Using Machine Learning to Predict Subscription to Bank Term Deposits for Clients with Python

Bank Marketing with Machine Learning using Scikit-Learn

Emeka Efidi Follow
May 15, 2020 · 18 min read ★

“No great marketing decisions have ever been made on qualitative data.” — John Sculley (CEO of Apple Inc.)

Credit: wpforms.com/best-google-analytics-plugins-for-wordpress/

Introduction

Marketing to potential clients has always been a crucial challenge in attaining success for banking institutions. It's not a surprise that banks usually deploy mediums such as social media, customer service, digital media and strategic partnerships to reach out to customers. But how can banks market to a specific location, demographic, and society with increased accuracy? With the inception of machine learning - reaching out to specific groups of people have been revolutionized by using data and analytics to provide detailed strategies to inform banks which customers are more likely to subscribe to a financial product. In this project on bank marketing with machine learning, I will explain how a particular Portuguese bank can use predictive analytics to help prioritize customers which would subscribe to a bank term deposit.

In this project I will demonstrate how to build a model predicting clients subscribing to a term deposit using the following steps -

- Project definition
- Data exploration
- Feature engineering
- Building training/validation/test samples
- Model selection
- Model evaluation

You can see my code in the Jupyter Notebook provided on my GitHub (https://github.com/emekaefidi/Bank-Marketing-with-Machine-Learning).

This project was inspired by Andrew Long!(check him out- https://towardsdatascience.com/@awlong20).

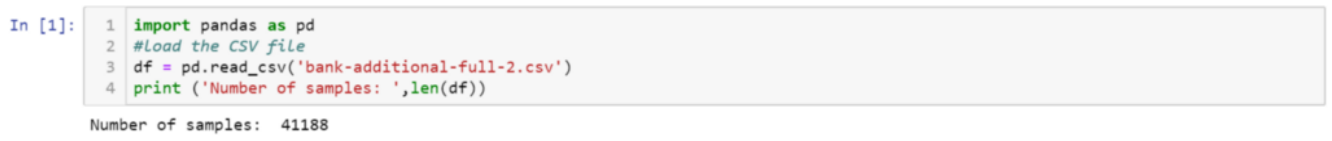
Project Definition

Predict if a client will subscribe (yes/no) to a term deposit — this is defined as a classification problem.

Data Exploration

The data that is used in this project originally comes from the UCI machine learning repository (link). 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.

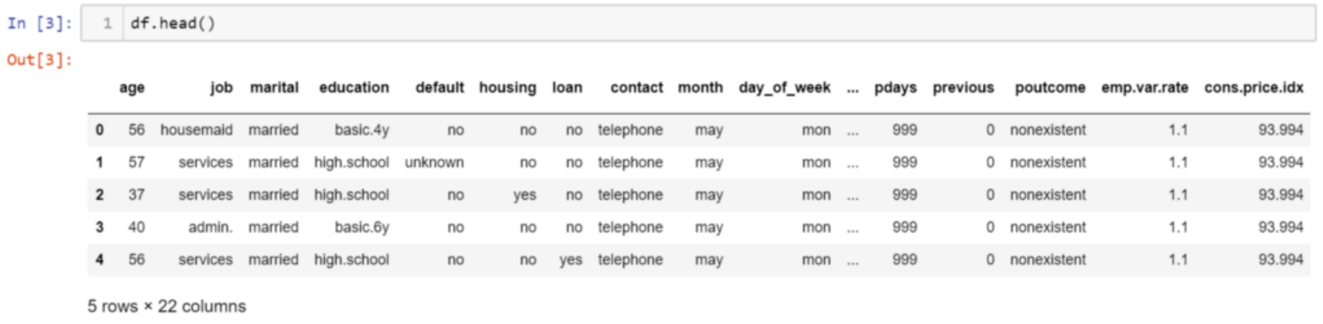
In this project, we are going to utilize python to develop a predictive machine learning model! Let's begin by loading our data and exploring the columns.



```
In [1]:
import pandas as pd
#Load the CSV file
df = pd.read_csv('bank-additional-full-2.csv')
print ('Number of samples: ',len(df))
```

```
Number of samples:  41188
```

Looking briefly at the data columns, we are can see that there are various numerical and categorical columns! These columns can be explained in more details below:



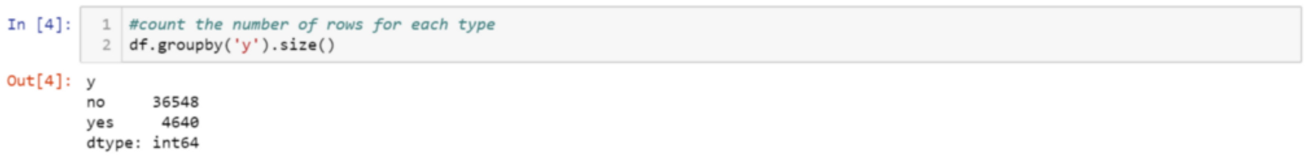
In [3]:
```
df.head()
```

Out[3]:

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	...	pdays	previous	poutcome	emp.var.rate	cons.price.idx
0	56	housemaid	married	basic.4y	no	no	no	telephone	may	mon	...	999	0	nonexistent	1.1	93.994
1	57	services	married	high.school	unknown	no	no	telephone	may	mon	...	999	0	nonexistent	1.1	93.994
2	37	services	married	high.school	no	yes	no	telephone	may	mon	...	999	0	nonexistent	1.1	93.994
3	40	admin.	married	basic.6y	no	no	no	telephone	may	mon	...	999	0	nonexistent	1.1	93.994
4	56	services	married	high.school	no	no	yes	telephone	may	mon	...	999	0	nonexistent	1.1	93.994

5 rows × 22 columns

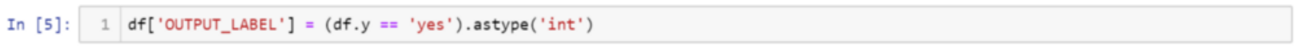
The most important column here is `y`, which is the output variable (desired target): this will tell us if the client subscribed to a term deposit(binary: 'yes','no').



In [4]:
```
#count the number of rows for each type
df.groupby('y').size()
```

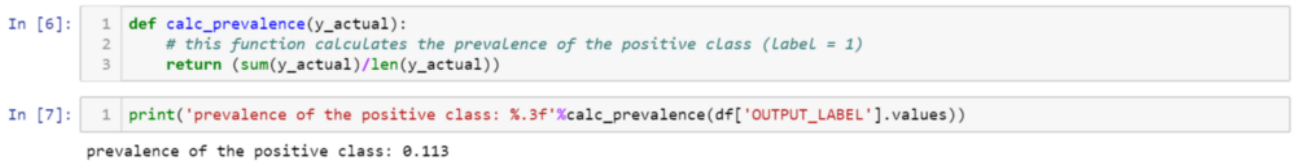
Out[4]:
```
y
no     36548
yes     4640
dtype: int64
```

Now let's define an output variable to use for our binary classification. We will try to predict if a client is likely to subscribe to a term deposit.



```
In [5]: df['OUTPUT_LABEL'] = (df.y == 'yes').astype('int')
```

Let's define a function in order to calculate the prevalence of population that subscribes to a term deposit.



```
In [6]: def calc_prevalence(y_actual):
            # this function calculates the prevalence of the positive class (label = 1)
            return (sum(y_actual)/len(y_actual))

In [7]: print('prevalence of the positive class: %.3f'%calc_prevalence(df['OUTPUT_LABEL'].values))

prevalence of the positive class: 0.113
```

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

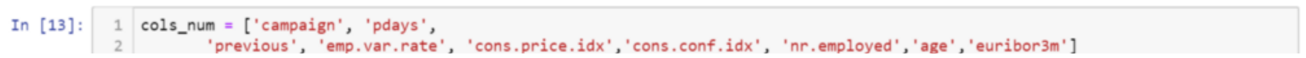
- default, housing and loan have 3 values each (yes, no and unknown)
- Output (y) has two values: “yes” and “no”
- We are discarding duration. This attribute highly affects the output target (e.g., if duration=0 then y=’no’). Yet, the duration is not known before a call is performed. Also, after the end of the call `y` is 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, We are going to create features for our machine learning model. In each section, we will add new variables to the dataframe and keep track of which columns of the dataframe we are going to engage as part of the features for the predictive model. We will divide this section into numerical and categorical features.

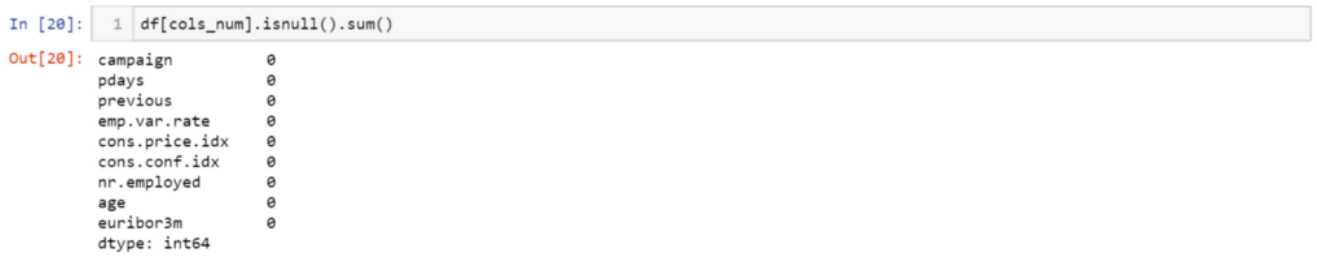
Numerical Features

These are numeric data. The numerical columns that we will use can be seen below:



```
In [13]: cols_num = ['campaign', 'pdays',
                'previous', 'emp.var.rate', 'cons.price.idx','cons.conf.idx', 'nr.employed','age','euribor3m']
```

Now, let's check if there are any missing values in the numerical data.

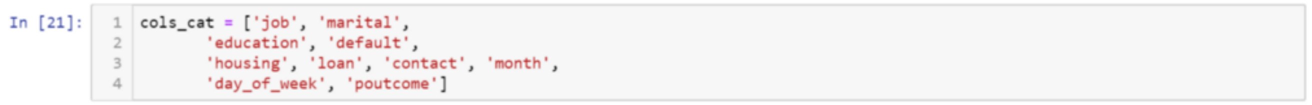


```
In [20]: df[cols_num].isnull().sum()
Out[20]: campaign          0
         pdays             0
         previous          0
         emp.var.rate      0
         cons.price.idx    0
         cons.conf.idx     0
         nr.employed       0
         age               0
         euribor3m         0
         dtype: int64
```

Categorical Features

Categorical variables are non-numeric data such as job and education. To turn these non-numerical data into variables, the simplest thing is to use a technique called ***one-hot encoding***, which will be explained below.

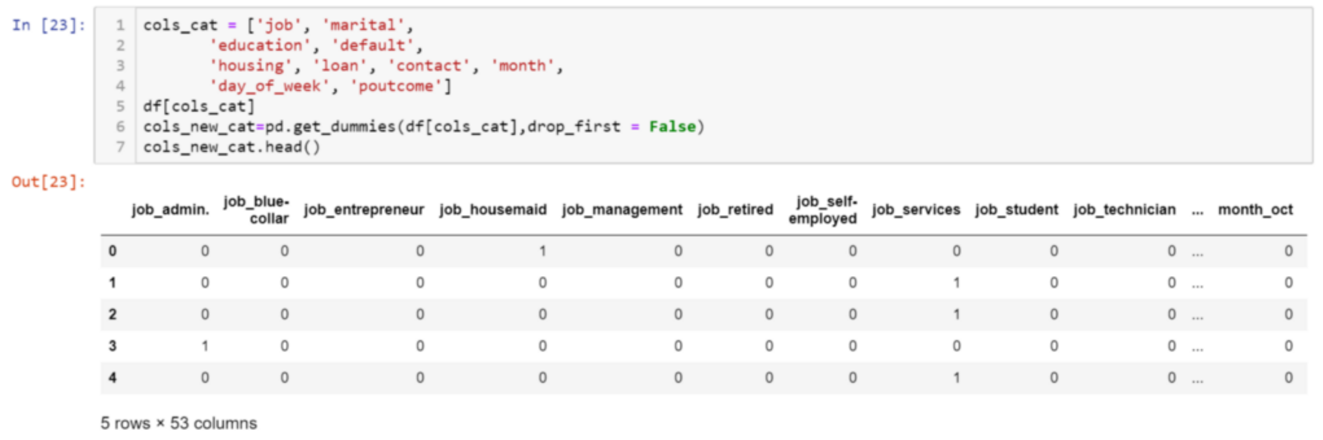
The first set of categorical data we will work on are these columns:



```
In [21]: cols_cat = ['job', 'marital',
                'education', 'default',
                'housing', 'loan', 'contact', 'month',
                'day_of_week', 'poutcome']
```

In one-hot encoding, we will create 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, we will utilize 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, we can use 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 [23]:

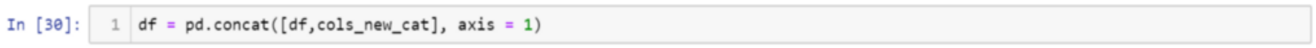
```
cols_cat = ['job', 'marital',
        'education', 'default',
        'housing', 'loan', 'contact', 'month',
        'day_of_week', 'poutcome']
df[cols_cat]
cols_new_cat=pd.get_dummies(df[cols_cat],drop_first = False)
cols_new_cat.head()
```

Out[23]:

	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_management	job_retired	job_self-employed	job_services	job_student	job_technician	...	month_oct
0	0	0	0	1	0	0	0	0	0	0	...	0
1	0	0	0	0	0	0	0	1	0	0	...	0
2	0	0	0	0	0	0	0	1	0	0	...	0
3	1	0	0	0	0	0	0	0	0	0	...	0
4	0	0	0	0	0	0	0	1	0	0	...	0

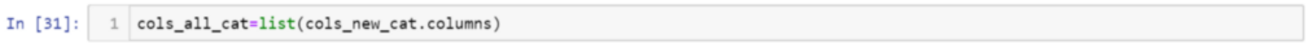
5 rows × 53 columns

In order to add the one-hot encoding columns to the dataframe, we use the `concat` function. axis = 1 is used to add the columns.



```
In [30]: df = pd.concat([df,cols_new_cat], axis = 1)
```

Let’s now save the column names of the categorical data to keep track of them.



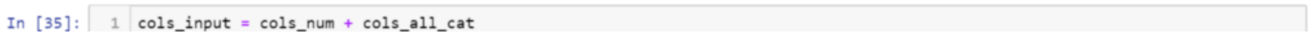
```
In [31]: cols_all_cat=list(cols_new_cat.columns)
```

Feature Engineering: Summary

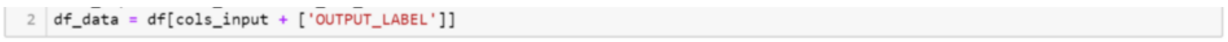
Through this process we created 62 features for the machine learning model! We separated the features to the following:

- 9 numerical features
- 53 categorical features

We will create a new dataframe that only has the features and the `OUTPUT_LABEL`



```
In [35]: cols_input = cols_num + cols_all_cat
```



```
df_data = df[cols_input + ['OUTPUT_LABEL']]
```

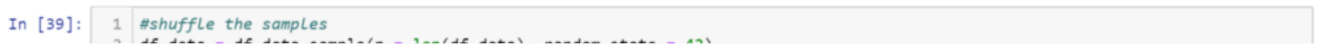
Building Training/Validation/Test Samples

Till this very point we have explored our data and created features from the categorical data. Now, It is now time for us to split our data. The reason why we split the data is so that you can measure how well your model would do on unseen data. We split into three parts:

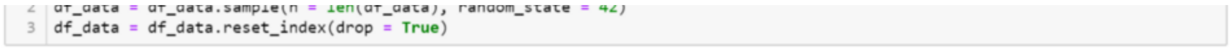
- Training samples: these samples are used to train the model
- Validation samples: these samples are held out from the training data and are used to make decisions on how to improve the model
- Test samples: these samples are held out from all decisions and are used to **test** (measure) the generalized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

In this project, we will split into 70% train, 15% validation, and 15% test!

Let's shuffle the samples using `sample` in case there was some order (e.g. all positive samples on top). Here `n` is the number of samples. `random_state` is just specified so the project is reproducible.

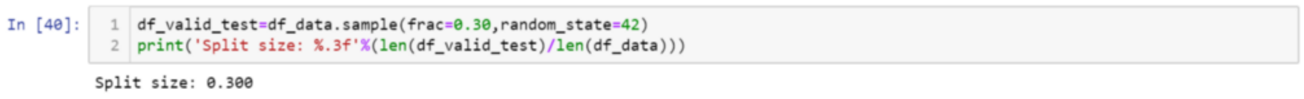


```
In [39]: #shuffle the samples
```



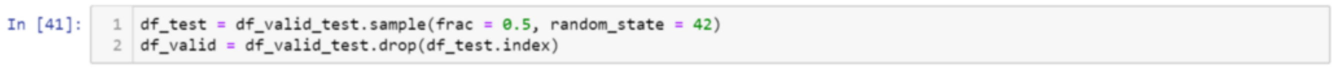
```
df_data = df_data.sample(n = len(df_data), random_state = 42)
df_data = df_data.reset_index(drop = True)
```

We can use `sample` again to extract 30% (using `frac`) of the data to be used for validation and test splits. An important note is that the validation and test come from similar distributions and this technique is one way to do it.



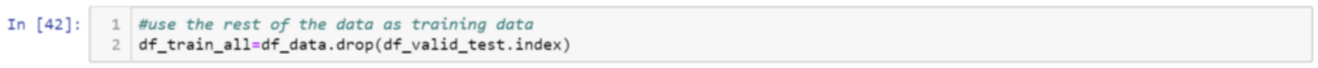
```
In [40]: df_valid_test=df_data.sample(frac=0.30,random_state=42)
         print('Split size: %.3f'%(len(df_valid_test)/len(df_data)))

Split size: 0.300
```

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



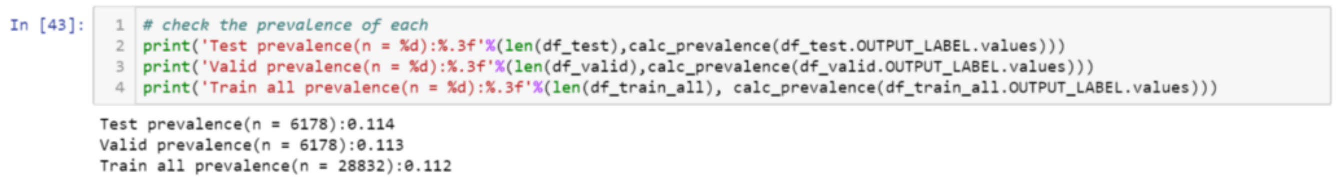
```
In [41]: df_test = df_valid_test.sample(frac = 0.5, random_state = 42)
         df_valid = df_valid_test.drop(df_test.index)
```

The `.drop` function just drops the rows from `df_test` to get the rows that were not part of the sample. We can use this same idea to get the training data.



```
In [42]: #use the rest of the data as training data
         df_train_all=df_data.drop(df_valid_test.index)
```

At this junction, let’s check what percent of our groups are likely to subscribe to a term deposit. This is known as prevalence. Ideally, all three groups would have similar prevalence.



```
In [43]:
# check the prevalence of each
print('Test prevalence(n = %d):%.3f'%(len(df_test),calc_prevalence(df_test.OUTPUT_LABEL.values)))
print('Valid prevalence(n = %d):%.3f'%(len(df_valid),calc_prevalence(df_valid.OUTPUT_LABEL.values)))
print('Train all prevalence(n = %d):%.3f'%(len(df_train_all), calc_prevalence(df_train_all.OUTPUT_LABEL.values)))

Test prevalence(n = 6178):0.114
Valid prevalence(n = 6178):0.113
Train all prevalence(n = 28832):0.112
```

Now we can see that the prevalence is about the same for each group.

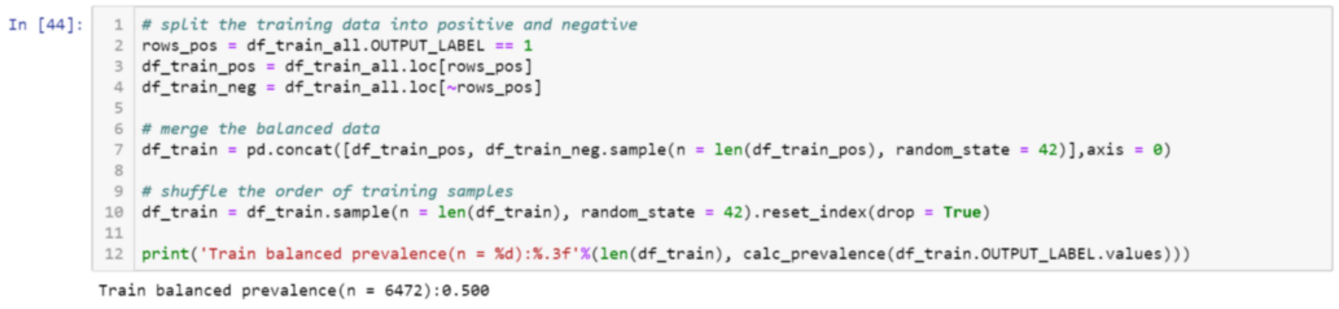
At this point, we might suggest dropping the training data into a predictive model and see the outcome. However, if we do this, there’s a chance that we will get back a model that is 89% accurate. But wait, we never caught any of the clients that will subscribe to a term deposit(recall= 0%). How can this be possible?

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.

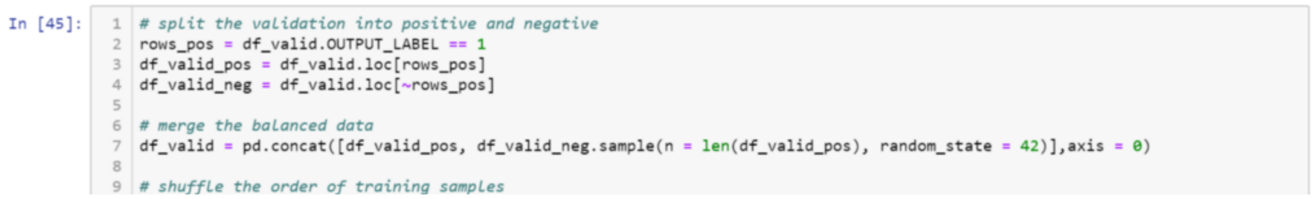
Typically, it is best practice to balance the data in some way to give the positives more weight. There are 3 techniques that are typically utilized:

- sub-sample the more dominant class: using random subset of the negatives
- over-sample the imbalanced class: using the same positive samples multiple times
- create synthetic positive data

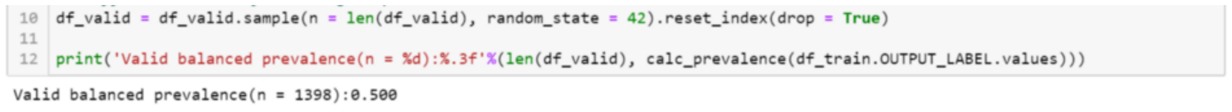
Usually, you will want to use the latter two methods if you only have a handful of positive cases. Since we have a few thousand positive cases, let's use the **sub-sample approach**. Here, we will create a balanced training, validatoin and test data set that has 50% positive and 50% negative. You can also try tweaking this ratio to see if you can get an improvement.



```
In [44]:
# split the training data into positive and negative
rows_pos = df_train_all.OUTPUT_LABEL == 1
df_train_pos = df_train_all.loc[rows_pos]
df_train_neg = df_train_all.loc[~rows_pos]

# merge the balanced data
df_train = pd.concat([df_train_pos, df_train_neg.sample(n = len(df_train_pos), random_state = 42)],axis = 0)

# shuffle the order of training samples
df_train = df_train.sample(n = len(df_train), random_state = 42).reset_index(drop = True)

print('Train balanced prevalence(n = %d):%.3f'%(len(df_train), calc_prevalence(df_train.OUTPUT_LABEL.values)))

Train balanced prevalence(n = 6472):0.500
```

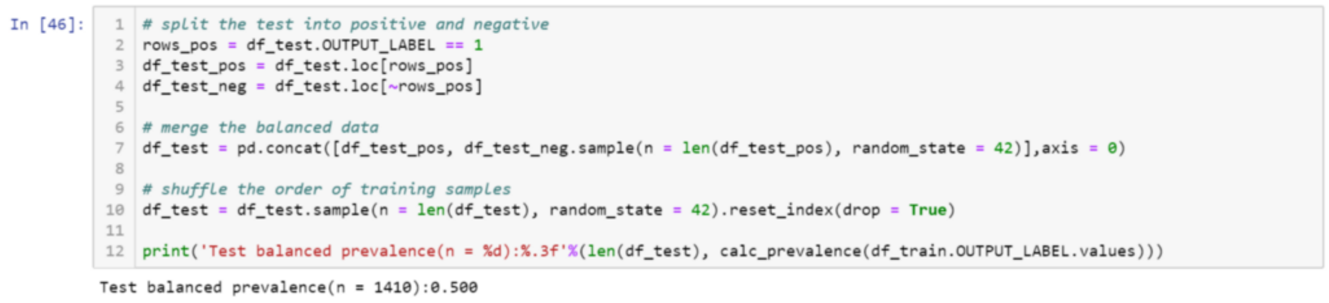


```
In [45]:
# split the validation into positive and negative
rows_pos = df_valid.OUTPUT_LABEL == 1
df_valid_pos = df_valid.loc[rows_pos]
df_valid_neg = df_valid.loc[~rows_pos]

# merge the balanced data
df_valid = pd.concat([df_valid_pos, df_valid_neg.sample(n = len(df_valid_pos), random_state = 42)],axis = 0)

# shuffle the order of training samples
```



```
df_valid = df_valid.sample(n = len(df_valid), random_state = 42).reset_index(drop = True)

print('Valid balanced prevalence(n = %d):%.3f'%(len(df_valid), calc_prevalence(df_train.OUTPUT_LABEL.values)))
```

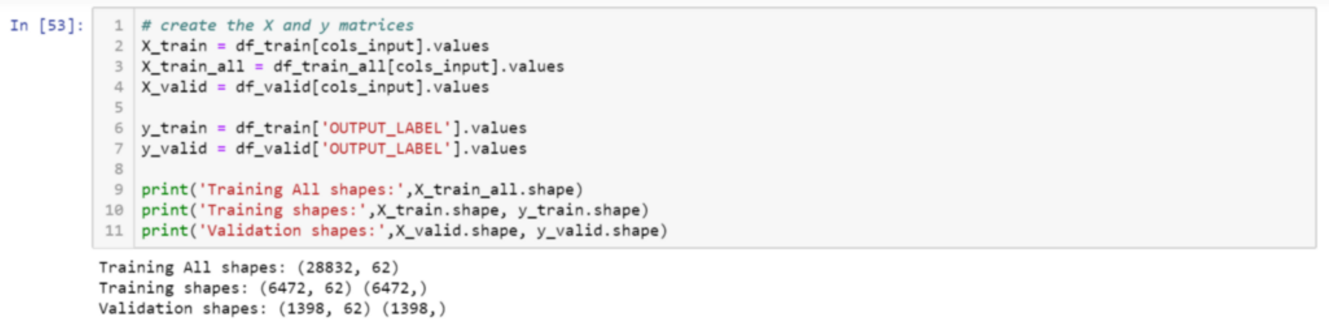
```
Valid balanced prevalence(n = 1398):0.500
```



```
In [46]:
# split the test into positive and negative
rows_pos = df_test.OUTPUT_LABEL == 1
df_test_pos = df_test.loc[rows_pos]
df_test_neg = df_test.loc[~rows_pos]

# merge the balanced data
df_test = pd.concat([df_test_pos, df_test_neg.sample(n = len(df_test_pos), random_state = 42)],axis = 0)

# shuffle the order of training samples
df_test = df_test.sample(n = len(df_test), random_state = 42).reset_index(drop = True)

print('Test balanced prevalence(n = %d):%.3f'%(len(df_test), calc_prevalence(df_train.OUTPUT_LABEL.values)))
```

```
Test balanced prevalence(n = 1410):0.500
```

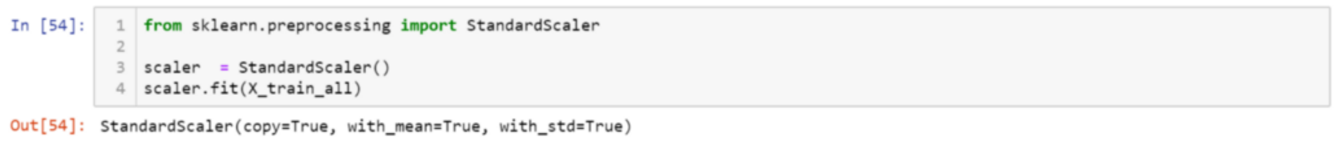
Most machine learning packages like to implement an input matrix X and output vector y, so let’s create those:



```
In [53]:
# create the X and y matrices
X_train = df_train[cols_input].values
X_train_all = df_train_all[cols_input].values
X_valid = df_valid[cols_input].values

y_train = df_train['OUTPUT_LABEL'].values
y_valid = df_valid['OUTPUT_LABEL'].values

print('Training All shapes:',X_train_all.shape)
print('Training shapes:',X_train.shape, y_train.shape)
print('Validation shapes:',X_valid.shape, y_valid.shape)
```

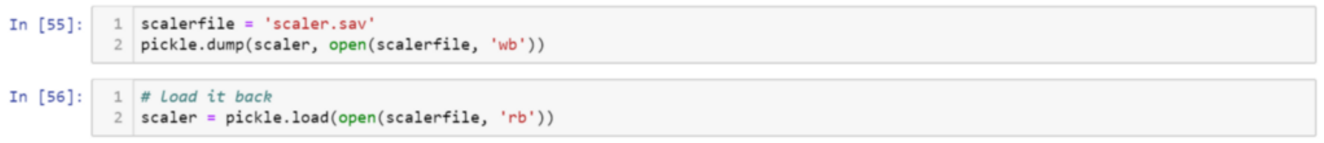
```
Training All shapes: (28832, 62)
Training shapes: (6472, 62) (6472,)
Validation shapes: (1398, 62) (1398,)
```

There can be troubles in machine learning models when the variables are of different size (0–100, vs 0–1000000). To combat this, we can scale the data. Here we will use scikit-learn’s `Standard Scaler` which removes the mean and scales to unit variance. Here I will create a scaler using all the training data, but you could also use the balanced one if you wanted.



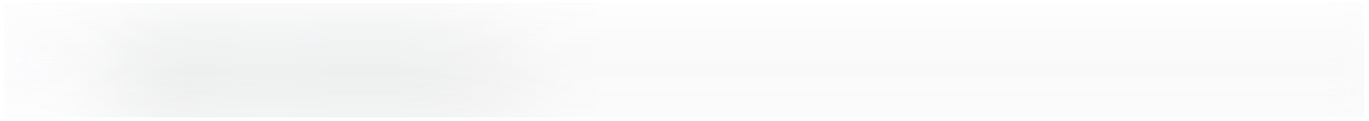
```
In [54]: from sklearn.preprocessing import StandardScaler

         scaler  = StandardScaler()
         scaler.fit(X_train_all)

Out[54]: StandardScaler(copy=True, with_mean=True, with_std=True)
```

We are going to need this scaler for the test data, so let's save it using a package called `pickle`.



```
In [55]: scalerfile = 'scaler.sav'
         pickle.dump(scaler, open(scalerfile, 'wb'))

In [56]: # Load it back
         scaler = pickle.load(open(scalerfile, 'rb'))
```

Now we can go ahead and transform our data matrices:



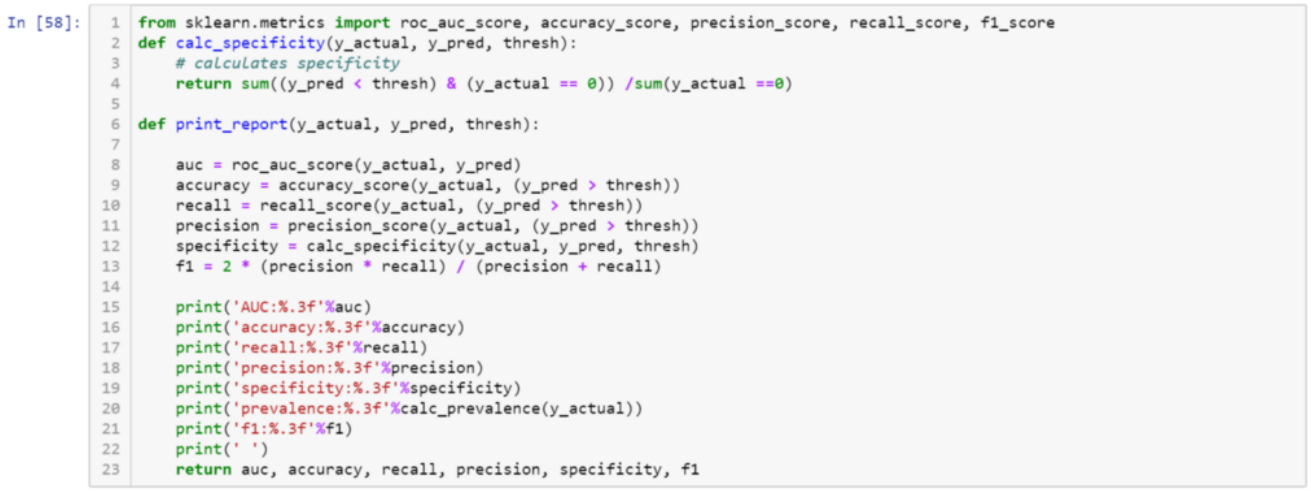
We won't transform the test matrix yet, to prevent us from being tempted to look at the performance until we are done with model selection.

Model Selection

Fantastic! We had to do a lot of work to prep the data! Which is the norm in data science. You can spend up to 90% cleaning and preparing data before analyzing!

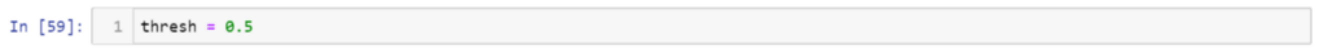
In this section, we train a few machine learning models and use a few techniques for optimizing them. We will then select the best model based on performance on the validation set.

We will utilize the following functions to evaluate the performance of the model — AUC (Area Under the ROC Curve), Accuracy, Recall, Precision, Specificity and F1!



In [58]:
```
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score, f1_score
def calc_specificity(y_actual, y_pred, thresh):
    # calculates specificity
    return sum((y_pred < thresh) & (y_actual == 0)) /sum(y_actual ==0)

def print_report(y_actual, y_pred, thresh):

    auc = roc_auc_score(y_actual, y_pred)
    accuracy = accuracy_score(y_actual, (y_pred > thresh))
    recall = recall_score(y_actual, (y_pred > thresh))
    precision = precision_score(y_actual, (y_pred > thresh))
    specificity = calc_specificity(y_actual, y_pred, thresh)
    f1 = 2 * (precision * recall) / (precision + recall)

    print('AUC:%.3f'%auc)
    print('accuracy:%.3f'%accuracy)
    print('recall:%.3f'%recall)
    print('precision:%.3f'%precision)
    print('specificity:%.3f'%specificity)
    print('prevalence:%.3f'%calc_prevalence(y_actual))
    print('f1:%.3f'%f1)
    print(' ')
    return auc, accuracy, recall, precision, specificity, f1
```

Since we have a balanced training data, let’s set our threshold at 0.5 to label a predicted sample as positive.



```
In [59]: thresh = 0.5
```

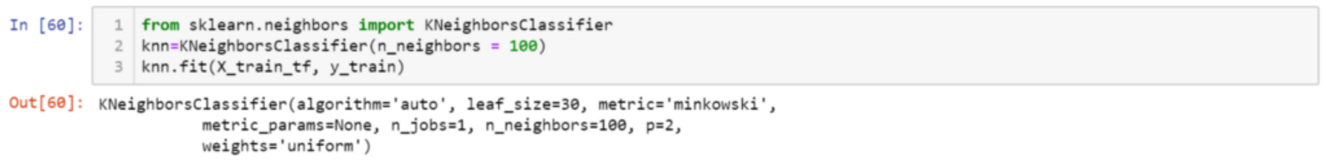
Model Selection: Baseline models

In this section, we will first compare 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

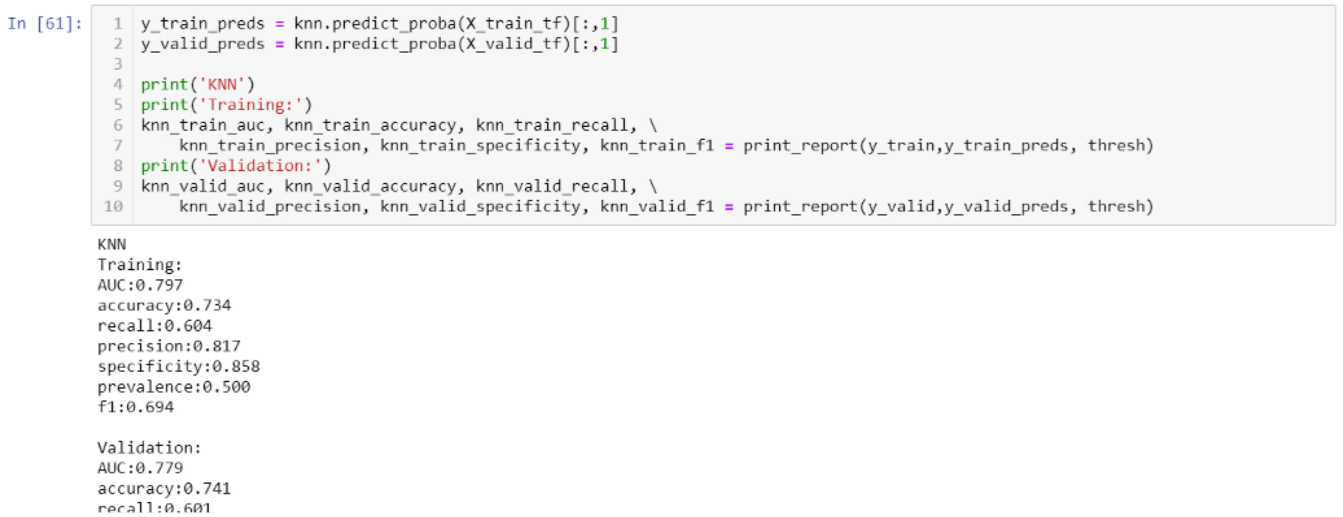
K Nearest Neighbors (KNN)

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. We can fit KNN using the following code from scikit-learn:

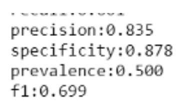


```
In [60]:  from sklearn.neighbors import KNeighborsClassifier
          knn=KNeighborsClassifier(n_neighbors = 100)
          knn.fit(X_train_tf, y_train)

Out[60]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=1, n_neighbors=100, p=2,
                    weights='uniform')
```

We can evaluate the model performance with the following code:



```
In [61]:  y_train_preds = knn.predict_proba(X_train_tf)[:,1]
          y_valid_preds = knn.predict_proba(X_valid_tf)[:,1]

          print('KNN')
          print('Training:')
          knn_train_auc, knn_train_accuracy, knn_train_recall, \
              knn_train_precision, knn_train_specificity, knn_train_f1 = print_report(y_train,y_train_preds, thresh)
          print('Validation:')
          knn_valid_auc, knn_valid_accuracy, knn_valid_recall, \
              knn_valid_precision, knn_valid_specificity, knn_valid_f1 = print_report(y_valid,y_valid_preds, thresh)

          KNN
          Training:
          AUC:0.797
          accuracy:0.734
          recall:0.604
          precision:0.817
          specificity:0.858
          prevalence:0.500
          f1:0.694

          Validation:
          AUC:0.779
          accuracy:0.741
          recall:0.601
```

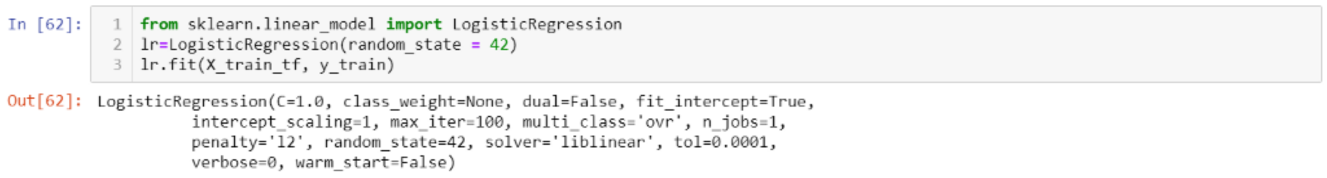


```
precision:0.835
specificity:0.878
prevalence:0.500
f1:0.699
```

To be brief, we will exclude the evaluation from the remaining models and only show the aggregated results below.

Logistic Regression

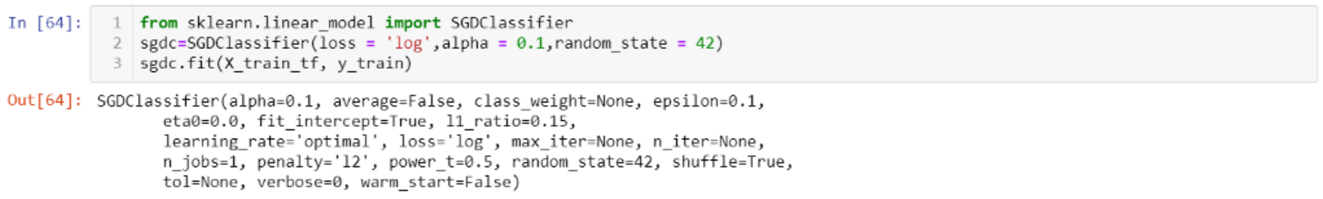
Logistic regression is a traditional machine learning model that fits a linear decision boundary between the positive and negative samples. Logsitic regression uses a line (Sigmoid function) in the form of an "S" to predict if the dependent variable is true or false based on the independent variables. 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. We can fit logistic regression using the following code from scikit-learn.



```
In [62]:  from sklearn.linear_model import LogisticRegression
          lr=LogisticRegression(random_state = 42)
          lr.fit(X_train_tf, y_train)

Out[62]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=42, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

Stochastic Gradient Descent

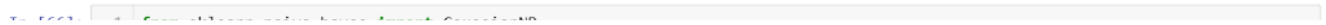
Stochastic gradient descent is similar to logistic regression. Stochastic Gradient Descent analyzes various sections of the data instead of the data as a whole and predicts the output using the independent variables. Stochastic Gradient Descent is faster than logistic regression in the sense that it doesn't run the whole dataset but instead looks at different parts of the dataset. We can fit stochastic gradient descent using the following code from scikit-learn.

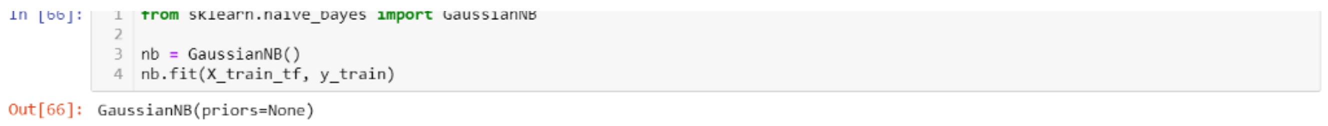


```
In [64]: from sklearn.linear_model import SGDClassifier
         sgdc=SGDClassifier(loss = 'log',alpha = 0.1,random_state = 42)
         sgdc.fit(X_train_tf, y_train)

Out[64]: SGDClassifier(alpha=0.1, average=False, class_weight=None, epsilon=0.1,
                eta0=0.0, fit_intercept=True, l1_ratio=0.15,
                learning_rate='optimal', loss='log', max_iter=None, n_iter=None,
                n_jobs=1, penalty='l2', power_t=0.5, random_state=42, shuffle=True,
                tol=None, verbose=0, warm_start=False)
```

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. This works well for robotics and computer vision, but we can also try it here! We can fit Naive Bayes with the following code.

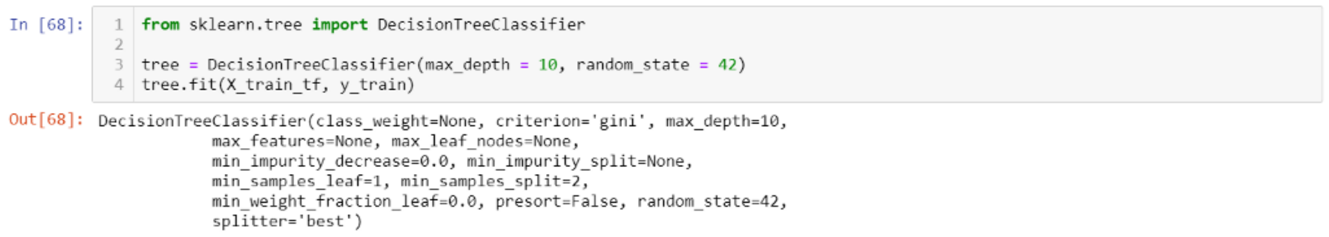




```
In [66]: from sklearn.naive_bayes import GaussianNB

         nb = GaussianNB()
         nb.fit(X_train_tf, y_train)

Out[66]: GaussianNB(priors=None)
```

Decision Tree

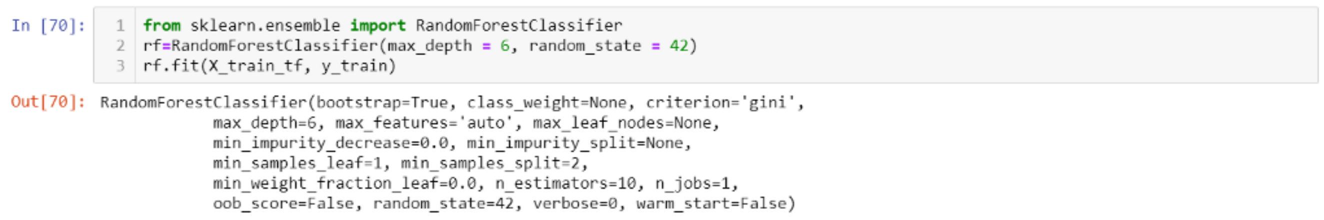
Another class of popular machine learning models is tree-based methods. The simplest tree-based method is known as a decision tree. The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by *learning simple decision rules* gotten 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 are able to pick up non-linear effects if given sufficient tree depth. We can fit decision trees using the following code.



```
In [68]: from sklearn.tree import DecisionTreeClassifier

         tree = DecisionTreeClassifier(max_depth = 10, random_state = 42)
         tree.fit(X_train_tf, y_train)

Out[68]: DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=10,
                     max_features=None, max_leaf_nodes=None,
                     min_impurity_decrease=0.0, min_impurity_split=None,
                     min_samples_leaf=1, min_samples_split=2,
                     min_weight_fraction_leaf=0.0, presort=False, random_state=42,
                     splitter='best')
```

Random forest

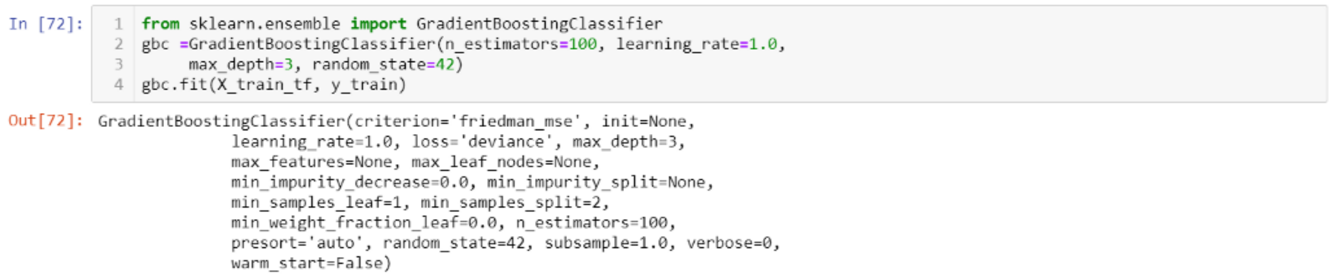
One disadvantage of decision trees is that they tend overfit very easily by memorizing the training data. Overfitting occurs when a model learns the detail and noise in the training data to the extent that it negatively impacts the performance of the model on new data. Random forests were created to 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 are able to generalize more easily. To fit random forests, we can use the following code.



```
In [70]: from sklearn.ensemble import RandomForestClassifier
         rf=RandomForestClassifier(max_depth = 6, random_state = 42)
         rf.fit(X_train_tf, y_train)

Out[70]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                     max_depth=6, max_features='auto', max_leaf_nodes=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=10, n_jobs=1,
                     oob_score=False, random_state=42, verbose=0, warm_start=False)
```

Gradient Boosting Classifier

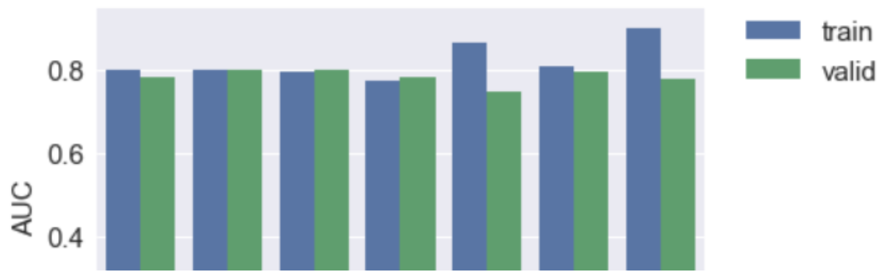
Boosting is a technique that builds a new decision tree algorithm that focuses on the errors on the dataset to fix them. This learns the whole model in other to fix it and improve the prediction of the model. A model that uses this technique combined with a gradient descent algorithm (controlling learning rate) is known as gradient boosting classifier. One advantage is the XGBoost library is the determining factor in winning a lot of Kaggle data science competitions! To fit the gradient boosting classifier, we can apply the following code.

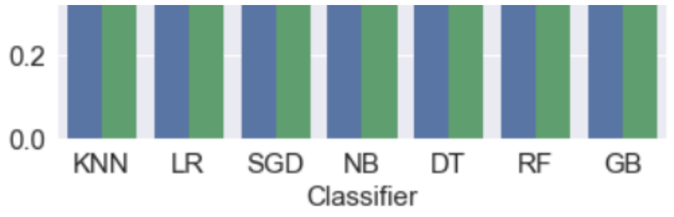


```
In [72]: from sklearn.ensemble import GradientBoostingClassifier
         gbc =GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
              max_depth=3, random_state=42)
         gbc.fit(X_train_tf, y_train)

Out[72]: GradientBoostingClassifier(criterion='friedman_mse', init=None,
                       learning_rate=1.0, loss='deviance', max_depth=3,
                       max_features=None, max_leaf_nodes=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,
                       presort='auto', random_state=42, subsample=1.0, verbose=0,
                       warm_start=False)
```

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` . We will utilize the AUC to evaluate the best model. This is a good data science performance metric for picking the best model since it captures the trade off between the true positive and false positive and does not require selecting a threshold.

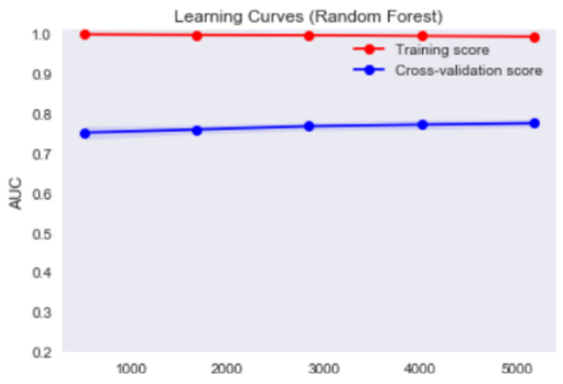


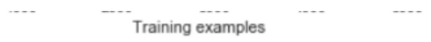


As we can see most of the models (except decision tree) have similar performance on the validation set. There is some overfitting as noted by the drop between training and validation. Let's check if we can improve this performance using a few more techniques.

Model Selection: Learning Curve

In this section, we can diagnose how our models are doing by plotting a learning curve. In this section, we 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

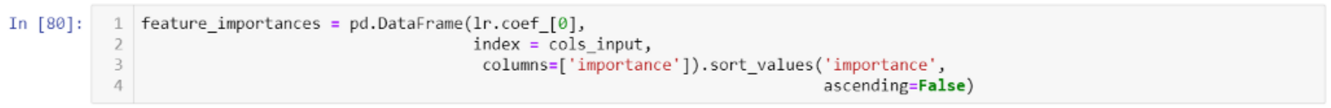
High Variance:
- Reduce number of features
- Decrease model complexity
- Add regularization
- Add more samples

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

Model Selection: Feature Importance

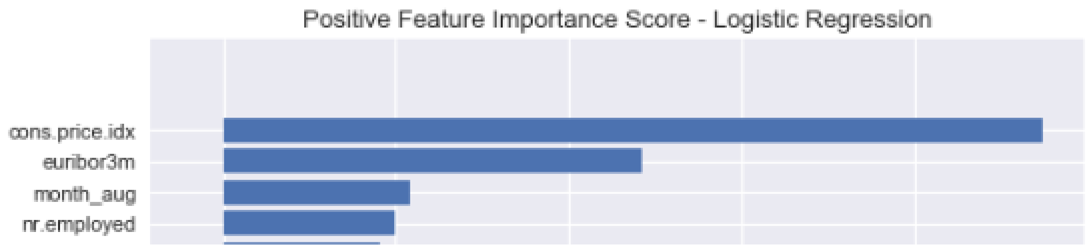
A way of improving your models to understand what features are important to your models. This can usually only be investigated for simpler models such as Logistic Regression or Random Forests. This analysis can help in certain areas:
— inspire new feature ideas : assists with both high bias and high variance
— obtain a list of the top features to be used for feature reduction: helps with high variance
— point out errors in your pipeline: helps with robustness of model

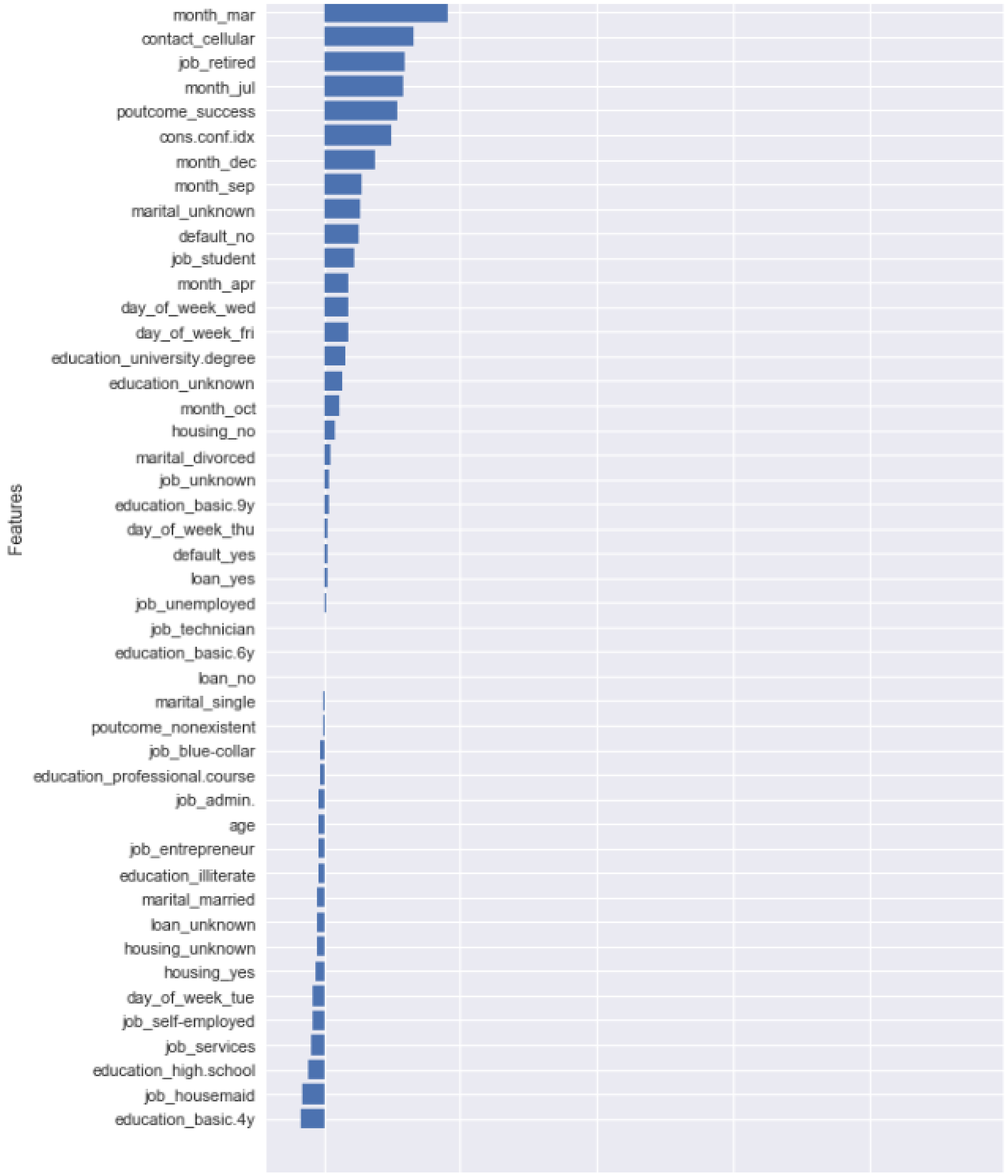
We can get the feature importance from logistic regression using the below.

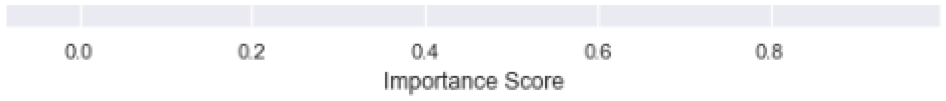


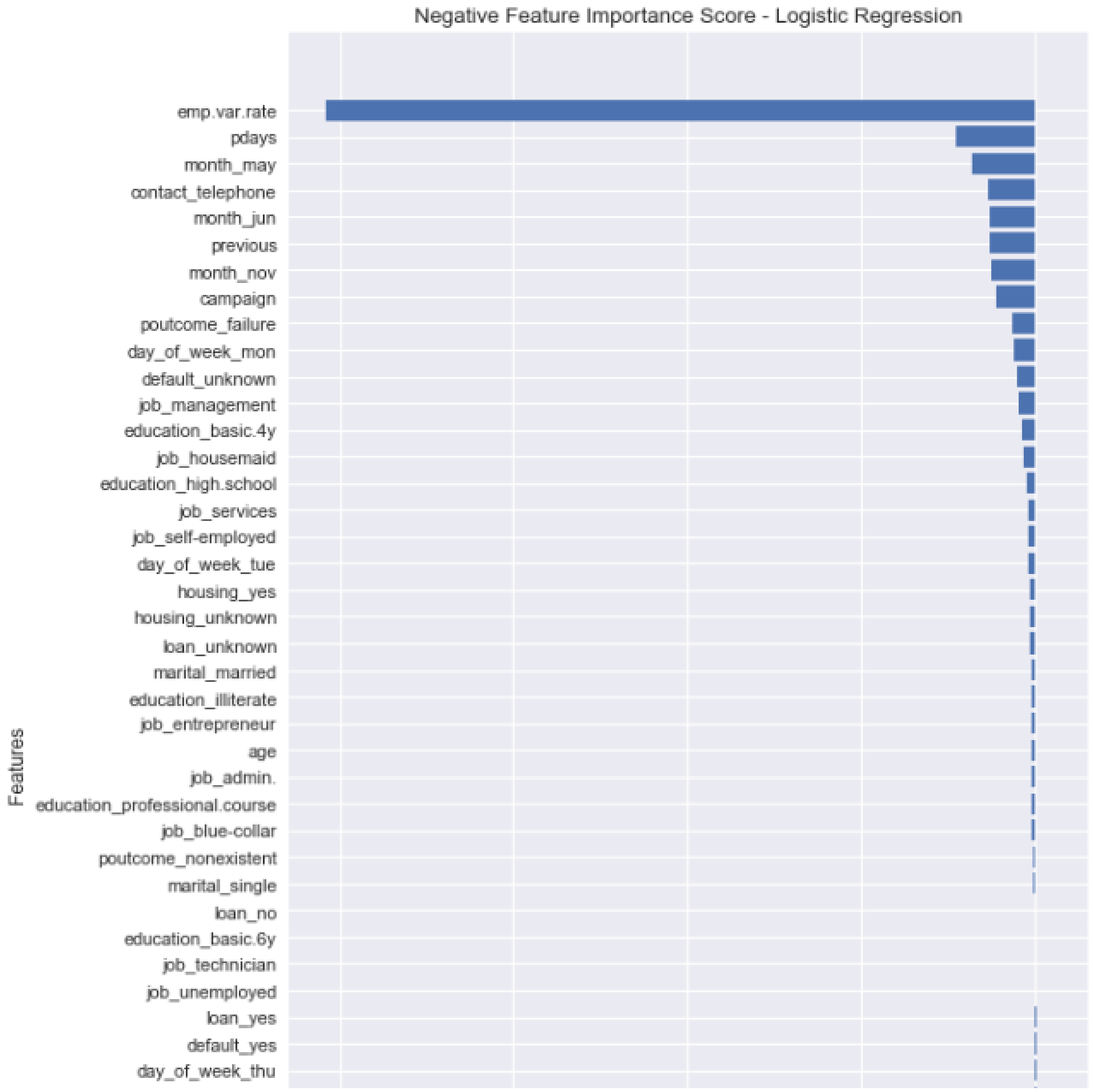
```
In [80]: feature_importances = pd.DataFrame(lr.coef_[0],
                                            index = cols_input,
                                             columns=['importance']).sort_values('importance',
                                                                                 ascending=False)
```

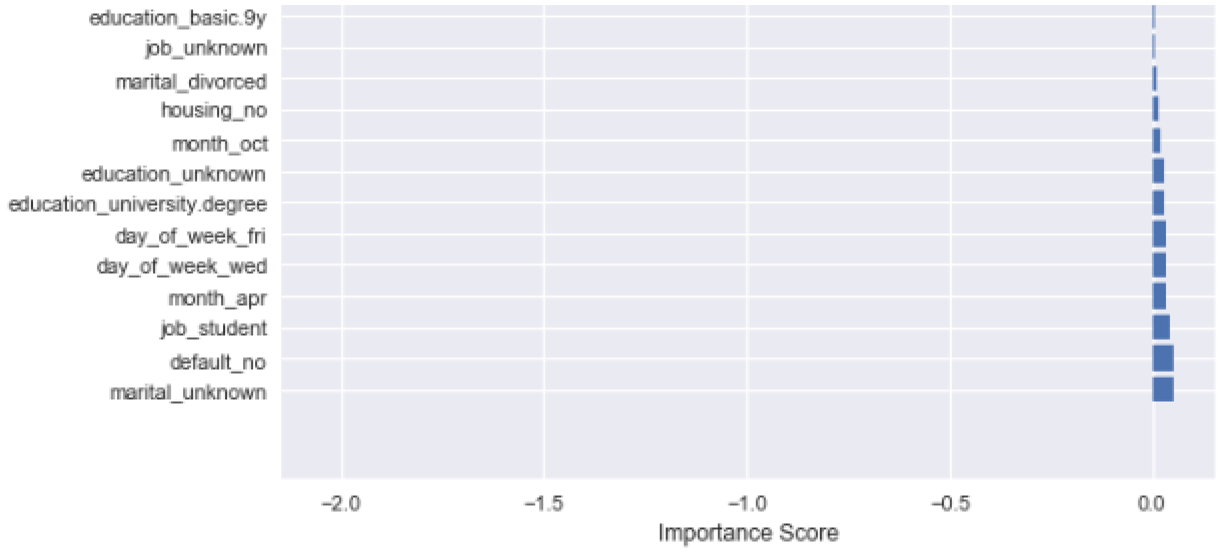
We can take a look at the top 50 positive and top 50 negative coefficients to get some insight.




month_mar
contact_cellular
job_retired
month_jul
poutcome_success
cons.conf.idx
month_dec
month_sep
marital_unknown
default_no
job_student
month_apr
day_of_week_wed
day_of_week_fri
education_university.degree
education_unknown
month_oct
housing_no
marital_divorced
job_unknown
education_basic.9y
day_of_week_thu
default_yes
loan_yes
job_unemployed
job_technician
education_basic.6y
loan_no
marital_single
poutcome_nonexistent
job_blue-collar
education_professional.course
job_admin.
age
job_entrepreneur
education_illiterate
marital_married
loan_unknown
housing_unknown
housing_yes
day_of_week_tue
job_self-employed
job_services
education_high.school
job_housemaid
education_basic.4y
Features


0.0
0.2
0.4
0.6
0.8
Importance Score


Negative Feature Importance Score - Logistic Regression
Features
emp.var.rate
pdays
month_may
contact_telephone
month_jun
previous
month_nov
campaign
poutcome_failure
day_of_week_mon
default_unknown
job_management
education_basic.4y
job_housemaid
education_high.school
job_services
job_self-employed
day_of_week_tue
housing_yes
housing_unknown
loan_unknown
marital_married
education_illiterate
job_entrepreneur
age
job_admin.
education_professional.course
job_blue-collar
poutcome_nonexistent
marital_single
loan_no
education_basic.6y
job_technician
job_unemployed
loan_yes
default_yes
day_of_week_thu



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.

In a high variance situation, a technique that can be used is to reduce the number of variables to minimize overfitting. After this analysis, you could apply the top N positive and negative features or the top N important random forest features. You might need to adjust N so that your performance does not drop drastically. An example is only using the top feature will likely drop the performance by a lot.

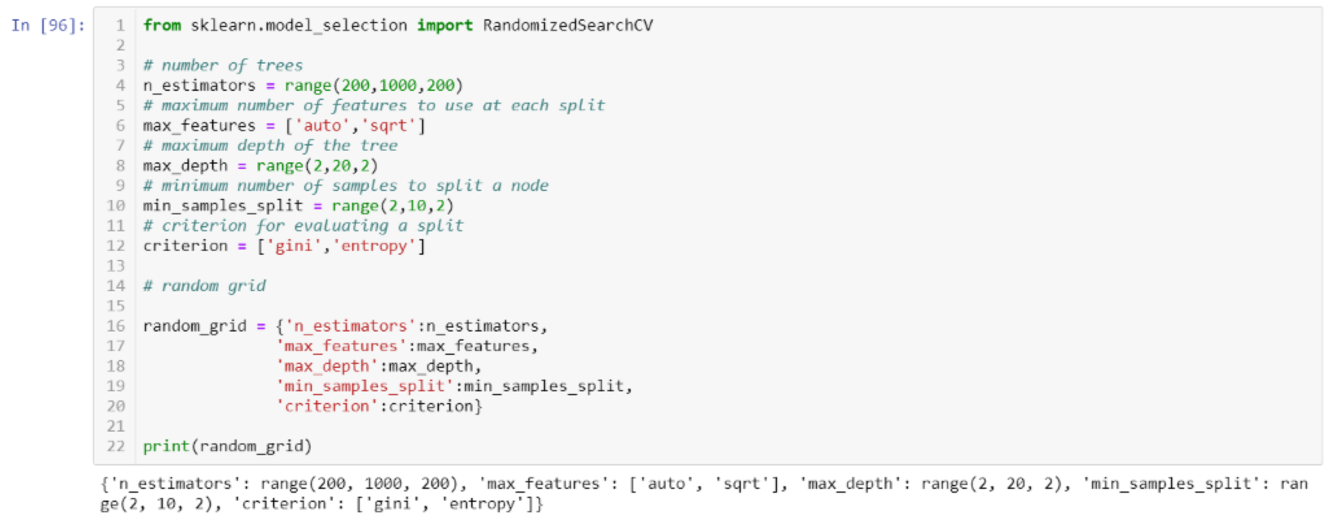
Feature importance plots may also alert errors in your predictive learning model. You may have some data leakage in the cleaning process. Data leakage can be explained as the process of accidentally including something in the training that allows the machine learning algorithm to artificially cheat. Similar things can also happen when you combine datasets. Supposing when you merged the datasets one of the classes ended up with nan for some of the variables.

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.

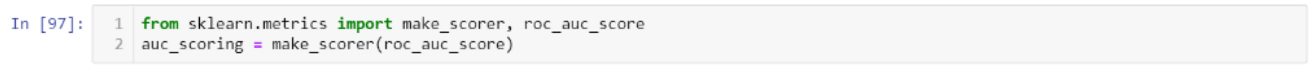
Now, we can create a grid over the random forest hyperparameters.



```
In [96]:
from sklearn.model_selection import RandomizedSearchCV

# number of trees
n_estimators = range(200,1000,200)
# maximum number of features to use at each split
max_features = ['auto','sqrt']
# maximum depth of the tree
max_depth = range(2,20,2)
# minimum number of samples to split a node
min_samples_split = range(2,10,2)
# criterion for evaluating a split
criterion = ['gini','entropy']

# random grid

random_grid = {'n_estimators':n_estimators,
               'max_features':max_features,
               'max_depth':max_depth,
               'min_samples_split':min_samples_split,
               'criterion':criterion}

print(random_grid)
```

```
{'n_estimators': range(200, 1000, 200), 'max_features': ['auto', 'sqrt'], 'max_depth': range(2, 20, 2), 'min_samples_split': ran
ge(2, 10, 2), 'criterion': ['gini', 'entropy']}
```

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



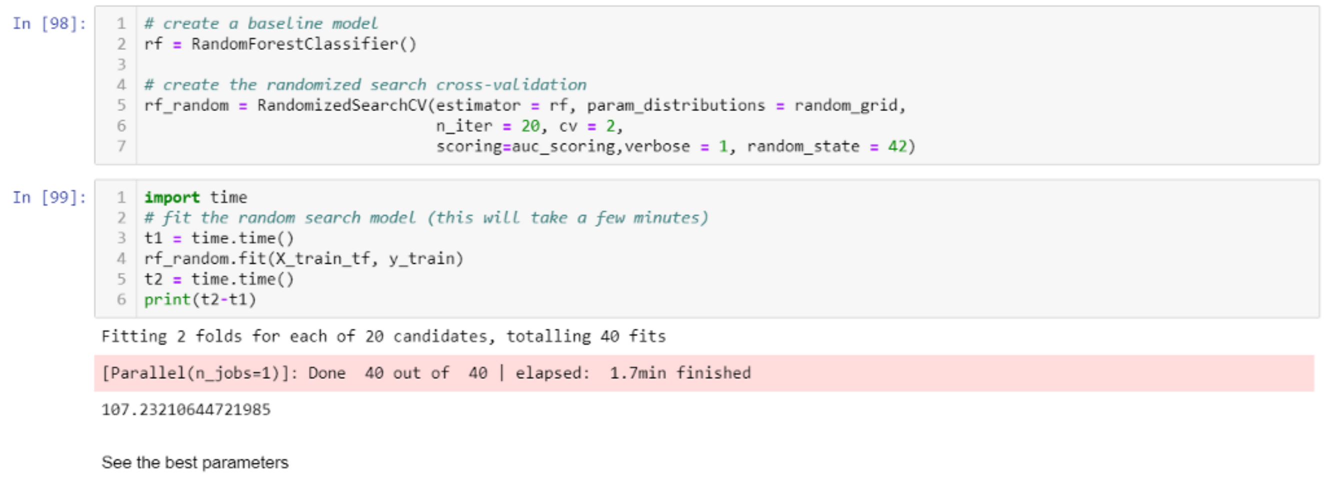
```
In [97]:
from sklearn.metrics import make_scorer, roc_auc_score
auc_scoring = make_scorer(roc_auc_score)
```

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

Note that increasing the last two of these will increase the run-time, but will decrease chance of overfitting. The number of variables and grid size also influences the runtime. Cross-validation is a method for splitting the data multiple times to get a better estimate of the performance metric. For the purposes of this project, we will limit to 2 CV to reduce the time.

Let's fit our Randomized Search random forest with the following code.

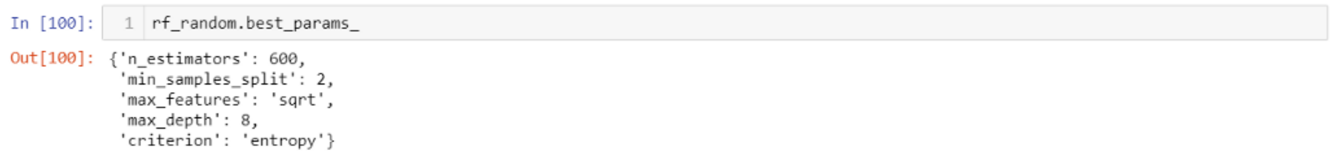


```
In [98]:
# create a baseline model
rf = RandomForestClassifier()

# create the randomized search cross-validation
rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid,
                               n_iter = 20, cv = 2,
                               scoring=auc_scoring,verbose = 1, random_state = 42)
```

```
In [99]:
import time
# fit the random search model (this will take a few minutes)
t1 = time.time()
rf_random.fit(X_train_tf, y_train)
t2 = time.time()
print(t2-t1)
```

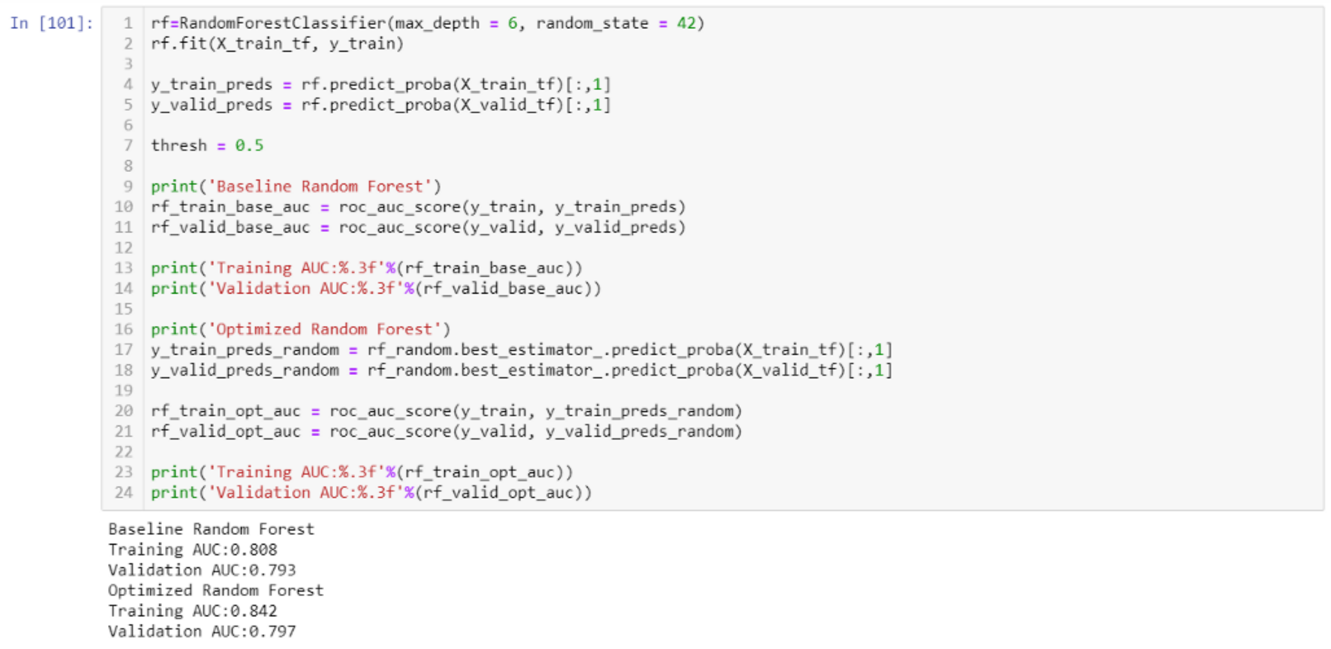
```
Fitting 2 folds for each of 20 candidates, totalling 40 fits
[Parallel(n_jobs=1)]: Done  40 out of  40 | elapsed:  1.7min finished
107.23210644721985
```

See the best parameters



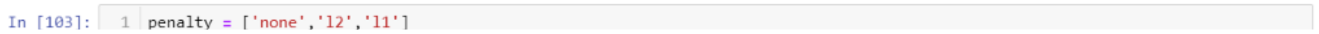
```
In [100]: rf_random.best_params_

Out[100]: {'n_estimators': 600,
           'min_samples_split': 2,
           'max_features': 'sqrt',
           'max_depth': 8,
           'criterion': 'entropy'}
```

We can analyze the performance of the best model compared to the baseline model.

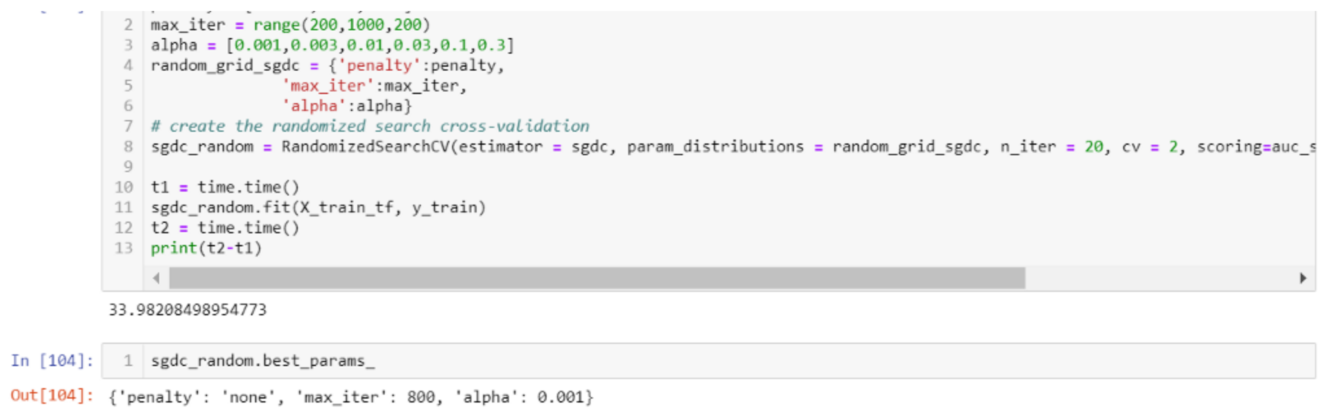


```
In [101]: rf=RandomForestClassifier(max_depth = 6, random_state = 42)
          rf.fit(X_train_tf, y_train)

          y_train_preds = rf.predict_proba(X_train_tf)[:,1]
          y_valid_preds = rf.predict_proba(X_valid_tf)[:,1]

          thresh = 0.5

          print('Baseline Random Forest')
          rf_train_base_auc = roc_auc_score(y_train, y_train_preds)
          rf_valid_base_auc = roc_auc_score(y_valid, y_valid_preds)

          print('Training AUC:%.3f'%(rf_train_base_auc))
          print('Validation AUC:%.3f'%(rf_valid_base_auc))

          print('Optimized Random Forest')
          y_train_preds_random = rf_random.best_estimator_.predict_proba(X_train_tf)[:,1]
          y_valid_preds_random = rf_random.best_estimator_.predict_proba(X_valid_tf)[:,1]

          rf_train_opt_auc = roc_auc_score(y_train, y_train_preds_random)
          rf_valid_opt_auc = roc_auc_score(y_valid, y_valid_preds_random)

          print('Training AUC:%.3f'%(rf_train_opt_auc))
          print('Validation AUC:%.3f'%(rf_valid_opt_auc))

          Baseline Random Forest
          Training AUC:0.808
          Validation AUC:0.793
          Optimized Random Forest
          Training AUC:0.842
          Validation AUC:0.797
```

In the same way,we can optimize the performance of stochastic gradient descent and gradient boosting classifiers.



```
In [103]: penalty = ['none','l2','l1']
```

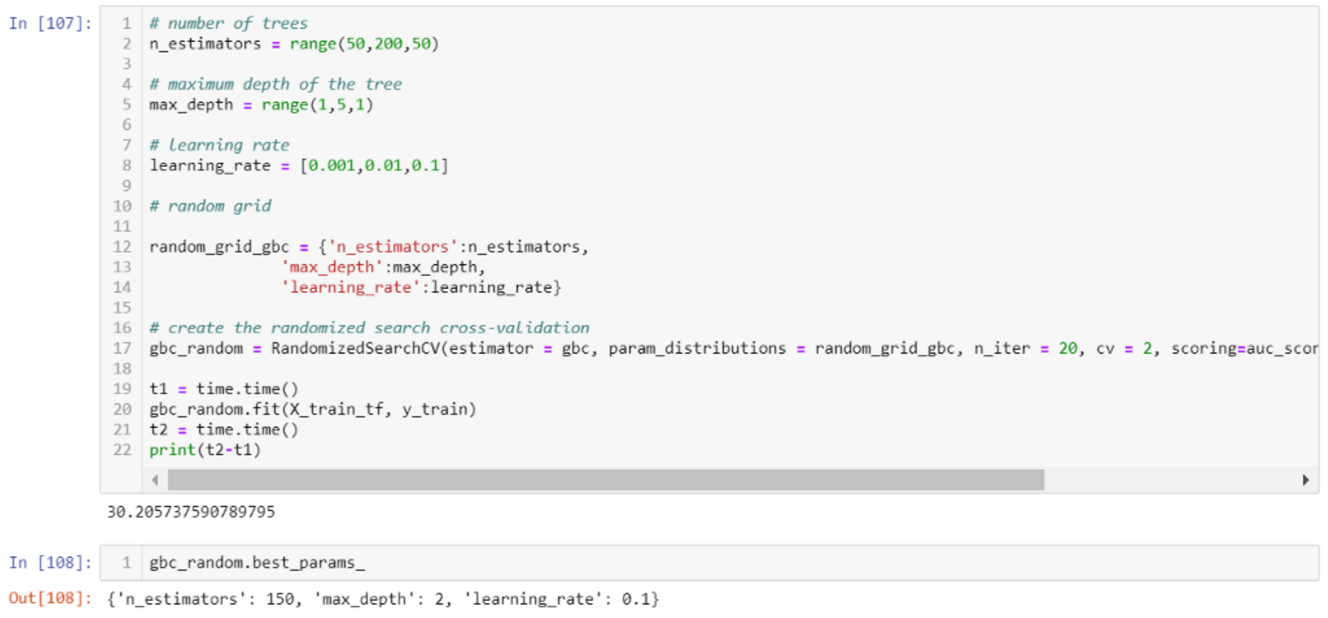


```
max_iter = range(200,1000,200)
alpha = [0.001,0.003,0.01,0.03,0.1,0.3]
random_grid_sgdc = {'penalty':penalty,
              'max_iter':max_iter,
              'alpha':alpha}
# create the randomized search cross-validation
sgdc_random = RandomizedSearchCV(estimator = sgdc, param_distributions = random_grid_sgdc, n_iter = 20, cv = 2, scoring=auc_s

t1 = time.time()
sgdc_random.fit(X_train_tf, y_train)
t2 = time.time()
print(t2-t1)
```

```
33.98208498954773
```

In [104]:
```
sgdc_random.best_params_
```

Out[104]:
```
{'penalty': 'none', 'max_iter': 800, 'alpha': 0.001}
```



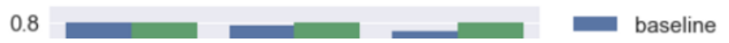
In [107]:
```
# number of trees
n_estimators = range(50,200,50)

# maximum depth of the tree
max_depth = range(1,5,1)

# learning rate
learning_rate = [0.001,0.01,0.1]

# random grid

random_grid_gbc = {'n_estimators':n_estimators,
              'max_depth':max_depth,
              'learning_rate':learning_rate}

# create the randomized search cross-validation
gbc_random = RandomizedSearchCV(estimator = gbc, param_distributions = random_grid_gbc, n_iter = 20, cv = 2, scoring=auc_scor

t1 = time.time()
gbc_random.fit(X_train_tf, y_train)
t2 = time.time()
print(t2-t1)
```

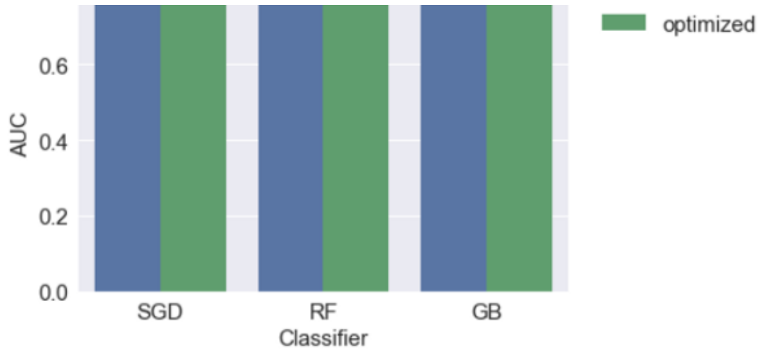
```
30.205737590789795
```

In [108]:
```
gbc_random.best_params_
```

Out[108]:
```
{'n_estimators': 150, 'max_depth': 2, 'learning_rate': 0.1}
```

Here We can aggregate the results and compare to the baseline models on the validation set.

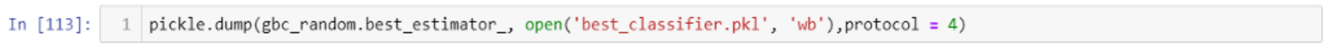




Looking at the results, we can see that the hyperparameter tuning improved the models, but not by much. This is most likely due to the fact that we have a high variance situation.

Model Selection: Best Classifier

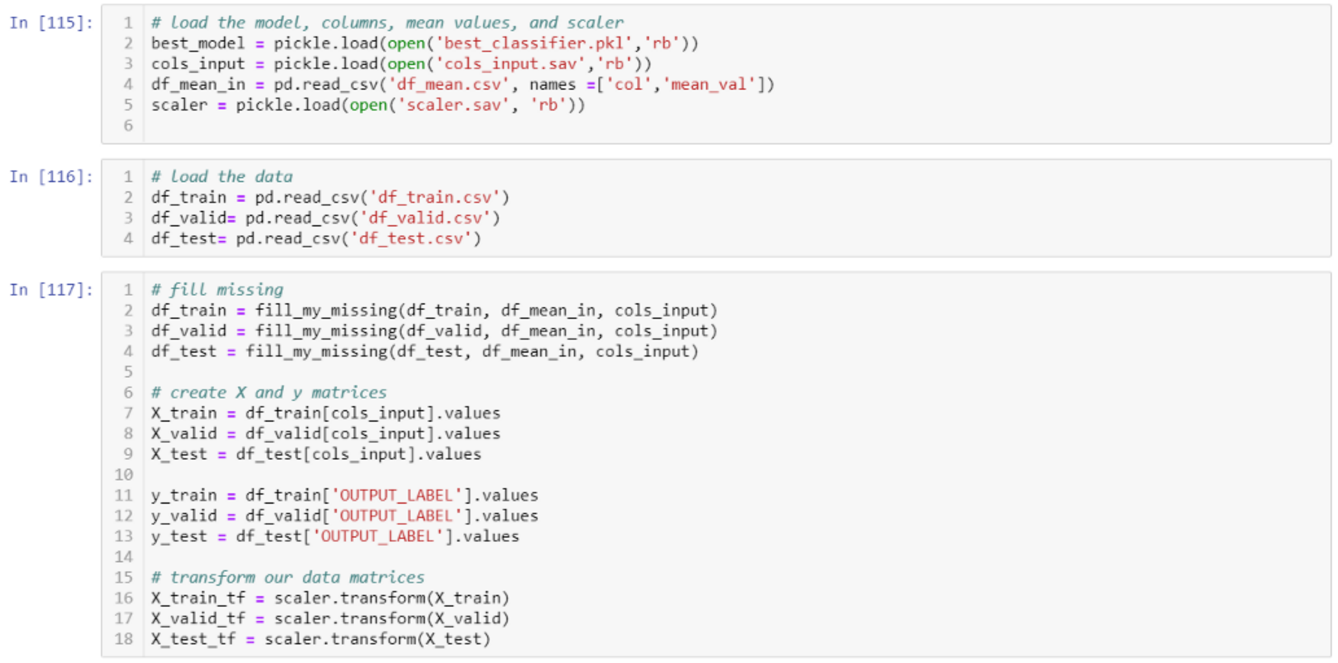
In this phase, we will chose the gradient boosting classifier since it has the best AUC on the validation set. You won't want to train your best classifier every time you want to run new predictions. Therefore, we need to save the classifier. We will use the package `pickle`.



```
In [113]: pickle.dump(gbc_random.best_estimator_, open('best_classifier.pkl', 'wb'),protocol = 4)
```

Model Evaluation

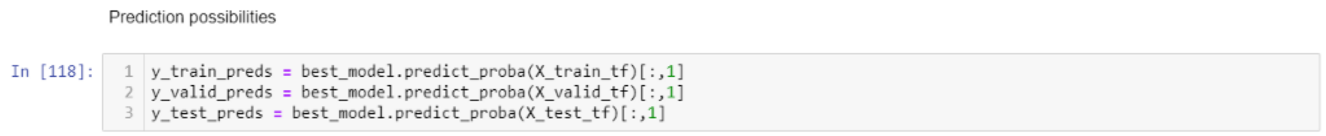
Now that we have chosen our best model (optimized gradient boosting classifier). Let's evaluate the performance of the test set.



```
In [115]:
# load the model, columns, mean values, and scaler
best_model = pickle.load(open('best_classifier.pkl','rb'))
cols_input = pickle.load(open('cols_input.sav','rb'))
df_mean_in = pd.read_csv('df_mean.csv', names =['col','mean_val'])
scaler = pickle.load(open('scaler.sav', 'rb'))

```

```
In [116]:
# load the data
df_train = pd.read_csv('df_train.csv')
df_valid= pd.read_csv('df_valid.csv')
df_test= pd.read_csv('df_test.csv')
```

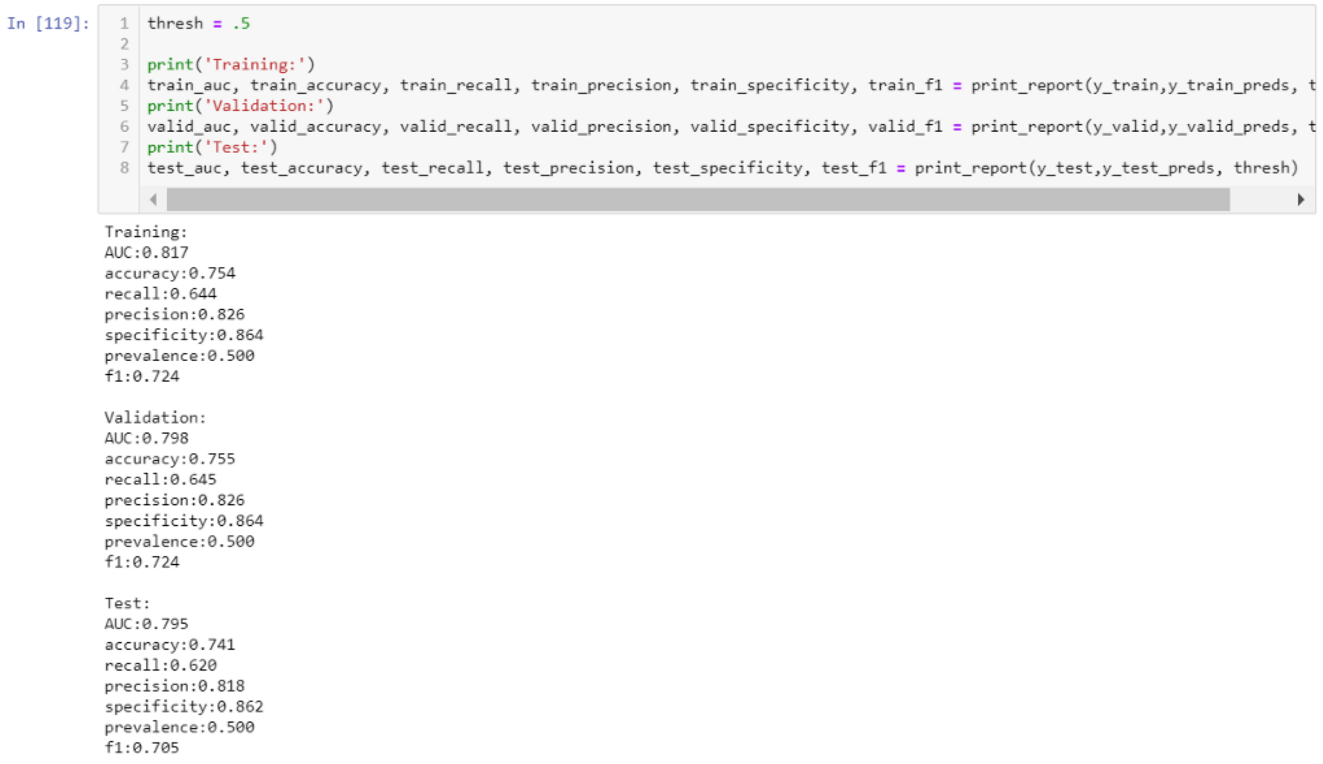
```
In [117]:
# fill missing
df_train = fill_my_missing(df_train, df_mean_in, cols_input)
df_valid = fill_my_missing(df_valid, df_mean_in, cols_input)
df_test = fill_my_missing(df_test, df_mean_in, cols_input)

# create X and y matrices
X_train = df_train[cols_input].values
X_valid = df_valid[cols_input].values
X_test = df_test[cols_input].values

y_train = df_train['OUTPUT_LABEL'].values
y_valid = df_valid['OUTPUT_LABEL'].values
y_test = df_test['OUTPUT_LABEL'].values

# transform our data matrices
X_train_tf = scaler.transform(X_train)
X_valid_tf = scaler.transform(X_valid)
X_test_tf = scaler.transform(X_test)
```



Prediction possibilities

```
In [118]:
y_train_preds = best_model.predict_proba(X_train_tf)[:,1]
y_valid_preds = best_model.predict_proba(X_valid_tf)[:,1]
y_test_preds = best_model.predict_proba(X_test_tf)[:,1]
```

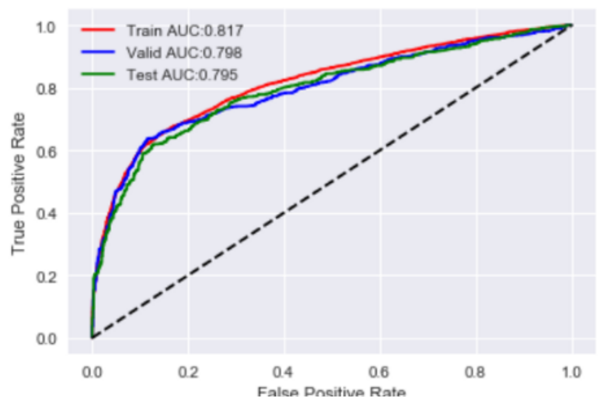
Lastly, The final evaluation is shown below!



```
In [119]:
thresh = .5

print('Training:')
train_auc, train_accuracy, train_recall, train_precision, train_specificity, train_f1 = print_report(y_train,y_train_preds, t
print('Validation:')
valid_auc, valid_accuracy, valid_recall, valid_precision, valid_specificity, valid_f1 = print_report(y_valid,y_valid_preds, t
print('Test:')
test_auc, test_accuracy, test_recall, test_precision, test_specificity, test_f1 = print_report(y_test,y_test_preds, thresh)
```

```
Training:
AUC:0.817
accuracy:0.754
recall:0.644
precision:0.826
specificity:0.864
prevalence:0.500
f1:0.724

Validation:
AUC:0.798
accuracy:0.755
recall:0.645
precision:0.826
specificity:0.864
prevalence:0.500
f1:0.724

Test:
AUC:0.795
accuracy:0.741
recall:0.620
precision:0.818
specificity:0.862
prevalence:0.500
f1:0.705
```

Additionally, We can create the ROC curve for the 3 datasets as shown below:



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.

References

S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22–31, June 2014

A. Long. Using Machine Learning to Predict Hospital Readmission for Patients with Diabetes with Scikit-Learn. October 2018

Data Science Machine Learning Python Marketing Scikit Learn