Udacity Machine Learning Nanodegree

Credit Card Approval Prediction using Machine Learning

Muskaan Patel
24 December 2020

- ## Definition:

Project Overview:

This project comes under the domain of Application of Machine Learning and Data Science in the Finance Industry. Data Science has inarguably been the trendiest domain of the decade, asserting its need in every single sphere of corporate life. It was not long ago when we discovered the massive potential of incorporating ML/AI in the financial world. Now, the very idea of two being disjointed sounds strange. Data Science has been incremental in providing powerful insights and helped massively increase the efficiency, helping everyone from a scalp trader to a long-term debt investor. Accurate predictions, unbiased analysis, powerful tools that run through millions of rows of data in the blink of an eye has transformed the industry in ways we could've never imagined.

In this project, I'll be using Credit Card Approval Dataset from UCI Machine Learning Repository. A snippet of the dataset is displayed below.

	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	+

On analyzing the dataframe, I find that since the data is confidential, the contributor of this dataset has anonymized the feature names. The features of this dataset have been anonymized to protect the privacy, but upon more research I found a blog that 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 [1,4].

The link to the dataset is: http://archive.ics.uci.edu/ml/datasets/credit+approval

Problem Statement:

Nowadays, banks receive a lot of applications for issuance of credit cards. Many of them rejected for many reasons, like high-loan balances, low-income levels, or too many inquiries on an individual's credit report. Manually analyzing these applications is error-prone and a time-consuming process. Luckily, this task can be automated with the power of machine learning and pretty much every bank does so nowadays. In this project, I will build an automatic credit card approval predictor using machine learning techniques, just like the real banks do [1].

The accurate assessment of consumer credit risk is of uttermost importance for lending organizations. Credit scoring is a widely used technique that helps financial institutions evaluates the likelihood for a credit applicant to default on the financial obligation and decide whether to grant credit or not. The precise judgment of the creditworthiness of applicants allows financial institutions to increase the volume of granted credit while minimizing possible losses. The credit industry has experienced a tremendous growth in the past few decades. The increased number of potential applicants impelled the development of sophisticated techniques that automate the credit approval procedure and supervise the financial health of the borrower. The goal of a credit scoring model is to classify credit applicants into two classes: the "good credit" class that is liable to reimburse the financial obligation and the "bad credit" class that should be denied credit due to the high probability of defaulting on the financial obligation. The classification is contingent on sociodemographic characteristics of the borrower (such as age, education level, occupation and income), the repayment Performance on previous loans and the type of loan [2].

This project aims to apply multiple machine learning algorithms to analyze the default payment of credit cards. By using the financial institutions client data provided by UCI MachineLearning Repository, we will evaluate and compare the performance of the model candidates in order to choose the most robust model. Moreover, we will also decide which are important features in our best predictive model [3].

A proposed solution strategy is as follows:

- First, I will start off by loading and viewing the dataset.
- I will have to pre-process 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.
- After uploading the data to S3, I'll create an Estimator along with Hyperparameters for our preferred ML algorithm.
- I will build and deploy a machine learning model using endpoint configuration that can predict if an individual's application for a credit card will be accepted.
- I'll compare a few different ML algorithms along with their accuracy metrics and choose the best one.

Metrics:

The evaluation metrics proposed are appropriate given the context of the data, the problem statement, and the intended solution. The performance of each classification model is evaluated using three statistical measures; classification accuracy, sensitivity and specificity.

A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. The confusion matrix itself is relatively simple to understand, but the related terminology can be confusing [10].

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

It is using true positive (TP), true negative (TN), false positive (FP) and false negative (FN). The percentage of Correct/Incorrect classification is the difference between the actual and predicted values of variables.

True Positive (TP) is the number of correct predictions that an instance is true, or in other words; it is occurring when the positive prediction of the classifier coincided with a positive prediction of target attribute.

True Negative (TN) is presenting a number of correct predictions that an instance is false, (i.e.) it occurs when both the classifier, and the target attribute suggests the absence of a positive prediction.

The False Positive (FP) is the number of incorrect predictions that an instance is true.

Finally, False Negative (FN) is the number of incorrect predictions that an instance is false. Figure above shows the confusion matrix for a two-class classifier [10].

For example, lets take the case of a model predicting emails as spam, then:

- True positives (TP): the cases for which the classifier predicted ‘spam’ and the emails were actually spam.
- True negatives (TN): the cases for which the classifier predicted ‘not spam’ and the emails were actually real.
- False positives (FP): the cases for which the classifier predicted ‘spam’ but the emails were actually real.
- False negatives (FN): the cases for which the classifier predicted ‘not spam’ but the

emails were actually spam.

Classification accuracy is defined as the ratio of the number of correctly classified cases and is equal to the sum of TP and TN divided by the total number of cases (TN + FN + TP + FP).

$$Accuracy = \frac{TP + TN}{TN + FN + TP + FP}$$

Precision is defined as the number of true positives (TP) over the number of true positives plus the number of false positives (FP).

$$Precision = \frac{TP}{TP + FP}$$

Recall is defined as the number of true positives (TP) over the number of true positives plus the number of false negatives (FN).

$$Recall = \frac{TP}{TP + FN}$$

- # Analysis:

Data Exploration:

In statistics, exploratory data analysis (EDA) is an approach to analyzing data sets to summarize their main characteristics, often with visual methods. A statistical model can be used or not, but primarily EDA is for seeing what the data can tell us beyond the formal modeling or hypothesis testing task [2].

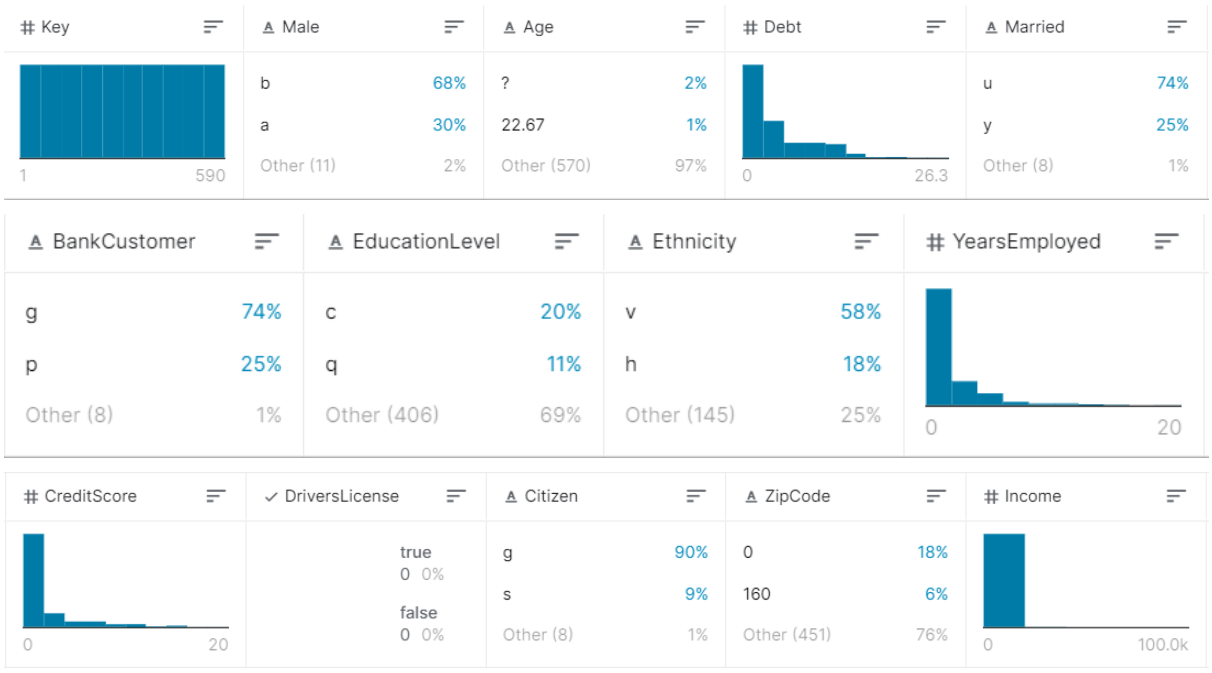
EDA tackle specific tasks such as:

- Spotting mistakes and missing data;
- Mapping out the underlying structure of the data;
- Identifying the most important variables;
- Listing anomalies and outliers;
- Estimating parameters and figuring out the associated confidence intervals or margins of error.
- Specific statistical functions and techniques you can perform with these tools include:
- Clustering and dimension reduction techniques, which help you to create graphical displays of high dimensional data containing many variables;
- Univariate visualization of each field in the raw dataset, with summary statistics;
- Bivariate visualizations and summary statistics that allow you to assess the relationship between each variable in the dataset and the target variable you're looking at;

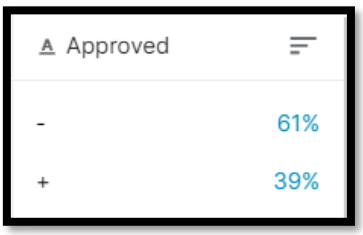
- Multivariate visualizations, for mapping and understanding interactions between different fields in the data;
- K-Means Clustering (creating “centres” for each cluster, based on the nearest mean); Predictive models, e.g. linear regression.

The first step in any analysis is to obtain the dataset and codebook. Both the dataset and the codebook can be downloaded for free from the UCI website. On analyzing the dataframe, I find that since the data is confidential, the contributor of this dataset has anonymized the feature names. The features of this dataset have been anonymized to protect the privacy, but upon more research I found a blog that 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 [1,4].

A descriptive distribution of my dataset is provided below. It describes the distribution of the various attributes of the data frame used [8].



The key label attribute, i.e the “Approved” column has a fairly balanced distribution. Its description is also displayed below [8].

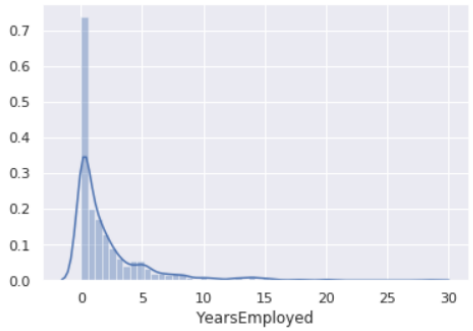


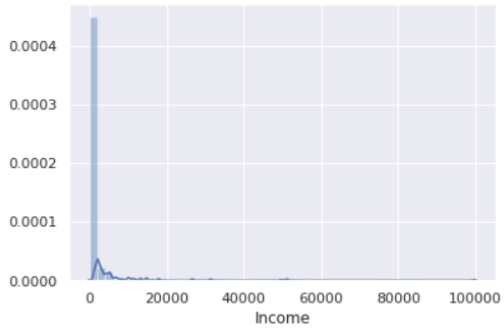
```
'data.frame':   689 obs. of  16 variables:
 $ Male          : num  1 1 0 0 0 0 1 0 0 0 ...
 $ Age           : chr  "58.67" "24.50" "27.83" "20.17" ...
 $ Debt          : num  4.46 0.5 1.54 5.62 4 ...
 $ Married       : chr  "u" "u" "u" "u" ...
 $ BankCustomer  : chr  "g" "g" "g" "g" ...
 $ EducationLevel: chr  "q" "q" "w" "w" ...
 $ Ethnicity     : chr  "h" "h" "v" "v" ...
 $ YearsEmployed : num  3.04 1.5 3.75 1.71 2.5 ...
 $ PriorDefault  : num  1 1 1 1 1 1 1 1 1 0 ...
 $ Employed      : num  1 0 1 0 0 0 0 0 0 0 ...
 $ CreditScore   : num  6 0 5 0 0 0 0 0 0 0 ...
 $ DriversLicense: chr  "f" "f" "t" "f" ...
 $ Citizen       : chr  "g" "g" "g" "s" ...
 $ ZipCode       : chr  "00043" "00280" "00100" "00120" ...
 $ Income        : num  560 824 3 0 0 ...
 $ Approved      : chr  "+" "+" "+" "+" ...
```

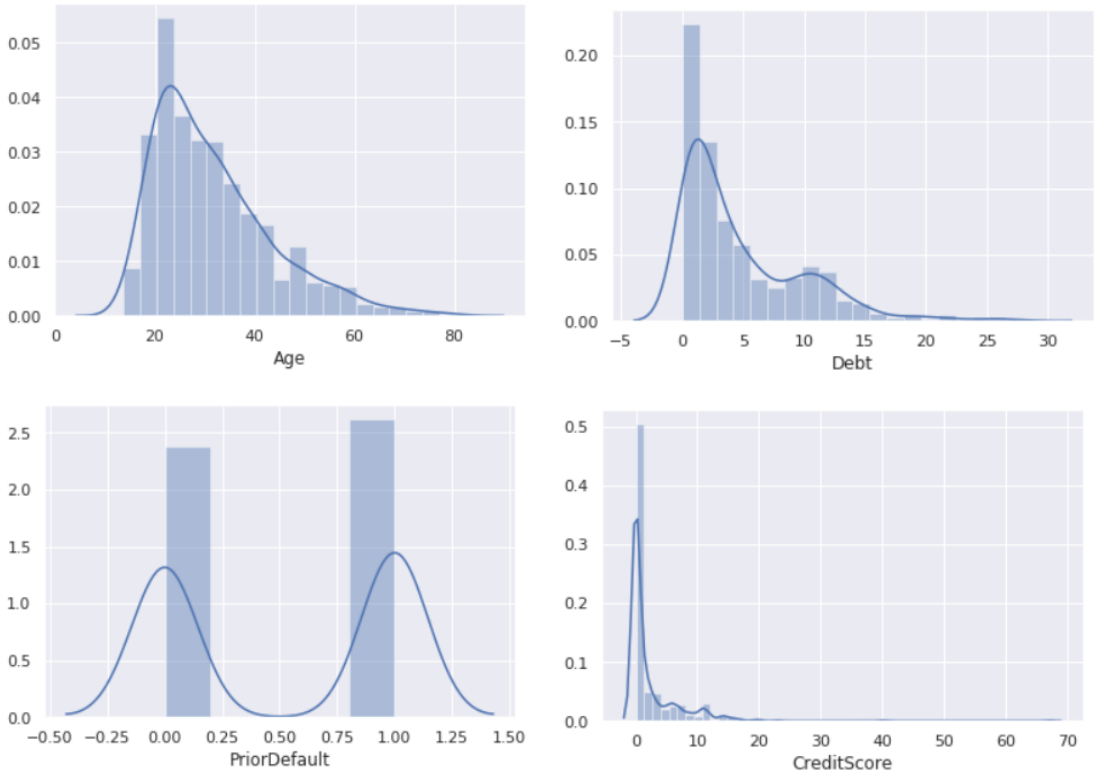
Exploratory Visualization:

The below figures displays DistPlot of various attributes like Age, Debt, PriorDefault, CreditScore, YearsEmployed and Income of our dataframe.

A Distplot or distribution plot, depicts the variation in the data distribution. Seaborn Distplot represents the overall distribution of continuous data variables [12].

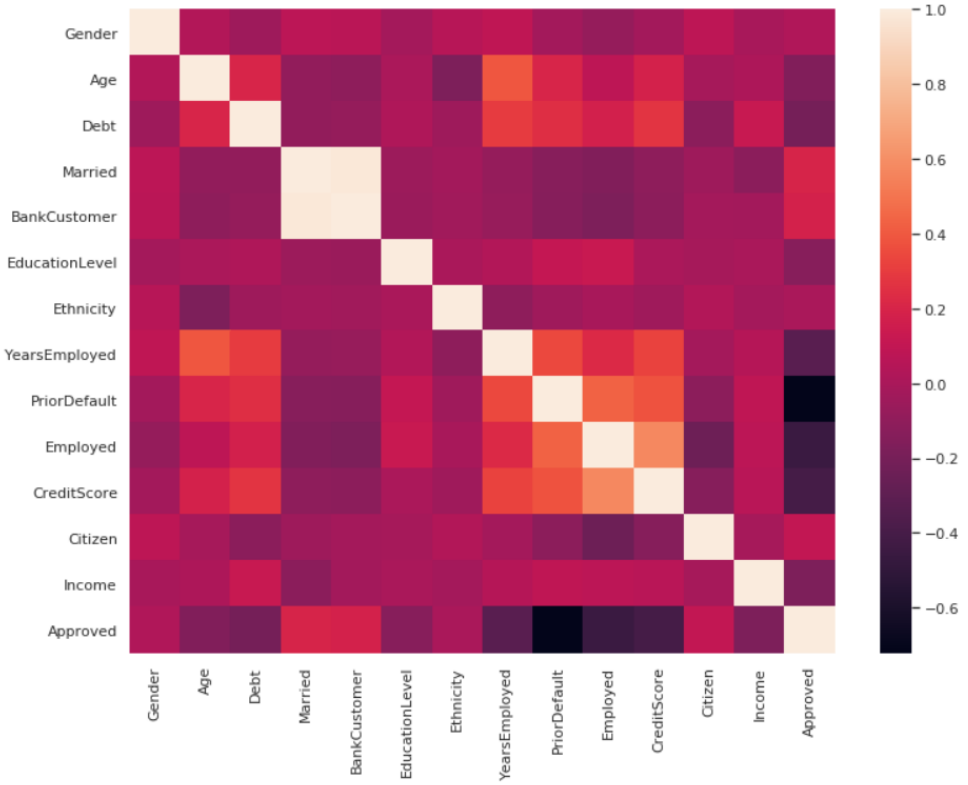




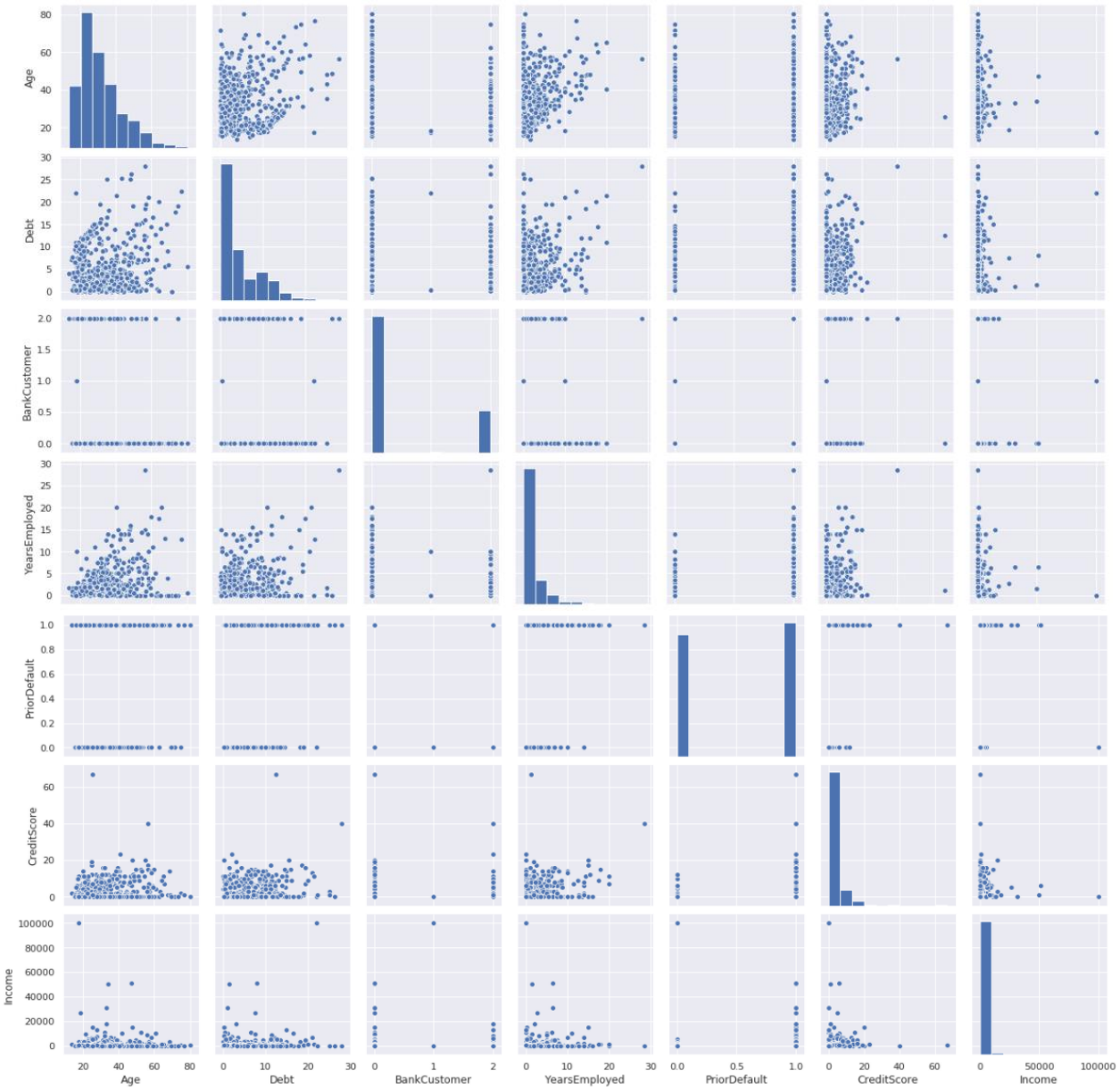


The Seaborn module along with the Matplotlib module is used to depict the distplot with different variations in it. The Distplot depicts the data by a histogram and a line in combination to it [12].

We also applied scatterplot matrix on whole dataset to visualize inherent relationships between variables in the data shown.



You already know that if you have a data set with many columns, a good way to quickly check correlations among columns is by visualizing the correlation matrix as a heatmap. To make a regular heatmap, we simply used the Seaborn heatmap function, with a bit of additional styling [13].



This is the ScatterPlot of attributes like Income, CreditScore, PriorDefault, YearsEmployed, BankCustomer, Debit and Age.

To plot multiple pairwise bivariate distributions in a dataset, you can use the pairplot() function. This shows the relationship for (n, 2) combination of variable in a DataFrame as a matrix of plots and the diagonal plots are the univariate plots [14].

Algorithms and Techniques:

Now that I've uploaded my training data, it's time to define and train a model! I've used the LinearLearner model for my project. A LinearLearner has two main applications:

1. For regression tasks in which a linear line is fit to some data points, and you want to produce a predicted output value given some data point (example: predicting house prices given square area).
2. For binary classification, in which a line is separating two classes of data and effectively outputs labels; either 1 for data that falls above the line or 0 for points that fall on or below the line.

In this case, I'll be using it for case 2, and I'll train it to separate data into our two classes: approved or not approved.

Linear models are supervised learning algorithms used for solving either classification or regression problems. For input, you give the model labelled examples (x, y). x is a high-dimensional vector and y is a numeric label. For binary classification problems, the label must be either 0 or 1. For multiclass classification problems, the labels must be from 0 to num_classes - 1. For regression problems, y is a real number. The algorithm learns a linear function, or, for classification problems, a linear threshold function, and maps a vector x to an approximation of the label y.

The Amazon SageMaker linear learner algorithm provides a solution for both classification and regression problems. With the SageMaker algorithm, you can simultaneously explore different training objectives and choose the best solution from a validation set. You can also explore a large number of models and choose the best. The best model optimizes either of the following:

Continuous objectives, such as mean square error, cross entropy loss, absolute error.

Discrete objectives suited for classification, such as F1 measure, precision, recall, or accuracy.

Compared with methods that provide a solution for only continuous objectives, the SageMaker linear learner algorithm provides a significant increase in speed over naive hyperparameter optimization techniques. It is also more convenient.

The linear learner algorithm requires a data matrix, with rows representing the observations, and columns representing the dimensions of the features. It also requires an additional column that contains the labels that match the data points. At a minimum, Amazon SageMaker linear learner requires us to specify input and output data locations, and objective type (classification or regression) as arguments. The feature dimension is also required. We can also specify additional parameters in the HyperParameters string map of the request body. These parameters control the optimization procedure, or specifics of the objective function that you train on. For example, the number of epochs, regularization, and loss type [15].

Benchmark Model:

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.
After doing a thorough research, I’ve found a number of papers on this topic. A number of algorithms have been used, more popular ones are Logistic Regression [2] Classification and Regression Trees (CART) [4]. But the maximum accuracy their model achieved was near to 76% [2]. I plan to exploit the AWS SageMaker’s interface and various Machine Learning features to achieve a good level of Explanatory Data Analysis, Feature Engineering, Data Pre-Processing, choosing an appropriate ML model and Hyperparameter Tuning to produce better and more accurate results.

The specific benchmark model that’ll be used for my project will be the Logistic Regression model along with 76% accuracy [2]. I’ll try to improve the other specific domains such as data pre-processing and exploratory data analysis to achieve a model with better accuracy and efficiency.

• Methodology:

Data Pre-Processing:

The binary values, such as Approved, need to be converted to 1s and 0s. I’ll need to do additional transformations such as filling in missing values. That process begins by first identifying which values are missing and then determining the best way to address them. We can remove them, zero them out, or estimate a plug value. A scan through the dataset shows that missing values are labeled with ‘?’. For each variable, we’ll convert the missing values to NA which R will interpret differently than a character value.

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 (of type object).

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 labelled with ‘?’, which can be seen in the last cell’s output. Now, I’ve temporarily replaced these missing value question marks with NaN.

So, to avoid this problem of missing values, I’ve to impute the missing values with a

strategy called mean imputation.

Even after this imputation, I still have pre-processing procedures to complete before I build out my ML models. Currently, I am still dealing with data broken down into numerical and categorical designations. To deal with this, I'll build-out another loop that will iterate over all of my columns, check if it's an object type, and use label encoder to create appropriate numerical values for those respective columns.

A note on LabelEncoder from the sklearn module pre-processing; it is an excellent tool and one that will designate target labels with a value between 0 and n_classes — 1.

The missing values are now successfully handled. There is still some minor but essential data pre-processing needed before we proceed towards building our machine learning model. We are going to divide these remaining pre-processing 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.

I will then split our data into train set and test set to prepare our data for two different phases of machine learning modelling: 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. I've dropped 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.

The data after being split into two separate sets — train and test sets respectively, are only left with one final pre-processing 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.

Our final dataframe after all the pre-processing looks something like this:

```
credit_drop.head()
```

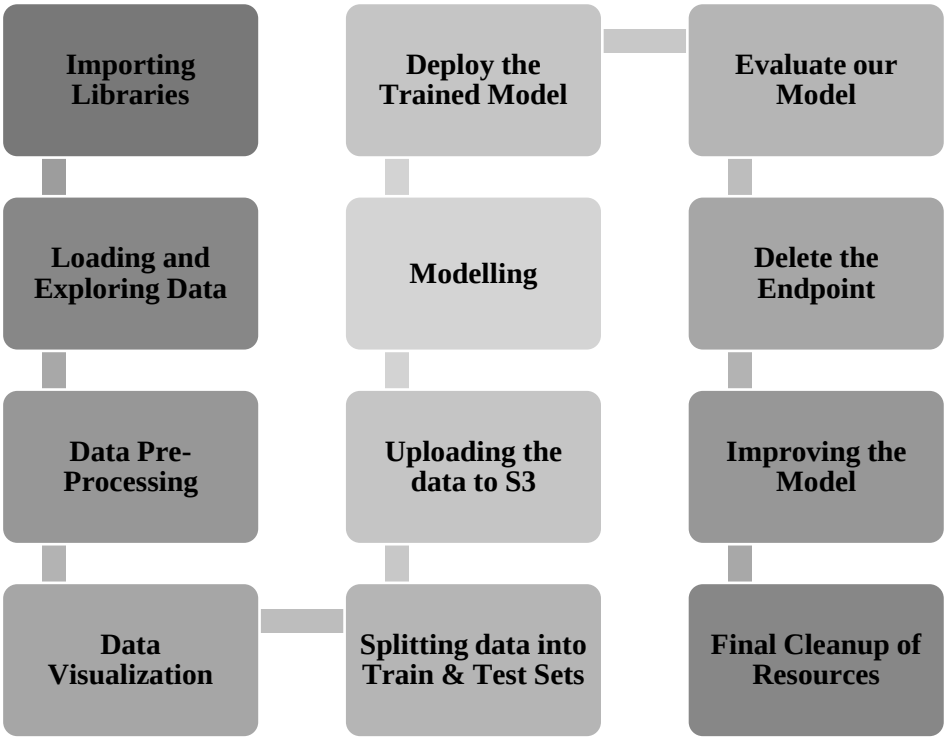
Gender	Age	Debt	Married	BankCustomer	EducationLevel	Ethnicity	YearsEmployed	PriorDefault	Employed	CreditScore	Citizen	Income	Approved
1	30.83	0.000	1	0	12	7	1.25	1	1	1	0	0	0
0	58.67	4.460	1	0	10	3	3.04	1	1	6	0	560	0
0	24.50	0.500	1	0	10	3	1.50	1	0	0	0	824	0
1	27.83	1.540	1	0	12	7	3.75	1	1	5	0	3	0
1	20.17	5.625	1	0	12	7	1.71	1	0	0	2	0	0

I then calculated the fraction of all data points that have a 'Class' label of 1; approved and fractional percentage of fraudulent data points/all points.

```
Approval percentage =  0.5550724637681159
Total # of approved pts:  383.0
Out of (total) pts:  690
```

Implementation:

The applied workflow of my model is displayed below.



1. Import the necessary Libraries:

   Firstly, I imported all the necessary libraries required for the implementation of my project. It includes libraries like io, os, matplotlib, numpy, pandas, warnings, seaborn, sagemaker, boto3, etc.

2. Loading and Exploring Data:

   It's important to look at the distribution of data since this will inform how we develop a credit card approval prediction model. We'll want to know: How many data points we have to work with, the number and type of features, and finally, the distribution of data over the classes (approved or not approved).

   We loaded our csv file and explored its attributes and values. A description of our initial dataframe is displayed below.

```
credit.describe()
```

	Age	Debt	YearsEmployed	CreditScore	ZipCode	Income
count	678.000000	690.000000	690.000000	690.00000	677.000000	690.000000
mean	31.568171	4.758725	2.223406	2.40000	184.014771	1017.385507
std	11.957862	4.978163	3.346513	4.86294	173.806768	5210.102598
min	13.750000	0.000000	0.000000	0.00000	0.000000	0.000000
25%	22.602500	1.000000	0.165000	0.00000	75.000000	0.000000
50%	28.460000	2.750000	1.000000	0.00000	160.000000	5.000000
75%	38.230000	7.207500	2.625000	3.00000	276.000000	395.500000
max	80.250000	28.000000	28.500000	67.00000	2000.000000	100000.000000

```
print('Data shape (rows, cols): ', credit.shape)
print()
```

```
Data shape (rows, cols):  (690, 16)
```

3. Data Pre-Processing

   This step is discussed in detail in the previous pages. A short summary is as follows:

   a. Convert the non-numeric data into numeric.

   b. Dealing with Missing Values.

   c. Dropping unnecessary columns.

   d. Label Encoding.

   e. Split the data into train and test sets.

   f. Scale the feature values to a uniform range.

4. Data Visualization

   This step has also been discussed in great details in the previous sections. We displayed the following plots to look for patterns and correlation in our data:

   a. Correlation HeatMap between columns of our dataframe.

   b. PairPlot between different columns of our dataframe.

   c. DistPlot or Distribution Plot of various attributes like Age, Debt, PriorDefault, CreditScore, YearsEmployed and Income of our dataframe.

5. Splitting the Dataset into Training and Test Set:

   I've split my data into train set and test set to prepare our data for two different phases of machine learning modeling: training and testing, using the test_train_split function from the sklearn library, with a test_size of 0.2.

6. Uploading the data to S3:

   I've uploaded locally-stored train.csv and test.csv file to S3 by using sagemaker_session.upload_data. This function needs to know: where the data is saved locally, and where to upload in S3 (a bucket and prefix).

7. Modelling:

   Firstly, I created a default LinearLearner estimator, which is displayed below. I trained my input data (after converting into RecordSet format, as required by the

model) on this default model, along with 30 epochs.

```
linear = LinearLearner(role=role,
                       train_instance_count=1,
                       train_instance_type='ml.c4.xlarge',
                       predictor_type='binary_classifier',
                       output_path=output_path,
                       sagemaker_session=sagemaker_session,
                       epochs=30)
```

8. Deploy the Trained Model:

   I've deployed my model to create a predictor. I'll use this to make predictions on our test data and evaluate the model.

9. Evaluate our Model:

   On evaluating our model using the confusion matrix metric, I got an accuracy of approx. 82% which is a good number but can be improved. The result of this model is also displayed below. The default LinearLearner got a high accuracy, but still classified some data points incorrectly.

   This is a snapshot of predicting the output of only one set of input data.

```
# test one prediction
test_x_np = X_test.astype('float32')
result = linear_predictor.predict(test_x_np[0])

print(result)
```

```
[label {
  key: "predicted_label"
  value {
    float32_tensor {
      values: 1.0
    }
  }
}
label {
  key: "score"
  value {
    float32_tensor {
      values: 0.9079181551933289
    }
  }
}
]
```

   The actual metric of our first model is shown below.

```
Metrics for simple, LinearLearner.


Recall:     0.853
Precision:  0.795
Accuracy:   0.819
```

10. Delete the Endpoint:

    Deleting the endpoint is the most important step if you don't want to be charged by Amazon!

11. Improving the Model:

    This section is described in detail in the next section. I applied the process of Model Optimization to increase the accuracy of our model.

12. Final Clean Up:

    When I am completely done with training and testing models, I have deleted my entire S3 bucket. I also ensure that I've deleted all my endpoints thoroughly. I also manually check about this in the AWS SageMaker console. I ensure that I've deleted all training jobs, endpoints, endpoint configurations, models, data uploaded on S3, etc.

Refinement:

If we imagine that we are designing this model for use in a bank application, we know that users do not want any valid credit card applications to be categorized as not approved. That is, we want to have as few false positives (0s classified as 1s) as possible.

On the other hand, if our bank manager asks for an application that will catch almost all cases of not approved, even if it means a higher number of false positives, then we'd want as few false negatives as possible.

To train according to specific product demands and goals, we do not want to optimize for accuracy only. Instead, we want to optimize for a metric that can help us decrease the number of false positives or negatives.

Optimizing according to a specific metric is called model tuning, and SageMaker provides a number of ways to automatically tune a model [15].

To aim for a specific metric, LinearLearner offers the hyperparameter binary_classifier_model_selection_criteria, which is the model evaluation criteria for the training dataset.

Therefore, firstly I tuned the model so to get a higher precision. I have displayed the second LinearLearner's estimator below.

```
# instantiate a LinearLearner
# tune the model for a higher precision
linear_precision = LinearLearner(role=role,
                                 train_instance_count=1,
                                 train_instance_type='ml.c4.xlarge',
                                 predictor_type='binary_classifier',
                                 output_path=output_path,
                                 sagemaker_session=sagemaker_session,
                                 epochs=15,
                                 binary_classifier_model_selection_criteria='recall_at_target_precision', # target precision
                                 target_precision=0.95) # 95% precision
```

The result we got after training our input data according to this estimator was even worse. The exact specifications are displayed below.

```
Metrics for first LinearLearner.


Recall:     0.632
Precision:  0.860
Accuracy:   0.768
```

Aiming for a higher precision led to a further decrease in the accuracy of our model. Hence, next we tried to target a different metric. Therefore, finally, I tuned the model so to get a higher recall. I have displayed the final LinearLearner's estimator below.

```
# instantiate a LinearLearner
# tune the model for a higher recall
linear_recall = LinearLearner(role=role,
                              train_instance_count=1,
                              train_instance_type='ml.c4.xlarge',
                              predictor_type='binary_classifier',
                              output_path=output_path,
                              sagemaker_session=sagemaker_session,
                              epochs=80,
                              binary_classifier_model_selection_criteria='precision_at_target_recall', # target recall
                              target_recall=0.9) # 90% recall
```

Finally, I achieved a model with a good accuracy, precision and recall, higher than those of our benchmark model. Hence, this is my final model that I trained, deployed and evaluated for my project. My final model has an accuracy of approx. 85%, recall of 90% and precision of 81.3%.

```
Metrics for final LinearLearner Model.


Recall:     0.897
Precision:  0.813
Accuracy:   0.848
```

- # Results:

Model Evaluation and Validation:

We generated a final model with above list of tuned parameters. The output of this tuned model came just about 9% higher in accuracy vs the benchmarked model. We used a helper function to evaluate the Recall, Precision and Accuracy of our model. The exact specifications of my model are displayed above. Code snippet of the helper function is displayed below.

```
# instantiate a LinearLearner
# tune the model for a higher recall
linear_recall = LinearLearner(role=role,
                              train_instance_count=1,
                              train_instance_type='ml.c4.xlarge',
                              predictor_type='binary_classifier',
                              output_path=output_path,
                              sagemaker_session=sagemaker_session,
                              epochs=80,
                              binary_classifier_model_selection_criteria='precision_at_target_recall', # target recall
                              target_recall=0.9) # 90% recall
```

```python
def evaluate1(predictor, X_test, y_test):
    """
    Evaluate a model on a test set given the prediction endpoint.
    Return binary classification metrics.
    :param predictor: A prediction endpoint
    :param test_features: Test features
    :param test_labels: Class labels for test data
    :param verbose: If True, prints a table of all performance metrics
    :return: A dictionary of performance metrics.
    """
    # We have a lot of test data, so we'll split it into batches of 100
    # split the test data set into batches and evaluate using prediction endpoint
    prediction_batches = [predictor.predict(batch) for batch in np.array_split(X_test, 50)]

    # LinearLearner produces a `predicted_label` for each data point in a batch
    # get the 'predicted_label' for every point in a batch
    test_preds = np.concatenate([np.array([x.label['predicted_label'].float32_tensor.values[0] for x in batch])
                                 for batch in prediction_batches])

    # calculate true positives, false positives, true negatives, false negatives
    tp = np.logical_and(y_test, test_preds).sum()
    fp = np.logical_and(1-y_test, test_preds).sum()
    tn = np.logical_and(1-y_test, 1-test_preds).sum()
    fn = np.logical_and(y_test, 1-test_preds).sum()

    # calculate binary classification metrics
    recall = tp / (tp + fn)
    precision = tp / (tp + fp)
    accuracy = (tp + tn) / (tp + fp + tn + fn)

    print(pd.crosstab(X_test, test_preds, rownames=['actual (row)'], colnames=['prediction (col)']))
    print("\n{:<11} {:.3f}".format('Recall:', recall))
    print("{:<11} {:.3f}".format('Precision:', precision))
    print("{:<11} {:.3f}".format('Accuracy:', accuracy))
    print()

    return {'TP': tp, 'FP': fp, 'FN': fn, 'TN': tn,
            'Precision': precision, 'Recall': recall, 'Accuracy': accuracy}
```

Justification:

There was room for improvement on my previously defined model. After applying Model Optimization and tuning the model to achieve a better specific metric, I finally achieved a model with a good accuracy, recall and precision. I used the LinearLearner model because it is something, I am comfortable with working in and is highly suitable for my project's binary classification problem. There could be more ways we could improve the score, particularly by selecting a subset of features using the ranking obtained from observing feature importance ranking and then performing the same exercise we did as describe above. But this will be out of scope of this report for now.

- ## References:

1. https://medium.com/datadriveninvestor/predicting-credit-card-approvals-using-ml-techniques-9cd8eaeb5b8c
2. http://www.ijrar.org/papers/IJRAR190B030.pdf
3. https://escholarship.org/uc/item/9zg7157q
4. http://rstudio-pubs-static.s3.amazonaws.com/73039_9946de135c0a49daa7a0a9eda4a67a72.html
5. https://medium.com/@kennymiyasato/binary-classification-project-using-decision-tree-with-kaggle-dataset-3123398a1c70#:~:text=Decision%20Tree%20Supervised%20Learning&text=Tree%20models%20where%20the%20target,lead%20to%20those%20class%20labels.
6. https://christophm.github.io/interpretable-ml-book/simple.html
7. https://towardsdatascience.com/interpretability-in-machine-learning-70c30694a05f
8. https://www.kaggle.com/redwuie/credit-card-approval?select=train.csv

9. https://medium.com/datadriveninvestor/predicting-credit-card-approvals-using-ml-techniques-9cd8eaeb5b8c
10. https://medium.com/hugo-ferreiras-blog/confusion-matrix-and-other-metrics-in-machine-learning-894688cb1c0a
11. https://www.kaggle.com/noureddineizmar/predicting-credit-card-approvals
12. https://www.journaldev.com/39993/seaborn-distplot
13. https://towardsdatascience.com/better-heatmaps-and-correlation-matrix-plots-in-python-41445d0f2bec
14. https://www.geeksforgeeks.org/python-seaborn-pairplot-method/
15. https://docs.aws.amazon.com/sagemaker/latest/dg/linear-learner.html