preprocessing:
  + [Preprocessing in Data Science (Part 1)](https://www.datacamp.com/community/tutorials/preprocessing-in-data-science-part-1-centering-scaling-and-knn)
  + [Preprocessing in Data Science (Part 2)](https://www.datacamp.com/community/tutorials/preprocessing-in-data-science-part-2-centering-scaling-and-logistic-regression)
  + [Preprocessing in Data Science (Part 3)](https://www.datacamp.com/community/tutorials/preprocessing-in-data-science-part-3-scaling-synthesized-data)
* For machine learning:
  + Google's [Machine Learning Crash Course](https://developers.google.com/machine-learning/crash-course/)
  + [Supervised Learning with scikit-learn](https://www.datacamp.com/courses/supervised-learning-with-scikit-learn)

Apart from the above, we encourage you to use your preferred search engine to find other useful resources.

## 

## 2. Inspecting the applications

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](http://rstudio-pubs-static.s3.amazonaws.com/73039_9946de135c0a49daa7a0a9eda4a67a72.html) 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.

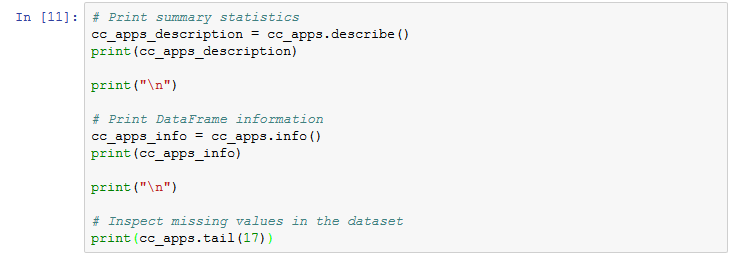
As we can see from our first glance at the data, the dataset has a mixture of numerical and non-numerical features. This can be fixed with some preprocessing, but before we do that, let's learn about the dataset a bit more to see if there are other dataset issues that need to be fixed.

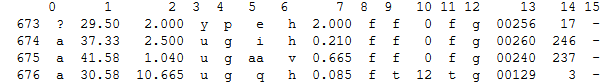
Inspect the structure, numerical summary, and specific rows of the dataset.

* Extract the summary statistics of the data using the describe() method of cc\_apps.
* Use the info() method of cc\_apps to get more information about the DataFrame.
* Print the last 17 rows of cc\_apps using the tail() method to display missing values

Helpful links:

* pandas tail() method [documentation](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.tail.html)





## 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.

Inspect the missing values in the dataset and replace the question marks with NaN.

* Import the numpy library under the alias np.
* Print the last 17 rows of the dataset.
* Replace the '?'s with NaNs using the replace() method.
* Print the last 17 rows of cc\_apps using the tail() method to confirm that the replace() method performed as expected.

Helpful links:

* pandas replace() method [documentation](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.replace.html)
* NumPy data types for [special values](https://docs.scipy.org/doc/numpy-1.13.0/user/misc.html)

## 

## 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 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.

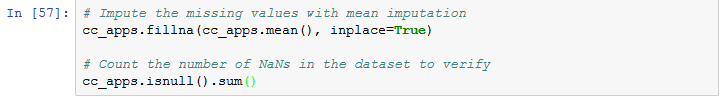
Impute the NaN values with the mean imputation approach.

* For the numeric columns, impute the missing values (NaNs) with pandas method fillna().
* Verify if the fillna() method performed as expected by printing the total number of NaNs in each column.

Remember that you have already marked all the question marks as NaNs. pandas provides fillna() to help you impute missing values with different strategies, mean imputation being one of them. pandas also has a mean() method to calculate the mean of a DataFrame. As your dataset contains both numeric and non-numeric data, for this task you will only impute the missing values (NaNs) present in the columns having numeric data-types (columns 2, 7, 10 and 14).

Helpful links:

* mean imputation [tutorial](https://machinelearningmastery.com/handle-missing-data-python/)
* pandas fillna() method [documentation](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html)
* pandas mean() method [documentation](https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.DataFrame.mean.html)
* pandas isnull() method [documentation](https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.isnull.html)



## 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](https://www.datacamp.com/community/tutorials/categorical-data) when it comes to imputing missing values for categorical data in general.

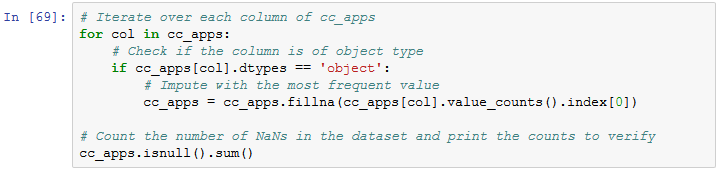
Impute the missing values in the non-numeric columns.

* Iterate over each column of cc\_apps using a for loop.
* Check if the data-type of the column is of object type by using the dtypes keyword.
* Using the fillna() method, impute the column's missing values with the most frequent value of that column with the value\_counts() method and index attribute and assign it to cc\_apps.
* Finally, verify if there are any more missing values in the dataset that are left to be imputed by printing the total number of NaNs in each column.

The column names of a pandas DataFrame can be accessed using columns attribute. The dtypes attribute provides the data type. In this part, object is the data type that you should be concerned about. The value\_counts() method returns the frequency distribution of each value in the column, and the index attribute can then be used to get the most frequent value.

Helpful links:

* pandas value\_counts() method [documentation](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.value_counts.html)
* Accessing the index attribute in a [tutorial](https://www.datacamp.com/community/tutorials/categorical-data)
* Method chaining with pandas [tutorial](https://www.datacamp.com/community/tutorials/pandas-idiomatic)



## 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:

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 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](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html).

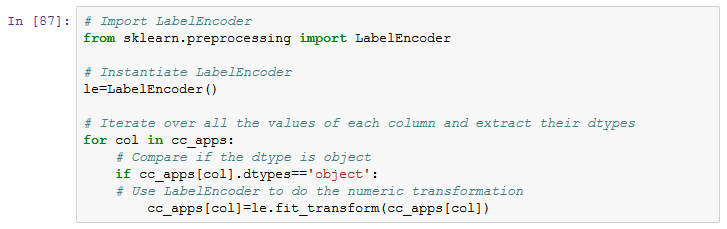
Convert the non-numeric values to numeric.

* Import the LabelEncoder class from sklearn.preprocessing module.
* Instantiate LabelEncoder() into a variable le.
* Iterate over all the **values** of each column cc\_apps and check their data types using a for loop.
* If the data type is found to be of object type, label encode it to transform into numeric (such as int64) type.

The values of each column a pandas DataFrame can be accessed using columns and values attributes consecutively. The dtypes attribute provides the data type. In this part, object is the data type that you should be concerned about.

Helpful links:

* Checking data types of the columns in a DataFrame [Stack Overflow answer](https://stackoverflow.com/questions/40353079/pandas-how-to-check-dtype-for-all-columns-in-a-dataframe)
* sklearn LabelEncoder class [documentation](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html)



## 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. In Data Science literature, this is often referred to as feature selection.

Split the preprocessed dataset into train and test sets.

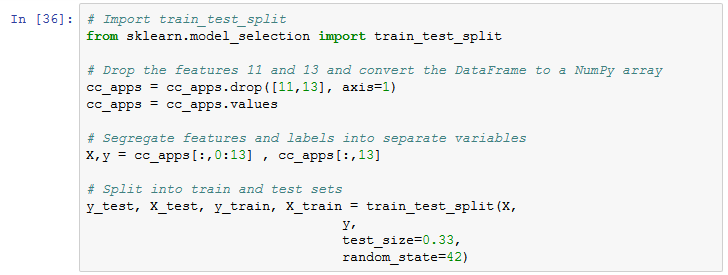
* Import train\_test\_split from the sklearn.model\_selection module.
* Drop features 11 and 13 using the drop() method and convert the DataFrame to a NumPy array using .values.
* Segregate the features and labels into X and y (the column with index 13 is the label column).
* Using the train\_test\_split() method, split the data into train and test sets with a split ratio of 33% (test\_size argument) and set the random\_state argument to 42.

A NumPy array can be segregated using array slicing. Before slicing, take note of the total number of columns that should be present in the array after dropping features 11 and 13.

Setting random\_state ensures the dataset is split with same sets of instances every time the code is run.

Helpful links:

* pandas drop() method [documentation](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.drop.html)
* NumPy indexing and slicing [tutorial](https://www.tutorialspoint.com/numpy/numpy_indexing_and_slicing.htm)
* sklearn train\_test\_split() method [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)



## 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 considered to be. So, a CreditScore of 1 is the highest since we're rescaling all the values to the range of 0-1.

Drop DriversLicense and ZipCode features and rescale the data.

* Import the MinMaxScaler class from the sklearn.preprocessing module.
* Instantiate MinMaxScaler class in a variable called scaler with the feature\_range parameter set to (0,1).
* Fit the scaler to X\_train and transform the data, assigning the result to rescaledX\_train.
* Use the scaler to transform X\_test, assigning the result to rescaledX\_test.

When a dataset has varying ranges as in this credit card approvals dataset, one a small change in a particular feature may not have a significant effect on the other feature, which can cause a lot of problems when predictive modeling.

Helpful links:

* sklearn's MinMaxScaler class [documentation](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)

## 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](https://en.wikipedia.org/wiki/Statistical_classification) task. [According to UCI](http://archive.ics.uci.edu/ml/machine-learning-databases/credit-screening/crx.names), 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.

Which model should we pick? A question to ask is: are the features that affect the credit card approval decision process correlated with each other? Although we can measure correlation, that is outside the scope of this notebook, so we'll rely on our intuition that they indeed are correlated for now. Because of this correlation, we'll take advantage of the fact that generalized linear models perform well in these cases. Let's start our machine learning modeling with a Logistic Regression model (a generalized linear model).

Fit a LogisticRegression classifier with rescaledX\_train and y\_train.

* Import LogisticRegression from the sklearn.linear\_model module.
* Instantiate LogisticRegression into a variable named logreg with default values.
* Fit rescaledX\_train and y\_train to logreg using the fit() method.

If a quick refresher on logistic regression's working mechanism is needed, check out this [tutorial](https://www.datacamp.com/community/tutorials/understanding-logistic-regression-python).

Helpful links:

* sklearn Logistic Regression [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)

## 10. Making predictions and evaluating performance

But how well does our model perform?

We will now evaluate our model on the test set with respect to [classification accuracy](https://developers.google.com/machine-learning/crash-course/classification/accuracy). But we will also take a look the model's [confusion matrix](http://www.dataschool.io/simple-guide-to-confusion-matrix-terminology/). 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 approving the application that should have been approved. The confusion matrix helps us to view our model's performance from these aspects.

Make predictions and evaluate performance.

* Import confusion\_matrix() from sklearn.metrics module.
* Use predict() on rescaledX\_test (which contains instances of the dataset that logreg has not seen until now) and store the predictions in a variable named y\_pred.
* Print the accuracy score of logreg using the score(). Don't forget to pass rescaledX\_test and y\_test to the score() method.
* Call confusion\_matrix() with y\_test and y\_pred to print the confusion matrix.

Helpful links:

* sklearn confusion matrix [documentation](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)

## 11. Grid searching and making the model perform better

Our model was pretty good! It was able to yield an accuracy score of almost 84%.

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.

Let's see if we can do better. We can perform a [grid search](https://machinelearningmastery.com/how-to-tune-algorithm-parameters-with-scikit-learn/) of the model parameters to improve the model's ability to predict credit card approvals.

[scikit-learn's implementation of logistic regression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) consists of different hyperparameters but we will grid search over the following two:

* tol
* max\_iter

Define the grid of parameter values for which grid searching is to be performed.

* Import GridSearchCV from the sklearn.model\_selection module.
* Define the grid of values for tol and max\_iter parameters into tol and max\_iter lists respectively.
* For tol, define the list with values 0.01, 0.001 and 0.0001. For max\_iter, define the list with values 100, 150 and 200.
* Using the dict() method, create a dictionary where tol and max\_iter are keys, and the lists of their values are the corresponding values. Name this dictionary as param\_grid.

Grid search can be very exhaustive if the model is very complex and the dataset is extremely large. Luckily, that is not the case for this project.

## 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](https://www.dataschool.io/machine-learning-with-scikit-learn/) of five folds.

We'll end the notebook by storing the best-achieved score and the respective best parameters.

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

Find the best score and best parameters for the model using grid search.

* Instantiate GridSearchCV() with the attributes set as estimator = logreg, param\_grid = param\_grid and cv = 5 and store this instance in grid\_model variable.
* Use scaler (which you created in Task-8) rescale X and assign it to rescaledX.
* Fit rescaledX and y to grid\_model and store the results in grid\_model\_result.
* Call the best\_score\_ and best\_params\_ attributes on the grid\_model\_result variable, then print both.

Grid searching is a process of finding an optimal set of values for the parameters of a certain machine learning model. This is often known as hyperparameter optimization which is an active area of research. Note that, here we have used the word parameters and hyperparameters interchangeably, but they are not exactly the same.

Helpful links:

* Hyperparameter Optimization in Machine Learning Models [tutorial](https://www.datacamp.com/community/tutorials/parameter-optimization-machine-learning-models?tap_a=5644-dce66f&tap_s=3575)

