# Project 4: Machine Learning

In this project, we are going to utilize python to develop a predictive machine learning model. The data that is used in this project originally comes from the UCI machine learning repository ([link](https://archive.ics.uci.edu/ml/datasets/Bank+Marketing)). The data is related with over 40,000 direct marketing campaigns of a Portuguese banking institution from May 2008 to November 2010. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be (‘yes’) or not (‘no’) subscribed.

# Data Exploration

I begin by loading the data and exploring the columns.

Looking briefly at the data columns, we can see that there are various numerical and categorical columns

The most important column here is y, which is the output variable (desired target): this will tell us if the client subscribed to a term deposit(binary: ‘yes’,’no’).

Then I defined an output variable to use for our binary classification.

Next I defined a function in order to calculate the prevalence of population that subscribes to a term deposit using calc\_prevalence.

Here we see that around 11% of the population has a term deposit. This is known as an imbalanced classification problem so we will address that below.

From digging deeply to analyze the columns, we see there are a mix of categorical (non-numeric) and numerical data. A few things to note —

* All the data inputted are non-null values, meaning that we have a value for every column
* age, duration, campaign, pdays, previous, emp.var.rate, cons.price.idx, cons.conf.idx, euribor3m and nr.employed are numerical variables
* default, housing and loan have 3 values each (yes, no and unknown)
* Output (y) has two values: “yes” and “no”
* We are discarding duration. This attribute highly affects the output target (e.g., if duration=0 then y=’no’). Yet, the duration is not known before a call is performed. Also, after the end of the call yis obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model

**Feature Engineering**

In this section, I am going to create features for the machine learning model. In each section, I added new variables to the dataframe and keep track of which columns of the dataframe that are going to engage as part of the features for the predictive model.

**Numerical Features**

Next, I will divide this section into numerical and categorical features and check if there are any missing values in the numerical data. Then graphed out each column

**Categorical Features**

Categorical variables are non-numeric data such as job and education. To turn these non-numerical data into variables I used One-Hot encoding.

**One-Hot Encoding**

In one-hot encoding, I created a new column for each unique value in that column. Now, the value of the column is 1 if the sample has that unique value or else 0 . For example, for the column *job*, we would create new columns (“job\_blue-collar”, “job\_entrepreneur”, etc). If the client’s job is blue-collar, the client gets a 1 under ‘job\_blue-collar’ and 0 under the rest of the job columns. To create these one-hot encoding columns, I utilized the get\_dummies function provided by pandas.

A problem that arises is by creating a column for each unique value, we have correlated columns. That is to say, the value in one column can be figured out by looking at the rest of the columns. For example, if *marital*is not “married”, “single”, or “divorced”, it must be “unknown”. In order to fix this, I used the drop\_first option, which will drop the first categorical value for each column. Now we are ready to make all of our categorical features.

In order to add the one-hot encoding columns to the dataframe, I use the concatfunction. axis = 1 is used to add the columns.

I then saved the column names of the categorical data to keep track of them.

**Feature Engineering: Summary**

Through this process I created 62 features for the machine learning model

* 9 numerical features
* 53 categorical features

Next I created a new dataframe that only has the features and the OUTPUT\_LABEL

**Building Training/Validation/Test Samples**

Now, It is now time to split the data. The reason why we split the data is so that you can measure how well your model would do on unseen data and I split into three parts:

* Training samples: these samples are used to train the model
* Validation samples: these samples are held out from the training data and are used to make decisions on how to improve the model
* Test samples: these samples are held out from all decisions and are used to test(measure) the generalized performance of the model

In this project, I split into 70% train, 15% validation, and 15% test, then shuffle the samples using sample in case there was some order Here n is the number of samples. random\_state is just specified so the project is reproducible.

I used sample again to extract 30% (using frac) of the data to be used for validation and test splits.

And now I can split into test and validation using 50% fraction.

The .drop function just drops the rows from df\_test to get the rows that were not part of the sample. I used this same idea to get the training data.

Lastly, I checked what percent of our groups are likely to subscribe to a term deposit. Now I can see that the prevalence is about the same for each group.

At this point, we might suggest dropping the training data into a predictive model and see the outcome. However, because recall= 0%, we can see that we never caught any of the clients that will subscribe to a term deposit. What is happening is that we have an imbalanced dataset where there are much more negatives than positives, therefore the model might just assign all samples as negative.

**Balancing the Data (Sub-Sampling)**

Here, I create a balanced training, validation and test data set that has 50% positive and 50% negative. Then create the x and y matrices.

There can be troubles in machine learning models when the variables are of different size (0–100, vs 0–1000000). To combat this, I scaled the data.

Here we will use scikit-learn’s Standard Scaler, which removes the mean and scales to unit variance and create a scaler using all the training data.

**Model Selection**

In this section I train a few machine learning models and use a few techniques for optimizing them and then select the best model based on performance on the validation set.

Since we now have a balanced training data, I set the threshold at 0.5 to label a predicted sample as positive.

**Model Selection: Baseline models**

In this section, I first compared the model performance of the following 7 machine learning models using default hyperparameters:

* K-Nearest Neighbors
* Logistic Regression
* Stochastic Gradient Descent
* Naive Bayes
* Decision Tree
* Random Forest
* Gradient Boosting Classifier

**K Nearest Neighbors (KNN)**

KNN is one the simplest machine learning models. KNN looks at the k closest datapoints and probability sample that has positive labels. This model is very easy to understand, versatile, and you don’t need an assumption for the data structure. KNN is also good for multivariate analysis. A caveat with this algorithm is being sensitivity to K and takes a long time to evaluate if the number of trained samples is large.

**Logistic Regression**

Logistic regression is a traditional machine learning model that fits a linear decision boundary between the positive and negative samples. Logistic regression uses a line (Sigmoid function) in the form of an “S” to predict if the dependent variable is true or false based on the independent variables. One advantage of logistic regression is the model is interpretable — we know which features are important for predicting positive or negative. Take note that the modeling is sensitive to the scaling of the features, so that is why we scaled the features above.

**Stochastic Gradient Descent**

Stochastic gradient descent is similar to logistic regression. Stochastic Gradient Descent analyzes various sections of the data instead of the data as a whole and predicts the output using the independent variables. Stochastic Gradient Descent is faster than logistic regression in the sense that it doesn’t run the whole dataset but instead looks at different parts of the dataset.

**Naive Bayes**

Naive Bayes is a model traditionally used in machine learning. This algorithm uses Bayes rule which calculated the probability of an event related to previous knowledge of the variables concerning the event. The “Naive” part is that the model assumes that all variables in the dataset are independent of each other — meaning there are no dependent variables or output.

**Decision Tree**

Another class of popular machine learning models is tree-based methods. The simplest tree-based method is known as a decision tree. The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by *learning simple decision rules* gotten from training data. In Decision Trees, for predicting a class label for a record we start from the **root** of the tree. One advantage of tree-based methods is that they have no assumptions about the structure of the data and can pick up non-linear effects if given sufficient tree depth.

**Random forest**

One disadvantage of decision trees is that they tend overfit very easily by memorizing the training data. Overfitting occurs when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. Random forests were created to reduce the overfitting. In random forest models, multiple trees are created, and the results are aggregated. The trees in a forest are decorrelated by using a random set of samples and random number of features in each tree. In most cases, random forests work better than decision trees because they can generalize more easily.

**Gradient Boosting Classifier**

Boosting is a technique that builds a new decision tree algorithm that focuses on the errors on the dataset to fix them. This learns the whole model in other to fix it and improve the prediction of the model. A model that uses this technique combined with a gradient descent algorithm (controlling learning rate) is known as gradient boosting classifier.

**Analysis of Baseline Models**

The next step is to make a dataframe with the results of all the baseline models and plot the outcomes using a package called seaborn. I utilized the AUC to evaluate the best model. This is a good data science performance metric for picking the best model since it captures the tradeoff between the true positive and false positive and does not require selecting a threshold.

As we can see most of the models (except decision tree) have similar performance on the validation set. There is some overfitting as noted by the drop between training and validation. Let’s check if we can improve this performance using a few more techniques.

**Model Selection: Learning Curve**

In this section, I diagnosed how the models are doing by plotting a learning curve. I will make use of the learning curve code from scikit-learn’s website with a small change of plotting the AUC instead of accuracy.

In the case of random forest, we can see the model has high variance because the training and cross-validation scores show data points which are very spread out from one another. High variance would cause an algorithm to model the noise in the training set (overfitting).

Depending on the learning curve, there are a few strategies we can employ to improve the models

High Variance:  
- Reduce number of features  
- Decrease model complexity  
- Add regularization  
- Add more samples

High Bias:  
- Add new features  
- Increase model complexity  
- Reduce regularization  
- Change model architecture

**Model Selection: Feature Importance**

A way of improving your models to understand what features are important to your models. This can usually only be investigated for simpler models such as Logistic Regression or Random Forests.

We can take a look at the top 50 positive and top 50 negative coefficients to get some insight.

After reviewing these charts, I realized the features that have more impact on the predictive outcomes of the model are cons.price.idx, and euribor3m due to their high importance score. cons.price.idx is the consumer price index which measures changes in the price level of a weighted average market basket of consumer goods and services purchased by households. A lower the price index will encourage clients to subscribe to a term deposit. Similarly, euribor3m is the Euribor (Euro InterBank Offered Rate) which is the average interest rate banks provide on short term loans (3 months). This is a metric that shows clients’ ability to pay off short terms loans.

**Model Selection: Hyperparameter Tuning**

Hyperparameter Tuning is the process of searching for the ideal model architecture. These are parameters which define the model architecture. We are only going to optimize the hyper parameters for stochastic gradient descent, random forest, and gradient boosting classifier. We will not optimize KNN since it took a while to train. We will not optimize logistic regression since it performs similarly to stochastic gradient descent. Similarly, we will not optimize decision trees since they tend to overfit and perform worse that random forests and gradient boosting classifiers.

A good tool for hyperparameter tuning is Grid search — where grid values are tested using all possible combinations. This is a computationally intensive method. Another option is to randomly test a permutation of them. This technique is called Random Search and is also deployed in scikit-learn.

Now, we can create a grid over the random forest hyperparameters.

To implement the RandomizedSearchCV function, we need something to score or evaluate a set of hyperparameters. Here we will use the AUC.

The three important parameters of RandomizedSearchCV are

* scoring = evaluation metric used to pick the best model
* n\_iter = number of different combinations
* cv = number of cross-validation splits

# Model Selection: Best Classifier

Looking at the results, we can see that the hyperparameter tuning improved the models, but not by much. This is most likely due to the fact that we have a high variance situation. In this phase, we will choose the gradient boosting classifier since it has the best AUC on the validation set. I don’t want to train your best classifier every time I want to run new predictions. Therefore, we need to save the classifier. We will use the package pickle.

**Model Evaluation**

Now that we have chosen our best model (optimized gradient boosting classifier). Let’s evaluate the performance of the test set.

In this case, we will choose the gradient boosting classifier since it has the best AUC on the validation set.

Additionally, we can create the ROC curve for the 3 datasets

**Conclusion**

Through this project, we created a machine learning model that is able to predict how likely clients will subscribe to a bank term deposit. The best model was **gradient boosting classifier** with optimized hyperparameters. Our model’s test performance (AUC) is 79.5%. A precision of 0.82 divided by a prevalence of 0.50 gives us 1.6, which means the model helps us 1.6 times better than randomly guessing. The model was able to catch 62% of customers that will subscribe to a term deposit. We should focus on targeting customers with high cons.price.idx (consumer price index) and euribor3m (3 month indicator for paying off loans) as they are high importance features for the model and business. Therefore, we save time and money knowing the characteristics of clients we should market to and that will lead to increased growth and revenue.