Predicting System Failure

Description

In this Jupyter notebook, we will examine sensory dataset from Nasa where we will create a model in order to predict system failure. This notebook can be generalized to any sensory driven data, such as manufacturing.

Import Modules

- You will first need to install a number of modules in order to follow along with this notebook.
- Most of these packages, such as numpy and pandas, are available using Anaconda (https://conda.io/docs/user-guide/install/index.html).
- For the machine learning pipeline, we will be making use of the BigML Python bindings (https://bigml.readthedocs.io/en/latest/).

In [2]:
```
import numpy as np
import pandas as pd
# import seaborn as sns
```

In [5]:
```
import sys
print(sys.version)
```

```
3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)]
```

Save our BigML Username and Api Key to our environment to access the API

In [26]:
```
import os
os.environ['BIGML_USERNAME'] = "ALANTURING"
os.environ['BIGML_API_KEY'] = "c987d2d9e24f32792ae14856ce79da6a478d9e79"
```

Creat our main API object that all the main functions will utilize.

In [30]:
```
from bigml.api import BigML
api = BigML(project="project/5b61a23d2a83473377000457")
```

Data Description

Data sets consists of multiple multivariate time series. Each time series is from a different engine ñ i.e., the data can be considered to be from a fleet of engines of the same type. Each engine starts with different degrees of initial wear and manufacturing variation which is unknown to the user. This wear and variation is considered normal, i.e., it is not considered a fault condition. There are three operational settings that have a substantial effect on engine performance. These settings are also included in the data. The data is contaminated with sensor noise.

The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure. The data are provided with 26 columns of numbers. Each row is a snapshot of data taken during a single operational cycle, each column is a different variable.

Important Note: Even though the data is presented as a time series, we will be using each row of the time series as an individual instance when modeling. We will add two new fields that use the 5 previous rows of an instance, but otherwise the order is not sustained. This proves to simplify the workflow and not substantially affect error.

Download txt train files into BigML. When creating a dataset in BigML, data can be presented in many forms and is not specific to CSV. We will use this to convert our txt files into CSVs using BigML.

Importing Data to BigML

In order to start a BigML workflow, a source object has to be created. The API function that creates a source is `create_source` . The method's inputs will be a file path to the txt file it will be converting.

In [31]:
```
train_FD001_source = api.create_source("CMAPSSData/train_FD001.txt")
train_FD002_source = api.create_source("CMAPSSData/train_FD002.txt")
train_FD003_source = api.create_source("CMAPSSData/train_FD003.txt")
train_FD004_source = api.create_source("CMAPSSData/train_FD004.txt")
```

BigML's `ok` method is called in order to assure that an object is created and will wait if it is not done being completed.

In [32]:
```
api.ok(train_FD001_source)
api.ok(train_FD002_source)
api.ok(train_FD003_source)
api.ok(train_FD004_source)
```

Out[32]:
```
True
```

Creating a Dataset

BigML will use the newly created source to create datasets which will enable the API to perform many more operations. In order to create a dataset, the API calls the function `create_dataset` . The method will take the source created by the API as an input.

In [33]:
```
train_FD001_origin_dataset = api.create_dataset(train_FD001_source)
train_FD002_origin_dataset = api.create_dataset(train_FD002_source)
train_FD003_origin_dataset = api.create_dataset(train_FD003_source)
train_FD004_origin_dataset = api.create_dataset(train_FD004_source)

api.ok(train_FD001_origin_dataset)
api.ok(train_FD002_origin_dataset)
api.ok(train_FD003_origin_dataset)
api.ok(train_FD004_origin_dataset)
```

Out[33]:
```
True
```

Downloading the Datasets as CSVs

BigML allows for the download of their dataset objects. In order to download a dataset as a CSV, the API calls the function `download_dataset` . The method will take the dataset created by the API and a file path as an input.

In [35]:
```
api.download_dataset(train_FD001_origin_dataset,
    filename='CMAPSSData/train_FD001.csv')

api.download_dataset(train_FD002_origin_dataset,
    filename='CMAPSSData/train_FD002.csv')

api.download_dataset(train_FD003_origin_dataset,
    filename='CMAPSSData/train_FD003.csv')

api.download_dataset(train_FD004_origin_dataset,
    filename='CMAPSSData/train_FD004.csv')
```

Out[35]:
```
'CMAPSSData/train_FD004.csv'
```

We have now converted the text files to CSV formats

Download Newly created CSV files into Notebook as Dataframes

In [36]:
```
train_FD001 = pd.read_csv('CMAPSSData/train_FD001.csv')
train_FD002 = pd.read_csv('CMAPSSData/train_FD002.csv')
train_FD003 = pd.read_csv('CMAPSSData/train_FD003.csv')
train_FD004 = pd.read_csv('CMAPSSData/train_FD004.csv')
```

In [37]:
```
df_list = [train_FD001,train_FD002,train_FD003,train_FD004]
```

An example how the dataframes are presented.

```
In [38]: train_FD001.head(2)
```

Out[38]:

	field1	field2	field3	field4	field5	field6	field7	field8	field9	field10	...	field19	fiel
0	1	1	-0.0007	-0.0004	100	518.67	641.82	1589.70	1400.60	14.62	...	8138.62	8.4
1	1	2	0.0019	-0.0003	100	518.67	642.15	1591.82	1403.14	14.62	...	8131.49	8.4

2 rows × 28 columns

Feature Engineering

We can see that the columns are not labeled since we converted a txt file, and we can also see that there are two columns with no data. This was a small error in converting.

We will frist drop the two unneeded columns at the end of each dataframe.

```
In [39]: for df in df_list:
             df.drop(columns=["field27","field28"],inplace=True)
```

Since we were given the column names in the data folder, we will create a function to label the columns.

1) unit number
2) time, in cycles
3) operational setting 1
4) operational setting 2
5) operational setting 3
6) sensor measurement 1
7) sensor measurement 2
...
26) sensor measurement 21

```
In [40]: def label_columns(df):
             df = df.rename(columns={"field1": "unit_number",
                                     "field2": "time_cycles",
                                     "field3": "op_setting_1",
                                     "field4": "op_setting_2",
                                     "field5": "op_setting_3"})
             for i in np.arange(6,27):
                 df= df.rename(columns={"field"+str(i): "sensory_measure_"+str(i-5)})
             return df
```

In [41]:
```
train_FD001 = label_columns(train_FD001)
train_FD002 = label_columns(train_FD002)
train_FD003 = label_columns(train_FD003)
train_FD004 = label_columns(train_FD004)
```

In [42]:
```
train_FD001.head(2)
```

Out[42]:

	unit_number	time_cycles	op_setting_1	op_setting_2	op_setting_3	sensory_measure_1	sensory_
0	1	1	-0.0007	-0.0004	100	518.67	
1	1	2	0.0019	-0.0003	100	518.67	

2 rows × 26 columns

We will add a column of the calculated sensory data mean for the previous 5 cycles of a row. If the row is in the first five columns of a unit then it will use the mean of its own row.

In [43]:
```
def add_mean(df):
    grouped = df.groupby("unit_number")
    prev_5_mean = []
    for name, group in grouped:
            for i in np.arange(len(group)):
                if (i-5>=0):
                    x = np.mean(df.iloc[2:5,5:26].values)
                    prev_5_mean = np.append(prev_5_mean,x)
                else:
                    x = np.mean(df.iloc[i,5:26])
                    prev_5_mean = np.append(prev_5_mean,x)
   df["prev_5_mean"] = prev_5_mean
    return df
```

In [44]:
```
train_FD001 = add_mean(train_FD001)
train_FD002 = add_mean(train_FD002)
train_FD003 = add_mean(train_FD003)
train_FD004 = add_mean(train_FD004)
```

In [45]:
```
train_FD001.head(2)
```

Out[45]:

	unit_number	time_cycles	op_setting_1	op_setting_2	op_setting_3	sensory_measure_1	sensory_
0	1	1	-0.0007	-0.0004	100	518.67	
1	1	2	0.0019	-0.0003	100	518.67	

2 rows × 27 columns

We will add a column of the calculated sensory data std for the previous 5 cycles of a row. If the row is in the first five columns of a unit then it will use the std of its own row.

In [46]:
```
def add_std(df):
    grouped = df.groupby("unit_number")
    prev_5_std = []
    for name, group in grouped:
            for i in np.arange(len(group)):
                if (i-5>=0):
                    x = np.std(df.iloc[2:5,5:26].values)
                    prev_5_std = np.append(prev_5_std,x)
                else:
                    x = np.std(df.iloc[i,5:26])
                    prev_5_std = np.append(prev_5_std,x)
    df["prev_5_std"] = prev_5_std
    return df
```

In [47]:
```
train_FD001 = add_std(train_FD001)
train_FD002 = add_std(train_FD002)
train_FD003 = add_std(train_FD003)
train_FD004 = add_std(train_FD004)
```

In [48]:
```
train_FD001.head(2)
```

Out[48]:

	unit_number	time_cycles	op_setting_1	op_setting_2	op_setting_3	sensory_measure_1	sensory_
0	1	1	-0.0007	-0.0004	100	518.67	
1	1	2	0.0019	-0.0003	100	518.67	

2 rows × 28 columns

Lastly we will want to create two different target columns.

- The fist will label the remaining useful life (RUL). The RUL is the remaning number of cycles before an engine fails.
- The second will label if the engine will fail in the next 30 cycles.

In [49]:
```
def add_RUL(df):
    grouped = df.groupby("unit_number")
    ser = []
    for name, group in grouped:
        ser = np.append(ser,list(reversed(np.arange(len(group)))))
    df["RUL"] = ser
    return df
```

In [50]:
```
train_FD001 = add_RUL(train_FD001)
train_FD002 = add_RUL(train_FD002)
train_FD003 = add_RUL(train_FD003)
train_FD004 = add_RUL(train_FD004)
```

In [51]:
```
def add_next_30_cycles(df):
    df["failure_next_30_cycles"] = df["RUL"].apply(lambda x: True if x <= 3(
    return df
```

In [52]:
```
train_FD001 = add_next_30_cycles(train_FD001)
train_FD002 = add_next_30_cycles(train_FD002)
train_FD003 = add_next_30_cycles(train_FD003)
train_FD004 = add_next_30_cycles(train_FD004)
```

In [53]:
```
train_FD001.head(2)
```

Out[53]:

	unit_number	time_cycles	op_setting_1	op_setting_2	op_setting_3	sensory_measure_1	sensory_
0	1	1	-0.0007	-0.0004	100	518.67	
1	1	2	0.0019	-0.0003	100	518.67	

2 rows × 30 columns

We will combine all dataframes and convert them into a CSV.

In [54]:
```
combined_df= train_FD001.append(train_FD002).append(train_FD003).append(trai
combined_df.to_csv('CMAPSSData/combined_df.csv')
```

We will download the newly created CSV into BigML and create our dataset using the functions discussed earlier.

In [55]:
```
combined_df_source = api.create_source("CMAPSSData/combined_df.csv")
api.ok(combined_df_source)

combined_df_dataset = api.create_dataset(combined_df_source)
api.ok(combined_df_dataset)
```

Out[55]:
```
True
```

Test-Train Split

Since we want our data to stay in the form of BigML's datasets, the test-train split of the data will be done through BigML's API. This form will allow for the API's computations. The test-train split will be created by the function `create_dataset` mentioned before. However, it will take advantage of the more available inputs of the function. Many BigML API functions take in a dictionary with many fields as an additional input. These fields allow for much manipulation of the original function's outcome. In a test-train split, the field of sample_rate will allow for the choosing of the percentage of data being sampled. The train dataset will have out_of_bag field set to False and the test dataset will have it set to True. Since the test out_of_bag is set to True, its size will be 20% when its sample rate is 80%.

In [77]:
```
train_dataset = api.create_dataset(
    combined_df_dataset, {"name": "Engine Failure | Training",
                    "sample_rate": 0.8, "seed": "my seed"})
test_dataset = api.create_dataset(
    combined_df_dataset, {"name": "Engine Failure | Test",
                    "sample_rate": 0.8, "seed": "my seed",
                    "out_of_bag": True})
api.ok(train_dataset)
api.ok(test_dataset)
```

Out[77]:
```
True
```

Creating Models

We will be having to select two models. One model will be for trying to predict the exact RUL, and will be trained on the “RUL” column, and the other model will be predicting True or False for if a model will fail in the next 30 cycles, and this model will be trained on the target column “failure_next_30_cycles”.

Since BigML has many models for both situations, we will use BigML’s savvy function of OpitML: an optimization process for model selection and parameterization that automatically finds the best supervised model to help solve classification and regression problems.

1) In the first use of optiml, we will exclude the field of “failure_next_30_cycles”, since we will be predicting for remaining useful life (RUL). We will do this since both columns contain similar information and would mislead the training of the new model

In [78]:
```
optiml_RUL = api.create_optiml(train_dataset, {
    "excluded_fields": ["failure_next_30_cycles","field1","unit_number"],
    "objective_field": "RUL"})

api.ok(optiml_RUL)
```

Out[78]:
```
True
```

2) In the second use of optiml, we will exclude the field of “RUL”, since we will be predicting for if an engine is going to fail in the next 30 cycles.

In [79]:
```
optiml_next_30 = api.create_optiml(train_dataset, {
    "excluded_fields": ["RUL","field1","unit_number"],
    "objective_field": "failure_next_30_cycles",
    "metric":"max_phi"})

api.ok(optiml_next_30)
```

Out[79]:
```
True
```

We will choose the two best models for both situations from our OptiML object. We will choose the best model that also has the highest count, both are ensembles.

```
In [80]: optiml_RUL["object"]["optiml"]["summary"]
```

```
Out[80]: {'deepnet': {'best': 'deepnet/5b62e270623db83244003b60', 'count': 2},
          'ensemble': {'best': 'ensemble/5b62ded208b07e51c9009469', 'count': 14},
          'model': {'best': 'model/5b62e273623db8324300c685', 'count': 1}}
```

```
In [81]: RUL_model = optiml_RUL["object"]["optiml"]["summary"]["ensemble"]["best"]
```

```
In [82]: optiml_next_30["object"]["optiml"]["summary"]
```

```
Out[82]: {'deepnet': {'best': 'deepnet/5b62eda18bf7d535f800c282', 'count': 2},
          'ensemble': {'best': 'ensemble/5b62ed78623db8324801c0db', 'count': 5},
          'logisticregression': {'best': 'logisticregression/5b62edd308b07e51d4020
         2cf',
           'count': 1},
          'model': {'best': 'model/5b62edd308b07e51d40202cc', 'count': 1}}
```

```
In [83]: next_30_model = optiml_next_30["object"]["optiml"]["summary"]["ensemble"]["b
```

After choosing our models, we will want to create an evaluation for each model and test our performance.

```
In [84]: RUL_evaluation = api.create_evaluation(RUL_model,test_dataset)
         api.ok(RUL_evaluation)

         next_30_evaluation = api.create_evaluation(next_30_model,test_dataset)
         api.ok(next_30_evaluation)
```

```
Out[84]: True
```

Our first model performs fairly well, with a root means squared error of around 44. This first model is useful because it predicts the remaining life of an engine at any stage. However, if we look at the evaluation of the classification model, we can see that it performs extremely well. This is because the model can detect much better when the sensory data of an engine is starting to change from the norm. So putting the last model up for production would be useful in identifying engines that might fail and sending resources before it actually happens. This could be applied to any sensory data, such as companies that use production lines.

```
In [91]: print("First Model: ")
         print(RUL_evaluation["object"]["result"]["model"])
         print("Roots Mean Squared Error: ", np.sqrt(RUL_evaluation["object"]["result
```

```
First Model: 
{'mean_absolute_error': 31.362, 'mean_squared_error': 1964.06253, 'per_cl
ass_statistics': [], 'r_squared': 0.7146}
Roots Mean Squared Error:  44.317745091554464
```

In [88]:
```
print("Classification model: ")
print("Accuracy: ",next_30_evaluation["object"]["result"]["model"]['accuracy
print("Average Recall: ",next_30_evaluation["object"]["result"]["model"]['av
print("Average Precision: ",next_30_evaluation["object"]["result"]["model"]|
```

```
Classification model:
Accuracy:  0.95825
Average Recall:  0.93867
Average Precision:  0.89692
```

In [ ]: