Shad Griffin Follow

Jul 25, 2020 · 27 min read ★ · Listen

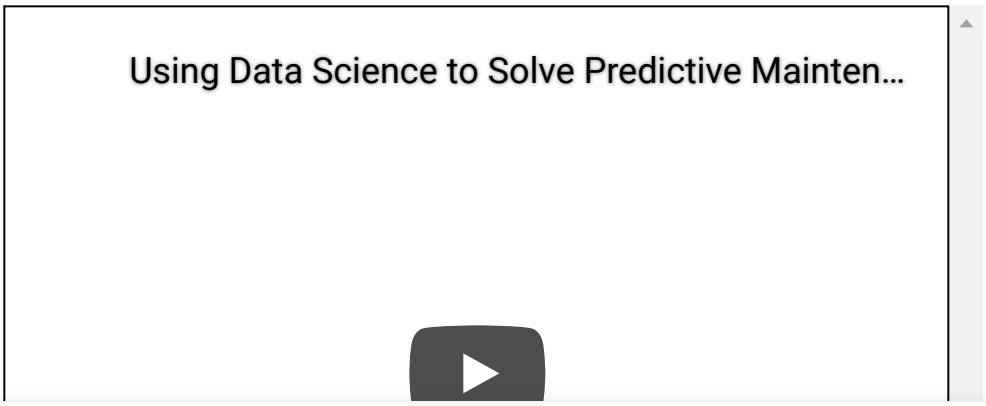
Machine Learning for Equipment Failure Prediction and Predictive Maintenance (PM)

> *I spent roughly four years of my life studying equipment failure problems as a Data Scientist. This article includes the better part of what I learned along the way.*

shadgriffin/machine_failure

Contribute to shadgriffin/machine_failure development by creating an account on GitHub.

github.com



Originally published in July 2020. Significantly revised February 2021.

In this notebook, I walk through a predictive maintenance problem in great detail. These types of problems can be tricky for several reasons. The first six sections deal with building a model. The last sections deal with evaluating model effectiveness and ensuring it will be effective when deployed in production.

When it comes to dealing with machines that require periodic maintenance, there are generally three possible outcomes.

One, you can maintain a machine too frequently. In other words, the machine gets maintenance when it is not required. In this scenario, you are throwing money out the window, wasting resources providing unnecessary maintenance. For example, you could change the oil in your car every single day. This is not optimal, and you will waste a lot of money on unnecessary maintenance.

Two, you don't maintain your machine frequently enough. Failing to maintain a machine means that the machine will break while operating. Here, the costs could be substantial. Not only do you have the repair costs, but also costs associated with lost production. If a machine on the assembly line goes down, the line cannot produce anything. No production means lost profit. Also, you will incur legal and medical costs if injuries

maintenance.

So, we need to maintain machines when they need maintenance, right? Unfortunately, this is easier said than done. Fortunately, we can use predictive maintenance (PM) to predict when machines need maintenance.

I should also mention that most machines come with manufacturer recommendations on maintenance. The problem with manufacturer recommendations is that they represent an average. For example, cars on average need an oil change every 3,000 miles, but how frequently does your car need an oil change? It may be more or less than 3,000 miles depending on several factors, including where your drive, how you drive, and how frequently you drive.

Predictive maintenance (PM) can tell you, based on data, when a machine requires maintenance. An effective PM program will minimize under and over-maintaining your machine. For a large manufacturer with thousands of machines, being precise on machine maintenance can save millions of dollars every year.

In this article, I will examine a typical Predictive Maintenance (PM) use case. As I walk through this example, I will describe some of the issues that arise with PM problems and suggest ways to solve them.

An important note about the data used in this exercise. It is entirely fake. I created the data based on my experience of dealing with these types of problems. Although it is entirely artificial, I believe the data and use case is very realistic and consistent with many real PM problems.

The firm in our use case provided a sample of data that includes 421 machines that failed over two years. They spent 11.766M dollars on maintenance, most of which came from running machines until failure.

Here is a summary of the maintained or repaired machines over the last two years.


Maintenance Scenario	Number of Incidents	Cost per Incident	Total Cost
Unnecessary Maintenance	9	$ 1,500	$ 13,500
Timely and Appropriate Maintenance	27	$ 7,500	$ 202,500
Machine Runs to Failure	385	$ 30,000	$ 11,550,000
Total	421	$ 27,948	$ 11,766,000

From the data above, it currently costs the firm about $28,000 per failed or maintained machine. Our goal is to lower this cost.

In the chart above, Timely Maintenance costs more than Unnecessary Maintenance. There is a good reason for this. For this machine, unnecessary maintenance means that that machine was moved off-line and checked, but the part in question showed insufficient wear to replace. Because parts were not replaced, there are no material costs, only labor.

Note that this company does very little predictive maintenance. Most of the time, they just run the machines to failure. Also, note that these machines will break in four to eight years if they don't receive maintenance. When they fail, they must be pulled off-line and repaired.

Our goal is to show the firm how a Predictive Maintenance program can save them money. To do this, we will build a predictive model that predicts machine failure within 90 days of actual failure. Note that an appropriate failure window will always depend on the context of the problem. If a machine breaks without maintenance in 6 months, a three-month window makes no sense. Here, where a machine will run between 4 to 6 years without maintenance, a 90-day window is reasonable.

Our objective is to develop a solution that will lower the costs of failure. Again, it currently costs the firm about 28,000 per machine. We will attempt to reduce this cost.

Note that I developed this exercise in Watson Studio on the IBM Cloud. If you have issues running the notebook, please set up a free account on IBM Cloud and try it there.

https://www.ibm.com/cloud/watson-studio

```
!pip install imblearn --upgrade
!pip install plotly --upgrade
!pip install chart-studio --upgrade
```

Import required libraries.

```
import chart_studio.plotly as py
import plotly.graph_objs as go
import plotly as plotly
import pandas as pd
import numpy as np
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import SMOTENC
from sklearn import metrics

from sklearn.preprocessing import LabelEncoder

import xgboost as xgb
from xgboost.sklearn import XGBClassifier

import types
import pandas as pd

def __iter__(self): return 0
```

Import the data from GitHub.

```
#Remove the data if you run this notebook more than once
!rm equipment_failure_data_1.csv

#import first half from github
!wget
https://raw.githubusercontent.com/shadgriffin/machine_failure/master/e
quipment_failure_data_1.csv

# Convert csv to pandas dataframe
pd_data_1 = pd.read_csv("equipment_failure_data_1.csv", sep=",",
header=0)
```


```
#Import the second half from github
!wget
https://raw.githubusercontent.com/shadgriffin/machine_failure/master/equipment_failure_data_2.csv

# convert to pandas dataframe
pd_data_2 = pd.read_csv("equipment_failure_data_2.csv", sep=",",
header=0)

#concatenate the two data files into one dataframe
pd_data=pd.concat([pd_data_1, pd_data_2])
```

2.0 Data Exporation

```
pd_data.head()
```

In [11]: pd_data.head()

Out[11]:

	ID	DATE	REGION_CLUSTER	MAINTENANCE_VENDOR	MANUFACTURER	WELL_GROUP	S15	S17	S13	S5	S16	S10	S18	EQUIPMENT_FAILURE	S8	AGE_OF_EQUIPMENT
0	100001	12/2/14	G	D	Y	1	11.088000	145.223448	38.34	3501.0	8.436885	1.9	24.610345	0	0.0	880
1	100001	12/3/14	G	D	Y	1	8.877945	187.673214	38.20	3489.0	8.483714	1.9	24.671429	0	0.0	881
2	100001	12/4/14	G	D	Y	1	8.878444	148.383704	38.57	3492.0	8.769655	2.0	24.733333	0	0.0	882
3	100001	12/5/14	G	D	Y	1	9.388338	133.660000	38.47	3513.0	8.330308	2.0	24.773077	0	0.0	883
4	100001	12/6/14	G	D	Y	1	8.475264	197.101000	40.33	3589.0	8.022940	1.5	24.800000	0	0.0	884

Now that we have the data imported into a Jupiter Notebook, we can explore it. Here is metadata explaining all of the fields in the data set.

ID — ID field that represents a specific machine.

DATE — The date of the observation.

REGION_CLUSTER — a field that represents the region in which the machine resides.

MAINTENANCE_VENDOR — a field that represents the company that provides maintenance and service to the machine.

MANUFACTURER — the company that manufactured the equipment in question

S15 — A Sensor Value.

S17 — A Sensor Value.

S13 — A Sensor Value.

S16 — A Sensor Value.

S19 — A Sensor Value.

S18 — A Sensor Value.

S8 — A Sensor Value.

EQUIPMENT_FAILURE — A '1' means that the equipment failed. A '0' means the equipment did not fail.

Our first goal in this exercise is to build a model that predicts equipment failure. In other words, we will use the other variables in the data frame to predict EQUIPMENT_FAILURE.

Now we will walk through the data.

Examine the number of rows and columns. The data has 307,751 rows and 16 columns.

```
pd_data.shape
```

```
Out[12]: (307751, 16)
```

There are 421 machines in the data set.

```
Out[13]: (421, 15)
```

There are 731 unique dates in the data set.

```
xxxx = pd.DataFrame(pd_data.groupby(['DATE']).agg(['count']))
xxxx.shape
```

```
Out[14]: (731, 15)
```

We have 731 unique dates. So if we have 421 machines and 731 unique dates, we should have 307,751 total records. Based on the .shape command, we have one record per machine per date value. There are no duplicates in the data frame.

And to triple confirm, remove all duplicates and count the rows again.

```
df_failure_thingy=pd_data
df_failure_thingy=df_failure_thingy.drop_duplicates(subset=
['ID','DATE'])
df_failure_thingy.shape
```

```
Out[15]: (307751, 16)
```

Look for null values in the fields — There are none.


```
Out[16]: ID                    0
         DATE                  0
         REGION_CLUSTER        0
         MAINTENANCE_VENDOR    0
         MANUFACTURER          0
         WELL_GROUP            0
         S15                   0
         S17                   0
         S13                   0
         S5                    0
         S16                   0
         S19                   0
         S18                   0
         EQUIPMENT_FAILURE     0
         S8                    0
         AGE_OF_EQUIPMENT      0
         dtype: int64
```

Now let's examine the dependent variable in more detail. It appears that out of 307,751 records, we only have 421 failures. This corresponds to a failure rate of about .14%. In other words, for every failure, you have over 700 non-failures. This data set is very unbalanced. Later in this article, I will use a few techniques to mitigate the impact of a small number of observed failures.

```
xxxx = pd.DataFrame(pd_data.groupby(['EQUIPMENT_FAILURE'])
['ID'].agg('count'))
xxxx
```


Out[17]:

EQUIPMENT_FAILURE	ID
0	307330
1	421

We can also explore the data with descriptive statistics.

```
pd_data.describe()
```

Out[18]:

	ID	WELL_GROUP	S15	S17	S13	S5	S16	S19	S18	EQUIPMENT_FAILURE	S8	AGE_OF_EQUIPMENT
count	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000	307751.000000
mean	100310.829609	4.643943	14.586192	60.265541	35.018249	4675.848232	7.972097	9.099123	137.963064	0.001368	144.666715	2524.192399
std	177.574390	2.284121	5.817056	66.804273	14.446586	2521.074632	2.321949	16.898887	235.890128	0.036961	240.779926	3158.930976
min	100001.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	-16.480000	0.000000
25%	100161.000000	3.000000	7.694100	0.000000	28.200000	3209.000000	6.621600	0.900000	11.798276	0.000000	9.250000	721.000000
50%	100311.000000	5.000000	11.881600	31.680000	34.940000	4237.047819	8.004000	4.200000	35.200000	0.000000	53.080000	1113.000000
75%	100467.000000	6.000000	22.560000	160.080000	41.610000	5743.000000	9.460000	10.600000	163.900000	0.000000	165.082908	2784.000000
max	100617.000000	8.000000	69.040000	2656.620000	592.890000	52767.000000	24.600000	511.000000	4161.700000	1.000000	2068.110000	16170.000000

3.0 Data transformations and Feature Engineering

Next, we can transform our data for a machine learning model. Specifically, we will create running summaries of the sensor values. Running summaries of sensor values are often useful in predicting equipment failure. For example, if a temperature gauge indicates a machine is warmer than average for the last five days, it may mean something is wrong.

Remember that we are working with a panel data set. That is, we have multiple machines measured over two years. As we create our running summaries, we have to make sure that our summaries do not include more than one machine. For example, if we create a ten-

Note that I create twenty-one-day summaries in this example. This works for this use case, but it may be advantageous to use more or different time intervals for other situations.

Convert dates from character to date.

```
pd_data['DATE'] = pd.to_datetime(pd_data['DATE'])
```

Create a new field called “flipper” that indicates when the id changes as the data are sorted by ID and DATE in ascending order. We will use this in a few other transformations.

```
pd_data=pd_data.sort_values(by=['ID','DATE'], ascending=[True, True])

pd_data['flipper'] = np.where((pd_data.ID != pd_data.ID.shift(1)), 1,
0)
pd_data.head()
```

Out[22]:

	ID	DATE	REGION	CLUSTER	MAINTENANCE_VENDOR	MANUFACTURER	WELL_GROUP	S15	S17	S13	S5	S16	S19	S18	EQUIPMENT_FAILURE	S9	AGE_OF_EQUIPMENT	flipper
0	100001	2014-12-02	G		D	Y	1	11.000000	145.223445	39.34	3501.0	8.426858	1.9	24.610345	0	0.0	880	1
1	100001	2014-12-03	G		D	Y	1	8.877943	187.573214	39.20	3480.0	8.483714	1.9	24.671429	0	0.0	881	0
2	100001	2014-12-04	G		D	Y	1	8.679444	148.983704	38.87	3458.0	8.159659	2.0	24.733333	0	0.0	882	0
3	100001	2014-12-05	G		D	Y	1	8.666038	133.000000	38.47	3513.0	8.320338	2.0	24.773077	0	0.0	883	0
4	100001	2014-12-06	G		D	Y	1	8.475284	197.181600	40.33	3558.0	8.022980	1.5	24.808000	0	0.0	884	0

Running summaries are often useful transformations for these types of problems. For example, a running mean would be the average value over the last x days. X, in this case, is the feature window. The feature window is a parameter that depends on the context of the business problem. I am setting the value to 21 days, but this may or may not work for your business problem.

```
#define your feature window. This is the window by which we will
aggregate our sensor values.
feature_window=21
```

"too_soon" is equal to 1, we have less than 21 days (feature_window) of history for the machine.

We will use these new variables to create a running mean, median, max, and min.

```
dfx=pd_data
#Select the first record of each machine

starter=dfx[dfx['flipper'] == 1]

starter=starter[['DATE','ID']]

#rename date to start_date
starter=starter.rename(index=str, columns={"DATE": "START_DATE"})

#convert START_DATE to date
starter['START_DATE'] = pd.to_datetime(starter['START_DATE'])

#Merge START_DATE to the original data set

dfx=dfx.sort_values(by=['ID', 'DATE'], ascending=[True, True])
starter=starter.sort_values(by=['ID'], ascending=[True])
dfx =dfx.merge(starter, on=['ID'], how='left')

# calculate the number of days since the beginning of each well.
dfx['C'] = dfx['DATE'] - dfx['START_DATE']
dfx['TIME_SINCE_START'] = dfx['C'] / np.timedelta64(1, 'D')
dfx=dfx.drop(columns=['C'])
dfx['too_soon'] = np.where((dfx.TIME_SINCE_START < feature_window) ,
1, 0)
```

Create a running mean, max, min, and median for the sensor variables.

```
dfx['S5_mean'] = np.where((dfx.too_soon == 0),
(dfx['S5'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S5)
dfx['S5_median'] = np.where((dfx.too_soon == 0),
(dfx['S5'].rolling(min_periods=1, window=feature_window).median()) ,
```


```
(dfx['S5'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S5)

dfx['S13_mean'] = np.where((dfx.too_soon == 0),
(dfx['S13'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S13)
dfx['S13_median'] = np.where((dfx.too_soon == 0),
(dfx['S13'].rolling(min_periods=1, window=feature_window).median()) ,
dfx.S13)
dfx['S13_max'] = np.where((dfx.too_soon == 0),
(dfx['S13'].rolling(min_periods=1, window=feature_window).max()) ,
dfx.S13)
dfx['S13_min'] = np.where((dfx.too_soon == 0),
(dfx['S13'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S13)

dfx['S15_mean'] = np.where((
(dfx['S15'].rolling(min_per
dfx.S15)
dfx['S15_median'] = np.where((dfx.too_soon == 0),
(dfx['S15'].rolling(min_periods=1, window=feature_window).median()) ,
dfx.S15)
dfx['S15_max'] = np.where((dfx.too_soon == 0),
(dfx['S15'].rolling(min_periods=1, window=feature_window).max()) ,
dfx.S15)
dfx['S15_min'] = np.where((dfx.too_soon == 0),
(dfx['S15'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S15)

dfx['S16_mean'] = np.where((dfx.too_soon == 0),
(dfx['S16'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S16)
dfx['S16_median'] = np.where((dfx.too_soon == 0),
(dfx['S16'].rolling(min_periods=1, window=feature_window).median()) ,
dfx.S16)
dfx['S16_max'] = np.where((dfx.too_soon == 0),
(dfx['S16'].rolling(min_periods=1, window=feature_window).max()) ,
dfx.S16)
dfx['S16_min'] = np.where((dfx.too_soon == 0),
(dfx['S16'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S16)

dfx['S17_mean'] = np.where((dfx.too_soon == 0),
(dfx['S17'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S17)
dfx['S17_median'] = np.where((dfx.too_soon == 0)
```



```
dfx['S17_min'] = np.where((dfx.too_soon == 0),
(dfx['S17'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S17)

dfx['S18_mean'] = np.where((dfx.too_soon == 0),
(dfx['S18'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S18)
dfx['S18_median'] = np.where((dfx.too_soon == 0),
(dfx['S18'].rolling(min_periods=1, window=feature_window).median()) ,
dfx.S18)
dfx['S18_max'] = np.where((dfx.too_soon == 0),
(dfx['S18'].rolling(min_periods=1, window=feature_window).max()) ,
dfx.S18)
dfx['S18_min'] = np.where((dfx.too_soon == 0),
(dfx['S18'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S18)

dfx['S19_mean'] = np.where((dfx.too_soon == 0),
(dfx['S19'].rolling(min_periods=1, window=feature_window).mean()) ,
dfx.S19)
dfx['S19_median'] = np.where((dfx.too_soon == 0),
(dfx['S19'].rolling(min_periods=1, window=feature_window).median()) ,
dfx.S19)
dfx['S19_max'] = np.where((dfx.too_soon == 0),
(dfx['S19'].rolling(min_periods=1, window=feature_window).max()) ,
dfx.S19)
dfx['S19_min'] = np.where((dfx.too_soon == 0),
(dfx['S19'].rolling(min_periods=1, window=feature_window).min()) ,
dfx.S19)
```

Another useful transformation is to look for sudden spikes in sensor values. This code creates a value indicating how far the current value is from the immediate norm.

```
dfx['S5_chg'] = np.where((dfx.S5_mean == 0),0 , dfx.S5/dfx.S5_mean)

dfx['S13_chg'] = np.where((dfx.S13_mean == 0),0 ,
dfx.S13/dfx.S13_mean)

dfx['S15_chg'] = np.where((dfx.S15_mean==0),0 , dfx.S15/dfx.S15_mean)
dfx['S16_chg'] = np.where((dfx.S16_mean == 0),0 ,
dfx.S16/dfx.S16_mean)
dfx['S17_chg'] = np.where((dfx.S17_mean == 0),0
```


```
#copy the data set to the original name
pd_data=dfx
```

For more information on feature engineering for predictive maintenance problems, please see the following.

4.0 Dealing with the small number of failures.

4.1 Expand the Failure (Target) Window

Machines are engineered to last. If something breaks all the time, you won't buy it, would you?

Because machines generally last a long time, we typically do not have many examples of failure. This means the data sets we use in PM are almost always unbalanced.

One way to increase the number of failures is to expand the failure or target window. That is, make the dependent variable, not just the day the equipment failed but the 28 days (or another appropriate interval) leading up to the failure.

In this example, I use a 28-day target window. We will use the 28 days leading up to a failure as the dependent variable in our model.

```
target_window=28
```


```
pd_data=pd_data.sort_values(by=['ID', 'DATE'], ascending=[True, True])
pd_data.reset_index(level=0, inplace=True)
```

Create a new data frame that contains the failure records. Rename DATE to FAILURE_DATE.

```
df_failure_thingy=pd_data[pd_data['EQUIPMENT_FAILURE'] == 1]

df_failure_thingy=df_failure_thingy[['DATE','ID']]

df_failure_thingy=df_failure_thingy.rename(index=str, columns={"DATE":
"FAILURE_DATE"})

pd_data=pd_data.sort_values(by=['ID'], ascending=[True])
df_failure_thingy=df_failure_thingy.sort_values(by=['ID'], ascending=
[True])
```

Append the FAILURE_DATE to each ID.

```
pd_data =pd_data.merge(df_failure_thingy, on=['ID'], how='left')
```

For each record, calculate the number of days until failure.

```
pd_data=pd_data.sort_values(by=['ID','DATE'], ascending=[True, True])

pd_data['FAILURE_DATE'] = pd.to_datetime(pd_data['FAILURE_DATE'])
pd_data['DATE'] = pd.to_datetime(pd_data['DATE'])
pd_data['C'] = pd_data['FAILURE_DATE'] - pd_data['DATE']

pd_data['TIME_TO_FAILURE'] = pd_data['C'] / np.timedelta64(1, 'D')
```

Change order to the original by ID and DATE


```
pd_data=pd_data.sort_values(by=['ID', 'DATE'], ascending=[True, True])

pd_data.reset_index(inplace=True)
```

Create a new variable, FAILURE_TARGET. It is equal to 1 if the record proceeds a failure by “failure_window” days or less.

```
pd_data['FAILURE_TARGET'] = np.where(((pd_data.TIME_TO_FAILURE <
target_window) & ((pd_data.TIME_TO_FAILURE>=0))), 1, 0)

tips_summed = pd_data.groupby(['FAILURE_TARGET'])['S5'].count()
tips_summed
```

```
Out[43]: FAILURE_TARGET
         0    296011
         1     11740
         Name: S5, dtype: int64
```

The new field occurs about 4% of the time.

```
pd_data['FAILURE_TARGET'].mean()
```

```
Out[44]: 0.03814772332177637
```

Now we have 11,740 target observations. This is better, but the data set is far from


4.2 Create the Testing, Training and Validation Groupings

Because we are dealing with a panel data set (cross-sectional time-series), it is better not to take a random sample of all records. Doing so would put the records from one machine in all three sample data sets. To avoid this, we'll randomly select IDs and place all of the records for each machine in either the training, testing, or validation data set.

```
#Get a Unique List of All IDs

aa=pd_data

pd_id=aa.drop_duplicates(subset='ID')
pd_id=pd_id[['ID']]
pd_id.shape
```

```
Out[45]: (421, 1)
```

Create a new variable with a random number between 0 and 1.

```
np.random.seed(42)

pd_id['wookie'] = (np.random.randint(0, 10000, pd_id.shape[0]))/10000

pd_id=pd_id[['ID', 'wookie']]
```

Give each record a 30% chance of being in the validation, a 35% chance of being in the testing, and a 35% chance of being in the training data set.

```
pd_id['MODELING_GROUP'] = np.where(((pd_id.wookie <= 0.35)),
'TRAINING', np.where(((pd_id.wookie <= 0.65)), 'VALIDATION',
'TESTING'))
```


```
tips_summed = pd_id.groupby(['MODELING_GROUP'])['wookie'].count()
tips_summed
```

```
Out[50]: MODELING_GROUP
         TESTING        149
         TRAINING       146
         VALIDATION     126
         Name: wookie, dtype: int64
```

Append the Group of each id to each individual record.

```
pd_data=pd_data.sort_values(by=['ID'], ascending=[True])
pd_id=pd_id.sort_values(by=['ID'], ascending=[True])

pd_data =pd_data.merge(pd_id, on=['ID'], how='inner')
```

This is how many records are in each group.

```
tips_summed = pd_data.groupby(['MODELING_GROUP'])['wookie'].count()
tips_summed
```

```
Out[53]: MODELING_GROUP
         TESTING        108919
         TRAINING       106726
         VALIDATION      92106
```

This is how many failure targets are in each group.

```
tips_summed = pd_data.groupby(['MODELING_GROUP'])
['FAILURE_TARGET'].sum()
tips_summed
```

```
Out[54]: MODELING_GROUP
         TESTING        4151
         TRAINING       4071
         VALIDATION     3518
         Name: FAILURE_TARGET, dtype: int64
```

Create a separate data frame for the training data. We will use this data set to build the model.

```
df_training=pd_data[pd_data['MODELING_GROUP'] == 'TRAINING']
df_training=df_training.drop(columns=
['MODELING_GROUP','C','wookie','TIME_TO_FAILURE','flipper','START_DATE
'])
```

Create a separate data frame for the training and testing data sets. We will use this to tweak our modeling results.

```
df_train_test=pd_data[pd_data['MODELING_GROUP'] != 'VALIDATION']

df_train_test=df_train_test.drop(columns=
['wookie','TIME_TO_FAILURE','flipper','START_DATE'])
df_train_test.shape
```


```
df_total=pd_data.drop(columns=
['C','wookie','TIME_TO_FAILURE','flipper','START_DATE'])
```

4.3 SMOTE the Training Data

Note that we are only balancing the training data set. You may be asking why. Remember that our goal is to build a model the represents reality, right? When we SMOTE the data, we change the failure rate to 50%. This is nowhere near what we see in the actual machine data. Thus, it makes sense to build the model on the SMOTE data but evaluate it on the unaltered data. The unaltered data will be a better reflection of what to expect when you deploy the model to production.

Define the Training features and Target.

```
training_features=df_training[['REGION_CLUSTER','MAINTENANCE_VENDOR','
MANUFACTURER','WELL_GROUP','AGE_OF_EQUIPMENT','S15','S17','S13','S5',

'S16','S19','S18','S8','S5_mean','S5_median','S5_max','S5_min','S13_me
an','S13_median','S13_max','S13_min','S15_mean','S15_median',

'S15_max','S15_min','S16_mean','S16_median','S16_max','S16_min','S17_m
ean','S17_median','S17_max','S17_min','S18_mean','S18_median','S18_max
','S18_min','S19_mean','S19_median','S19_max','S19_min',

'S5_chg','S13_chg','S15_chg','S16_chg','S17_chg','S18_chg','S19_chg']]

training_target=df_training[['FAILURE_TARGET']]
```

Synthetically Balance the training data sets with a SMOTE algorithm. After we apply the SMOTE algorithm, we will have a balanced data set. 50% Failures and 50% Non-Failures. Note that this takes a while to run.

```
from imblearn.over_sampling import SMOTE
from imblearn.over_sampling import SMOTENC
smx = SMOTENC(random_state=12,  categorical_features=[0, 1, 2, 3])
```


Convert the SMOTE output back to complete data frames with independent and dependent variables. Examine the results.

Format the Independent Variables.

```
df_x=pd.DataFrame(x_res)

df_x.columns = [

'REGION_CLUSTER','MAINTENANCE_VENDOR','MANUFACTURER','WELL_GROUP','AGE
_OF_EQUIPMENT','S15','S17','S13','S5','S16','S19',

'S18','S8','S5_mean','S5_median','S5_max','S5_min','S13_mean','S13_med
ian','S13_max','S13_min','S15_mean','S15_median','S15_max',

'S15_min','S16_mean','S16_median','S16_max','S16_min','S17_mean','S17_
median','S17_max','S17_min','S18_mean','S18_median','S18_max','S18_min
',

'S19_mean','S19_median','S19_max','S19_min','S5_chg','S13_chg','S15_ch
g','S16_chg','S17_chg','S18_chg','S19_chg']
```

Format the Dependent Variable.

```
df_y=pd.DataFrame(y_res)
df_y.columns = ['FAILURE_TARGET']
```

Check that the dependent variable is balanced. It is.

```
df_y.mean(axis = 0)
```

```
Out[65]: FAILURE_TARGET    0.5
```


Merge the dependent and independent variables post SMOTE into a data frame.

```
df_balanced = pd.concat([df_y, df_x], axis=1)
```

5.0 More data transformation and feature engineering

Convert the categorical variables into binary dummy variables. We need to do this because the XGBT model (below) doesn't like categorical fields.

```
df_dv = pd.get_dummies(df_balanced['REGION_CLUSTER'])

df_dv=df_dv.rename(columns={"A":
"CLUSTER_A","B":"CLUSTER_B","C":"CLUSTER_C","D":"CLUSTER_D","E":"CLUST
ER_E","F":"CLUSTER_F","G":"CLUSTER_G","H":"CLUSTER_H"})

df_balanced= pd.concat([df_balanced, df_dv], axis=1)

df_dv = pd.get_dummies(df_balanced['MAINTENANCE_VENDOR'])

df_dv=df_dv.rename(columns={"I":
"MV_I","J":"MV_J","K":"MV_K","L":"MV_L","M":"MV_M","N":"MV_N","O":"MV_
O","P":"MV_P"})

df_balanced = pd.concat([df_balanced, df_dv], axis=1)

df_dv = pd.get_dummies(df_balanced['MANUFACTURER'])

df_dv=df_dv.rename(columns={"Q":
"MN_Q","R":"MN_R","S":"MN_S","T":"MN_T","U":"MN_U","V":"MN_V","W":"MN_
W","X":"MN_X","Y":"MN_Y","Z":"MN_Z"})

df_balanced = pd.concat([df_balanced, df_dv], axis=1)

df_dv = pd.get_dummies(df_balanced['WELL_GROUP'])

df_dv=df_dv.rename(columns={1:
"WG_1",2:"WG_2",3:"WG_3",4:"WG_4",5:"WG_5",6:"WG_6",7:"WG_7",8:"WG_8"}
)
```


```
df_dv = pd.get_dummies(df_train_test['REGION_CLUSTER'])

df_dv=df_dv.rename(columns={"A":
"CLUSTER_A","B":"CLUSTER_B","C":"CLUSTER_C","D":"CLUSTER_D","E":"CLUST
ER_E","F":"CLUSTER_F","G":"CLUSTER_G","H":"CLUSTER_H"})

df_train_test= pd.concat([df_train_test, df_dv], axis=1)

df_dv = pd.get_dummies(df_train_test['MAINTENANCE_VENDOR'])

df_dv=df_dv.rename(columns={"I":
"MV_I","J":"MV_J","K":"MV_K","L":"MV_L","M":"MV_M","N":"MV_N","O":"MV_
O","P":"MV_P"})

df_train_test = pd.concat([df_train_test, df_dv], axis=1)

df_dv = pd.get_dummies(df_train_test['MANUFACTURER'])

df_dv=df_dv.rename(columns={"Q":
"MN_Q","R":"MN_R","S":"MN_S","T":"MN_T","U":"MN_U","V":"MN_V","W":"MN_
W","X":"MN_X","Y":"MN_Y","Z":"MN_Z"})

df_train_test = pd.concat([df_train_test, df_dv], axis=1)

df_dv = pd.get_dummies(df_train_test['WELL_GROUP'])

df_dv=df_dv.rename(columns={1:
"WG_1",2:"WG_2",3:"WG_3",4:"WG_4",5:"WG_5",6:"WG_6",7:"WG_7",8:"WG_8"}
)

df_train_test = pd.concat([df_train_test, df_dv], axis=1)
```

And, also on the df_total data set.

```
df_dv = pd.get_dummies(df_total['REGION_CLUSTER'])

df_dv=df_dv.rename(columns={"A":
"CLUSTER_A","B":"CLUSTER_B","C":"CLUSTER_C","D":"CLUSTER_D","E":"CLUST
ER_E","F":"CLUSTER_F","G":"CLUSTER_G","H":"CLUSTER_H"})
```


```
df_dv=df_dv.rename(columns={"I":
"MV_I","J":"MV_J","K":"MV_K","L":"MV_L","M":"MV_M","N":"MV_N","O":"MV_
O","P":"MV_P"})

df_total = pd.concat([df_total, df_dv], axis=1)

df_dv = pd.get_dummies(df_total['MANUFACTURER'])

df_dv=df_dv.rename(columns={"Q":
"MN_Q","R":"MN_R","S":"MN_S","T":"MN_T","U":"MN_U","V":"MN_V","W":"MN_
W","X":"MN_X","Y":"MN_Y","Z":"MN_Z"})

df_total = pd.concat([df_total, df_dv], axis=1)

df_dv = pd.get_dummies(df_total['WELL_GROUP'])

df_dv=df_dv.rename(columns={1:
"WG_1",2:"WG_2",3:"WG_3",4:"WG_4",5:"WG_5",6:"WG_6",7:"WG_7",8:"WG_8"}
)

df_total = pd.concat([df_total, df_dv], axis=1)
```

6.0 Build the model on the balanced training data set

Remove the newly redundant categorical variables. This are now represented by dummy variables.

```
df_balanced=df_balanced.drop(columns=
['REGION_CLUSTER','MAINTENANCE_VENDOR','MANUFACTURER','WELL_GROUP'])
```

In the balanced data set, separate the dependent and independent variables to feed the model development process.

```
features = [x for x in df_balanced.columns if x not in
['FAILURE_TARGET','EQUIPMENT_FAILURE']]
dependent=pd.DataFrame(df_balanced['FAILURE_TARGET'])
```


```
df_balanced = df_balanced.apply(pd.to_numeric)
```

Define model specs.

```
import matplotlib.pylab as plt
%matplotlib inline

def evaluate_model(alg, train, target, predictors,
early_stopping_rounds=1):


    #Fit the algorithm on the data
    alg.fit(train[predictors], target['FAILURE_TARGET'],
eval_metric='auc')

    #Predict training set:
    dtrain_predictions = alg.predict(train[predictors])
    dtrain_predprob = alg.predict_proba(train[predictors])[:,1]

    feat_imp =
pd.Series(alg.get_booster().get_fscore()).sort_values(ascending=False)
    feat_imp.plot(kind='bar', title='Feature Importance', color='g')
    plt.ylabel('Feature Importance Score')

    #Print model report:
    print("\nModel Report")
    print("Accuracy : %.4g" %
metrics.accuracy_score(target['FAILURE_TARGET'].values,
dtrain_predictions))
    print("AUC Score (Balanced): %f" %
metrics.roc_auc_score(target['FAILURE_TARGET'], dtrain_predprob))
```

We are initializing our model with default model parameters. Note that we could probably improve the results by tweaking the parameters, but we will save that exercise for another day.

```
xgb0 = XGBClassifier(
 objective= 'binary:logistic')
```


Probably the most confusing element of PM problems is building a realistic assessment of the model. Because of timing and the small number of failures, understanding how the model will work once deployed in production is challenging.

There are standard metrics for evaluating models like accuracy, AUC, and a confusion matrix. In sections 7.1 and 7.2, I will show how, given the transformations we used to build our model and the complexity of the problem, these metrics do not give us a realistic view of model performance when deployed into production. These standard metrics are definitely useful but are not sufficient.

In section 7.3, I lay out how I typically evaluate PM models.

7.1 Evaluate the model using an AUC and accuacy metrics.

First, we will evaluate the balanced training data with the default, a 50% cut-off.

For information on how to find the best cut-off for these types of problems, please see the following.

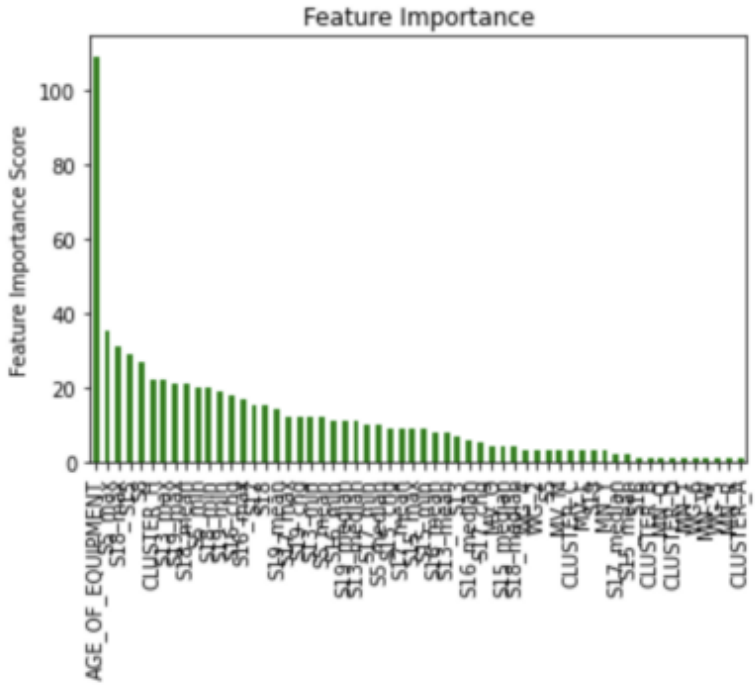
Determining a Cut-Off or Threshold When Working With a Binary Dependent (Target) Variable.

Where is the best place to draw the line?

medium.com

```
evaluate_model(xgb0, independent, dependent,features)
```


```
Model Report
Accuracy : 0.8346
AUC Score (Balanced): 0.915649
```



With a cut-off of 50%, the accuracy is .835, and the AUC is .916 on the balanced training data.

For more information on how to determine the best cut-off for these types of problems, please see the following article.

https://medium.com/swlh/determining-a-cut-off-or-threshold-when-working-with-a-binary-dependent-target-variable-7c2342cf2a7c

These results are not bad, but there are a few issues. Balancing the data significantly alters your base probability of failure. Remember, before we applied the SMOTE algorithm, about 3.8% of our records were labeled as a FAILURE_TARGET. After using the SMOTE algorithm, 50% were failures. Because of this, I always prefer to evaluate the model on the unbalanced data. Below we will examine the unbalanced data with a 50% cut-off.

Also, note that the predicted probability from our model is based on the balanced data.

I am using the term “probability,” perhaps a little too loosely in this article. Just understand that the probability that the equipment will fail in the real world is about .14% (based on EQUIPMENT_FAILURE). The likelihood of selecting a day within 28 days of a failure (FAILURE_TARGET) is about 3.8%. The probability of a failure after we balance the data is 50%.

Isolate the unbalanced training and testing data sets.

```
df_testing=df_train_test[df_train_test['MODELING_GROUP'] ==
'TESTING'].copy()
df_training=df_train_test[df_train_test['MODELING_GROUP'] !=
'TESTING'].copy()
```

Score the unbalance3 training data with a 50% cut-off.

```
df_training[‘P_FAIL’]= xgb0.predict_proba(df_training[features])[:,1];
df_training[‘Y_FAIL’] = np.where(((df_training.P_FAIL <= .50)), 0, 1)
#Print model report:
print(“Accuracy : %.4g” %
metrics.accuracy_score(df_training[‘FAILURE_TARGET’].values,
df_training[‘Y_FAIL’]))
print(“AUC Score (Train): %f” %
metrics.roc_auc_score(df_training[‘FAILURE_TARGET’],
df_training[‘P_FAIL’]))
```

```
Accuracy : 0.7863
AUC Score (Train): 0.891580
```

With a 50% cut-off, we get an accuracy of .79 and an AUC of .89

Now, lets’ try with a 67% cut-off. Probabilities larger than 67% are labeled as failures.

```
df_training['P_FAIL']= xgb0.predict_proba(df_training[features])[:,1];
df_training['Y_FAIL'] = np.where(((df_training.P_FAIL <= .67)), 0, 1)
#Print model report:
print("Accuracy : %.4g" %
metrics.accuracy_score(df_training['FAILURE_TARGET'].values,
df_training['Y_FAIL']))
print("AUC Score (Train): %f" %
metrics.roc_auc_score(df_training['FAILURE_TARGET'],
df_training['P_FAIL']))
```

```
Accuracy : 0.9504
AUC Score (Train): 0.891580
```

Now, let's apply the model to the testing data set.

```
df_testing['P_FAIL']= xgb0.predict_proba(df_testing[features])[:,1];
df_testing['Y_FAIL'] = np.where(((df_testing.P_FAIL <= .67)), 0, 1)
#Print model report:
print("Accuracy : %.4g" %
metrics.accuracy_score(df_testing['FAILURE_TARGET'].values,
df_testing['Y_FAIL']))
print("AUC Score (Test): %f" %
metrics.roc_auc_score(df_testing['FAILURE_TARGET'],
df_testing['P_FAIL']))
```

```
Accuracy : 0.9415
AUC Score (Test): 0.588301
```

Before we go any further, let's take a step back and think about what we have done. Ok, so our model validates with an accuracy of .942.

I don't think we can know if this is good or bad. To fully grasp how a model will perform, you have to put it into the context of its deployment. In the next section, I will attempt, but fail, to do this with a confusion matrix.

7.2 Evaluating with a Confusion Matrix.

Previously, we looked at the AUC and accuracy as ways to evaluate a predictive maintenance model. I don't think they cut the mustard, honestly. A confusion matrix is better because it directly correlates to the cost structure of the problem.

A confusion matrix specifically lays out the following metrics.

1. True Positive. The model predicts the machine will fail, and it does.
2. True Negative. The model predicts the machine will not fail and it does not fail.
3. False Positive. The model predicts the machine will fail and it does not fail.
4. False Negative. The model predicts the machine will not fail and it fails.

These metrics directly correlate to the economic costs of our PM problem.

Remember this?

Maintenance Scenario	Number of Incidents	Cost per Incident	Total Cost
Unnecessary Maintenance	9	$ 1,500	$ 13,500
Timely and Appropriate Maintenance	27	$ 7,500	$ 202,500
Machine Runs to Failure	385	$ 30,000	$ 11,550,000
Total	421	$ 27,948	$ 11,766,000

A false positive is "Unnecessary Maintenance." When your model predicts failure and is not going to fail, you incur unnecessary maintenance costs.

A true positive is "Timely and Appropriate Maintenance." When your model predicts failure, and it is going to fail, you incur timely and appropriate maintenance costs.

A false negative costs $30,000.

A true positive costs $7,500.

A true negative has no cost because no action is taken.

A confusion matrix lays out what is important for us to evaluate.

A confusion matrix is a cross-tabulation between the binary prediction of failure/non-failure and the actual binary failure/non-failure variable. Remember that we have two actual failure values.

The first is the original failure variable, 'EQUIPMENT_FAILURE.'

```
xxxx = pd.DataFrame(pd_data.groupby(['EQUIPMENT_FAILURE'])
['ID'].agg('count'))
xxxx
```

Out[81]:

EQUIPMENT_FAILURE	ID
0	307330
1	421

'EQUIPMENT_FAILURE' has 421 failure indicators. Remember that a '1' appears on the day that the failure occurs.

```
print(pd.crosstab(df_testing.Y_FAIL, df_testing.EQUIPMENT_FAILURE,
dropna=False))
```

Y_FAIL \ EQUIPMENT_FAILURE	0	1
0	105977	108
1	2793	41

According to the confusion matrix above, we have 41 true positives, 108 false negatives, 2793 false positives, and 105,977 true negatives.

Is this right?

No, it isn't. Think about it.

The 'EQUIPMENT_FAILURE' variable identifies a failure on the day it occurs. Let's say that failure occurs on Friday, and we have signals for failure on Monday, Tuesday, Wednesday, and Thursday, but not on Friday. I would argue that the failure on Friday is a True Positive, given that there were four failure signals in the days leading up to the failure. However, the confusion matrix above counts Monday, Tuesday, Wednesday, and Thursday as false positives and Friday as a false negative. This doesn't work, does it?

Now, we will create a confusion matrix using 'FAILURE_TARGET.' This is the predicted binary variable where the 28 days leading up to a failure are flagged with a '1'.

```
print(pd.crosstab(df_testing.Y_FAIL, df_testing.FAILURE_TARGET,
dropna=False))
```

FAILURE_TARGET	0	1
Y_FAIL		
0	102240	3845
1	2528	306

With 'FAILURE_TARGET,' we have 306 True Positives, 3,845 False Negatives, 2,528 False Positives, and 102,240 True Negatives.

Does this work?

Not really, huh. Think about it.

Each failure is represented 28 different times. If a machine fails on Friday the 28th, the previous twenty-seven days of the month are also flagged as a failure.

Friday can be a true positive, the previous Wednesday a true positive, and the previous Thursday a false positive.

The example above clearly does not reflect what we can expect when we push this model into production.

So, what does all this mean? To get an accurate accounting of how the model will work in production, we use additional logic and business rules/heuristics.

7.3 Using Heuristics to Define a False Positive, True Positive, False Negative, and True Negative.

To accurately evaluate our machine learning model, we first need to define a few parameters. I usually fine-tune these parameters with the testing and training data sets, then confirm the accuracy with the validation data set.

The first parameter is the Forecast Window.

Does it mean the machine will break in the next second, minute, hour, day, or month?

Note that the length of the forecast window depends on the context of the problem. If a machine runs to failure in 10 years, the forecast window will be relatively long, perhaps as long as six or nine months. If a machine runs to failure in 1 day, the forecast window will be much shorter, maybe an hour or a few minutes. The length of the forecast window must be helpful to the problem. For example, a twenty-eight-day forecast window for a machine that runs to failure in thirty days is of little value.

In our current use case, the machine runs to failure in four to six years, so a ninety-day forecast window is reasonable.

```
forecast_window=90
```

Now that we defined the forecast window we will now apply it to the data. This means that a signal is good for ninety days. For example, if a signal appears on January 1st, that signal is good until March 31st (ninety days)

We will create a new failure indicator.

This failure indicator (signal) can only appear every ninety days. Note that we have panel data and must ensure that the signals don't "bleed" from one machine to the next. For example, if a signal occurs on machine X's last possible day, we do not want the window for machine X+1 to be affected by this signal.

The following steps will score df_train_test, create a unique list of machines, and create a sequential id for each machine. After completing these tasks, I set up parameters to loop through each machine.

```
#Score df_train_test
```


```
#sort the data by id and date.
xx=df_train_test
xx=xx.sort_values(by=['ID','DATE'], ascending=[True, True])

#create a unique list of machines
aa=xx

pd_id=aa.drop_duplicates(subset='ID')
pd_id=pd_id[['ID']]

#label each machine with a sequential number
pd_id=pd_id.reset_index(drop=True)
pd_id=pd_id.reset_index(drop=False)
pd_id=pd_id.rename(columns={"index": "SCOOBYDOO"})
pd_id['SCOOBYDOO']=pd_id['SCOOBYDOO']+1
```

Out[88]:

	SCOOBYDOO	ID
0	1	100001
1	2	100002
2	3	100014
3	4	100017
4	5	100018

```
#grab the max number of machines +1

column = pd_id["SCOOBYDOO"]
max_value = column.max()+1
```

```
xx =xx.merge(pd_id, on=['ID'], how='inner')
xx.head()
```

Out[90]:

	index	ID	DATE	REGION	CLUSTER	MAINTENANCE VENDOR	MANUFACTURER	WELL GROUP	S16	S17	S13	..	WG_2	WG_3	WG_4	WG_5	WG_6	WG_7	WG_8	P_FAIL	Y_FAIL	SCOOBYDOO
0	0	100031	2014-12-02		G	O	Y	1	11.000	145.223448	39.34	..	0	0	0	0	3	0	0	0.349602	0	1
1	1	100031	2015-03-29		G	O	Y	1	18.060	0.000000	38.87	..	0	0	0	0	3	0	0	0.644506	0	1
2	2	100031	2015-03-30		G	O	Y	1	29.040	0.000000	37.36	..	0	0	0	0	3	0	0	0.449370	0	1
3	3	100031	2015-03-31		G	O	Y	1	18.000	0.000000	38.81	..	0	0	0	0	3	0	0	0.644506	0	1
4	4	100031	2015-04-01		G	O	Y	1	26.160	0.000000	39.47	..	0	0	0	0	3	0	0	0.616145	0	1

```
#sort data
xx=xx.sort_values(by=['ID','DATE'], ascending=[True,True])

#reset index
xx=xx.reset_index(drop=True)

#create a null dataframe for the next step
df_fred=xx
df_fred['Y_FAIL_sumxx']=0
df_fred=df_fred[df_fred['SCOOBYDOO'] == max_value+1]
```

The next few steps assign a new failure indicator that incorporates the forecast window. Note, this calculation occurs at a machine level. Doing this keeps a signal from one machine affecting another machine.

This takes a while to run.

```
#sum the number of signals occuring over the last 90 days for each
machine individually

for x in range(max_value):
        dffx=xx[xx['SCOOBYDOO'] ==x]
        dff=dffx.copy()
        dff['Y_FAIL_sumxx'] =(dff['Y_FAIL'].rolling(min_periods=1,
window=(forecast_window)).sum())
        df_fred= pd.concat([df_fred,dff])
```

Now that we have defined the failure window and used this definition to clean up the failure indicator, we now need to associate the failure indicators or signals with the actual failures to determine prediction accuracy.

In the next few steps, we will create a unique id for each failure signal, the machine (ID) associated with each signal, and each signal's date.

```
#sort the data by id and date.

xx=xx.sort_values(by=['ID','DATE'], ascending=[True, True])

#create signal id with the cumsum function.
xx['SIGNAL_ID'] = xx['Y_FAILZ'].cumsum()
```

Now we will pull the records with a signal into a different data frame.

Here we will create a new field that identifies the date of each signal (SIGNAL_DATE).

Also, we will identify the ID Associated with each signal (ID_OF_SIGNAL)

```
df_signals=xx[xx['Y_FAILZ'] == 1]
df_signal_date=df_signals[['SIGNAL_ID','DATE','ID']]
df_signal_date=df_signal_date.rename(index=str, columns={"DATE":
"SIGNAL_DATE"})
df_signal_date=df_signal_date.rename(index=str, columns={"ID":
"ID_OF_SIGNAL"})
```

We have a total of 536 signals. Now each has a unique id.

```
df_signal_date.shape
```


Append SIGNAL_ID to the primary data frame.

```
xx =xx.merge(df_signal_date, on=['SIGNAL_ID'], how='outer')
```

Simplify by only keeping the fields we need going forward.

```
xx=xx[['DATE', 'ID', 'EQUIPMENT_FAILURE',
'FAILURE_TARGET','FAILURE_DATE',
       'P_FAIL', 'Y_FAILZ','SIGNAL_ID',
       'SIGNAL_DATE','ID_OF_SIGNAL','MODELING_GROUP']]
```

Create a field called “Warning” that indicates the time from signal to failure.

```
xx['C'] = xx['FAILURE_DATE'] - xx['SIGNAL_DATE']
xx['WARNING'] = xx['C'] / np.timedelta64(1, 'D')
```

Finally, we have enough information to define a false positive, false negative, true positive, and true negative.

My definition makes sense here but is unique to this specific business problem.

A true positive occurs if and only if the machine fails, and there was a signal within the last 90 days. Also, we have to ensure that the signal id belongs to the machine (ID). Note that this prohibits a signal from another machine from being applied to the machine in question.

A false negative occurs if and only if the machine fails, and it is not a true positive.

A False Positive occurs if there is a failure signal, and a failure does not happen in the next 90 days. Also, if a signal occurs after the failure, this is a false positive. We also have to


If an observation is not a false positive, a false negative, or a true positive, it is a true negative.

```
# define a true positive
xx['TRUE_POSITIVE'] = np.where(((xx.EQUIPMENT_FAILURE == 1) &
(xx.WARNING<=forecast_window) &(xx.WARNING>=0) &
(xx.ID_OF_SIGNAL==xx.ID)), 1, 0)

# define a false negative
xx['FALSE_NEGATIVE'] = np.where((xx.TRUE_POSITIVE==0) &
(xx.EQUIPMENT_FAILURE==1), 1, 0)

# define a false positive
xx['BAD_S']=np.where((xx.WARNING<0) | (xx.WARNING>=forecast_window),
1, 0)

xx['FALSE_POSITIVE'] = np.where(((xx.Y_FAILZ == 1) & (xx.BAD_S==1) &
(xx.ID_OF_SIGNAL==xx.ID)), 1, 0)

xx['bootie']=1

xx['CATEGORY']=np.where((xx.FALSE_POSITIVE==1),'FALSE_POSITIVE',

(np.where((xx.FALSE_NEGATIVE==1),'FALSE_NEGATIVE',

(np.where((xx.TRUE_POSITIVE==1),'TRUE_POSITIVE','TRUE_NEGATIVE')))))

table = pd.pivot_table(xx, values=['bootie'], index=
['MODELING_GROUP'],columns=['CATEGORY'], aggfunc=np.sum)
table
```

Out[109]:

	bootie			
CATEGORY	FALSE_NEGATIVE	FALSE_POSITIVE	TRUE_NEGATIVE	TRUE_POSITIVE
MODELING_GROUP				
TESTING	98	131	108639	51
TRAINING	41	107	106473	105

Now we can apply the same logic to the validation data set to make sure it is not sample-specific.

7.4 Apply Model and Heuristics the Training, Testing and Validation Data Sets.

Predict the probability of failure for all records.

Create a predicted failure indicator based on a cut-off of .67.

```
df_total['P_FAIL']= xgb0.predict_proba(df_total[features])[:,1];
df_total['Y_FAIL'] = np.where(((df_total.P_FAIL <= .67)), 0, 1)
```

Define the forecast window.

```
forecast_window=90
```

Ensure that the failure indicator occurs only once every 90 days for each machine.

```
#get a the number of machines +1 and label each machine with a
sequential number.

aa=df_total

pd_id=aa.drop_duplicates(subset='ID')
pd_id=pd_id[['ID']]
pd_id=pd_id.reset_index(drop=True)
pd_id=pd_id.reset_index(drop=False)
pd_id=pd_id.rename(columns={"index": "SCOOBYDOO"})
pd_id['SCOOBYDOO']=pd_id['SCOOBYDOO']+1

column = pd_id["SCOOBYDOO"]
max_value = column.max()+1

yy=df_total
```

Out[114]:

	Index	ID	DATE	REGION_CLUSTER	MAINTENANCE_VENDOR	MANUFACTURER	WELL_GROUP	S15	S17	S13	...	W0_2	W0_3	W0_4	W0_5	W0_6	W0_7	W0_8	P_FAIL	Y_FAIL	SCOOBYDOO
0	0	100001	2014-12-03	G	O	Y	1	11.080000	145.220440	39.04	...	0	0	0	0	0	0	0	0.349692	0	1
1	485	100001	2014-12-04	G	O	Y	1	8.671444	148.563704	38.87	...	0	0	0	0	0	0	0	0.360758	0	1
2	484	100001	2014-12-05	G	O	Y	1	9.986538	130.000000	39.47	...	0	0	0	0	0	0	0	0.560861	0	1
3	486	100001	2014-12-06	G	O	Y	1	8.475264	107.161600	40.08	...	0	0	0	0	0	0	0	0.204767	0	1
4	488	100001	2014-12-07	G	O	Y	1	7.971160	164.545833	38.74	...	0	0	0	0	0	0	0	0.307750	0	1

5 rows × 94 columns

```
#sort data
yy=yy.sort_values(by=['ID','DATE'], ascending=[True,True])

#reset index
yy=yy.reset_index(drop=True)

#create a null dataframe for the next step
df_fred=yy
df_fred['Y_FAIL_sumxx']=0
df_fred=df_fred[df_fred['SCOOBYDOO'] == max_value+1]
```

The next few steps assign a new failure indicator that incorporates the forecast window. Note, this calculation occurs at a machine level. This keeps a signal from one machine affecting another machine. This takes a while to run.

```
#sum the number of signals occuring over the last 90 days for each
machine individually
for x in range(max_value):
        dffx=yy[yy['SCOOBYDOO'] ==x]
        dff=dffx.copy()
        dff['Y_FAIL_sumxx'] =(dff['Y_FAIL'].rolling(min_periods=1,
window=(forecast_window)).sum())
        df_fred= pd.concat([df_fred,dff])

yy=df_fred

# if a signal has occured in the last 90 days, the signal is 0.

yy['Y_FAILZ']=np.where((yy.Y_FAIL_sumxx>1), 0, yy.Y_FAIL)
```

In the next few steps, we will create a unique id for each failure signal, the machine (ID) associated with each signal, and each signal's date.

```
#sort the data by id and date.
yy=yy.sort_values(by=['ID','DATE'], ascending=[True, True])

#create signal id with the cumsum function.
yy['SIGNAL_ID'] = yy['Y_FAILZ'].cumsum()
```

Now we will pull the records with a signal into a different data frame. Here we will create a new field that identifies the date of each signal (SIGNAL_DATE). Also, we will identify the ID Associated with each signal (ID_OF_SIGNAL)

```
#create the signal date and ID_OF_SIGNAL

yy_signals=yy[yy['Y_FAILZ'] == 1]
yy_signal_date=yy_signals[['SIGNAL_ID','DATE','ID']]
yy_signal_date=yy_signal_date.rename(index=str, columns={"DATE":
"SIGNAL_DATE"})
yy_signal_date=yy_signal_date.rename(index=str, columns={"ID":
"ID_OF_SIGNAL"})

#merge the two data frames back into one.

yy =yy.merge(yy_signal_date, on=['SIGNAL_ID'], how='outer')

#Keep on the fields we need
yy=yy[['DATE', 'ID', 'EQUIPMENT_FAILURE',
'FAILURE_TARGET','FAILURE_DATE','MODELING_GROUP',
       'P_FAIL', 'Y_FAILZ','SIGNAL_ID',
       'SIGNAL_DATE','ID_OF_SIGNAL']]

# Calculate the warning time between each failure date and signal
date.
yy['C'] = yy['FAILURE_DATE'] - yy['SIGNAL_DATE']
yy['WARNING'] = yy['C'] / np.timedelta64(1, 'D')
yy['WARNING'].fillna(9999, inplace=True)
```


```
# define a true positive
yy['TRUE_POSITIVE'] = np.where(((yy.EQUIPMENT_FAILURE == 1) &
(yy.WARNING<=forecast_window) &(yy.WARNING>=0) &
(yy.ID_OF_SIGNAL==yy.ID)), 1, 0)

# define a false negative
yy['FALSE_NEGATIVE'] = np.where((yy.TRUE_POSITIVE==0) &
(yy.EQUIPMENT_FAILURE==1), 1, 0)

# define a false positive
yy['BAD_S']=np.where((yy.WARNING<0) | (yy.WARNING>=forecast_window),
1, 0)

yy['FALSE_POSITIVE'] = np.where(((yy.Y_FAILZ == 1) & (yy.BAD_S==1) &
(yy.ID_OF_SIGNAL==yy.ID)), 1, 0)

yy['bootie']=1

yy['CATEGORY']=np.where((yy.FALSE_POSITIVE==1),'FALSE_POSITIVE',

(np.where((yy.FALSE_NEGATIVE==1),'FALSE_NEGATIVE',

(np.where((yy.TRUE_POSITIVE==1),'TRUE_POSITIVE','TRUE_NEGATIVE')))))
```

Define metrics for the testing, training, and validation Data sets.

```
table = pd.pivot_table(yy, values=['bootie'], index=
['MODELING_GROUP'],columns=['CATEGORY'], aggfunc=np.sum)
table
```

Out[132]:

	bootie			
CATEGORY	FALSE_NEGATIVE	FALSE_POSITIVE	TRUE_NEGATIVE	TRUE_POSITIVE
MODELING_GROUP				
TESTING	98	131	108639	51
TRAINING	41	107	106473	105
VALIDATION	79	98	91882	47


Maintenance Scenario	Number of Incidents	Cost per Incident	Total Cost
Unnecessary Maintenance	9	$ 1,500	$ 13,500
Timely and Appropriate Maintenance	27	$ 7,500	$ 202,500
Machine Runs to Failure	385	$ 30,000	$ 11,550,000
Total	421	$ 27,948	$ 11,766,000

A false positive is “Unnecessary Maintenance.” A true positive is a “Timely and Appropriate Maintenance.” A false negative is “Machine Runs to Failure.”

This means that a false positive costs $1,500.

A false negative costs $30,000.

A true positive costs $7,500.

A true negative has no cost because no action is taken.

Now we can calculate the total cost.

```
yy['TOTAL_COST']=yy.FALSE_NEGATIVE*30000+yy.FALSE_POSITIVE*1500+yy.TRU
E_POSITIVE*7500
```

Aggregate the costs by modeling group.

```
table = pd.pivot_table(yy, values=['TOTAL_COST'],index=
['MODELING_GROUP'], aggfunc=np.sum)
table
```


Out[135]:

MODELING_GROUP	TOTAL_COST
TESTING	3519000
TRAINING	2178000
VALIDATION	2869500

Calculate the number of machines in each modeling group.

```
wells=yy[['ID','MODELING_GROUP']]

wells=wells.drop_duplicates(subset='ID')

wells = wells.groupby(['MODELING_GROUP'])['ID'].count()
wells=pd.DataFrame(wells)
wells=wells.rename(columns={"ID": "WELLS"})

wells
```

Out[139]:

MODELING_GROUP	WELLS
TESTING	149
TRAINING	146

Merge the total costs and total machines into one data frame.

```
tc = yy.groupby(['MODELING_GROUP'])['TOTAL_COST'].sum()
tc=pd.DataFrame(tc)
```

Calculate the average cost per machine.

```
ac =tc.merge(wells, on=['MODELING_GROUP'], how='inner')

ac['AVERAGE_COST']=ac.TOTAL_COST/ac.WELLS
ac['LIFT']=27948-ac.AVERAGE_COST

ac
```

Out[143]:

MODELING_GROUP	TOTAL_COST	WELLS	AVERAGE_COST	LIFT
TESTING	3519000	149	23617.449664	4330.550336
TRAINING	2178000	146	14917.808219	13030.191781
VALIDATION	2869500	126	22773.809524	5174.190476

8.0 Conclusions

Now we have everything we need to examine the effectiveness of the model.

Maintenance currently costs the firm about $27,948 dollars per machine in the current data set. In the validation data set, the cost per machine is $22,773.81. This means a predictive maintenance solution will lower the cost per machine by about $5,174 dollars per machine. Multiplied by 421 machines, the equates to a $2.17 million saving or an 18% reduction in total expenses.

problem.

Nonetheless, this exercise should give you a useful reference as you approach these types of problems in the future.

As far as the next steps, I would encourage you to see if you can improve the solution by optimizing the model. Maybe incorporate some hyper-parameter optimization or even try a different model. Let me know how it turns out!