Tagup data science challenge

ExampleCo, Inc has a problem: maintenance on their widgets is expensive. They have contracted with Tagup to help them implement predictive maintenance. They want us to provide a model that will allow them to prioritize maintenance for those units most likely to fail, and in particular to gain some warning---even just a few hours!---before a unit does fail.

They collect two kinds of data for each unit. First, they have a remote monitoring system for the motors in each unit, which collects information about the motor (rotation speed, voltage, current) as well as two temperature probes (one on the motor and one at the inlet). Unfortunately, this system is antiquated and prone to communication errors, which manifest as nonsense measurements. Second, they have a rule-based alarming system, which can emit either warnings or errors; the system is known to be noisy, but it's the best they've got.

They have given us just over 100MB of historical remote monitoring data from twenty of their units that failed in the field. The shortest-lived units failed after a few days; the longest-lived units failed after several years. Typical lifetimes are on the order of a year. This data is available in .csv files under `data/train` in this repository. In addition, they have provided us with operating data from their thirty active units for the past month; this data is available under `data/test` in this repository.

You have two main objectives. First, **tell us as much as you can about the process that generated the data**. Does it show meaningful clustering? Do the observations appear independent? How accurately can we forecast future observations, and how long a window do we need to make an accurate forecast? Feel free to propose multiple models, but be sure to discuss the ways each is useful and the ways each is not useful. Second, **predict which of the thirty active units are most likely to fail**. The data from these units are in `data/test` . Be sure to quantify these predictions, and especially your certainty.

A few notes to help:

1. A good place to start is by addressing the noise due to comm errors.
2. There is a signal in the data that you can identify and exploit to predict failure. Each machine failed immediately after the last recorded timestamp in the remote monitoring timeseries data.
3. If you can't find the signal in the noise, don't despair! We're much more interested in what you try and how you try it than in how successful you are at helping a fictional company with their fictional problems.
4. Feel free to use any libraries you like, or even other programming languages. Your final results should be presented in this notebook, however.
5. There are no constraints on the models or algorithms you can bring to bear. Some ideas include: unsupervised clustering algorithms such as k-means; hidden Markov models; forecasting models like ARMA; neural networks; survival models built using features extracted from the data; etc.
6. Don't feel compelled to use all the data if you're not sure how. Feel free to focus on data from a single unit if that makes it easier to get started.
7. Be sure to clearly articulate what you did, why you did it, and how the results should be interpreted. In particular you should be aware of the limitations of whatever approach or approaches you take.
8. Don't hesitate to reach out with any questions.

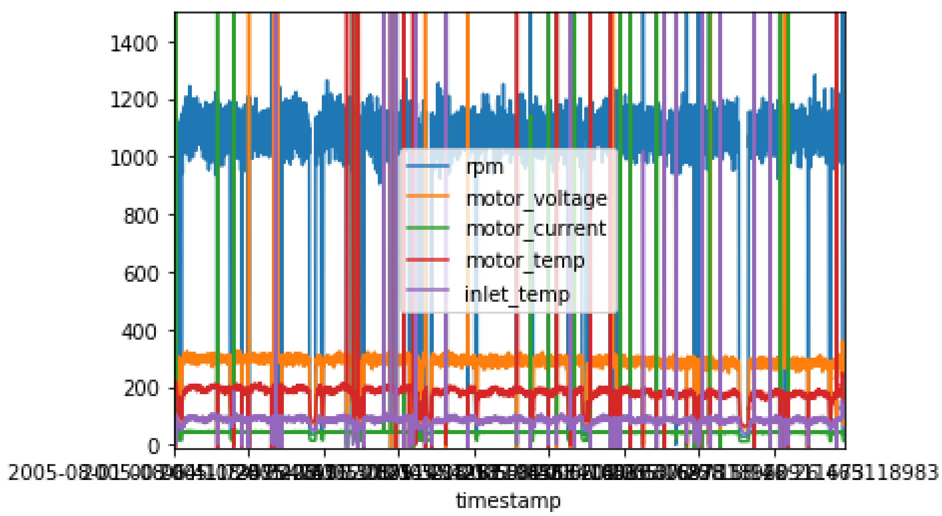
In [651]:
```
#!pip3 install statsmodels
#!pip3 install ipdb
#!pip3 install seaborn
#!pip3 install rpy2
#!pip3 install lifelines
#!pip3 install keras
#!pip3 install tensorflow
#!pip3 install pypandoc
```

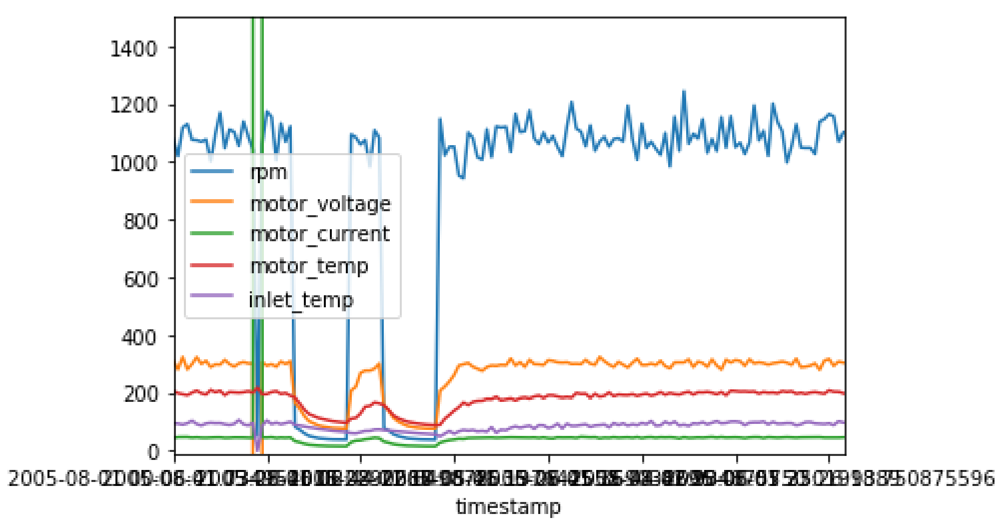
```
Requirement already satisfied: pypandoc in /Users/ajeyakempegowda/.virt
ualenvs/tagup/lib/python3.6/site-packages (1.4)
Requirement already satisfied: setuptools in /Users/ajeyakempegowda/.vi
rtualenvs/tagup/lib/python3.6/site-packages (from pypandoc) (41.0.1)
Requirement already satisfied: wheel>=0.25.0 in /Users/ajeyakempegowd
a/.virtualenvs/tagup/lib/python3.6/site-packages (from pypandoc) (0.33.
4)
Requirement already satisfied: pip>=8.1.0 in /Users/ajeyakempegowda/.vi
rtualenvs/tagup/lib/python3.6/site-packages (from pypandoc) (19.1.1)
```

In [1]:
```
# To help you get started...
from IPython.display import display
import pandas as pd
import matplotlib.pyplot as plt

import lifelines as ll
import seaborn as sns
%matplotlib inline

def load_rms(filename):
    return pd.read_csv(filename, index_col="timestamp")

def load_alarms(filename):
    return pd.read_csv(filename, header=None, names=["timestamp", "messa
ge"], index_col="timestamp")

rms = load_rms('data/train/unit0000_rms.csv')
alarms = load_alarms('data/train/unit0000_alarms.csv')
rms.loc["2005-08-01":"2005-09-01"].plot(ylim=(-10, 1500))
rms.loc["2005-08-01":"2005-08-02"].plot(ylim=(-10, 1500))

display(rms.describe())
display(alarms.describe())
```

	rpm	motor_voltage	motor_current	motor_temp	inlet_temp
count	9.171500e+04	9.171500e+04	9.171500e+04	9.171500e+04	9.171500e+04
mean	-8.111152e+46	-4.431337e+60	-1.387827e+56	-6.640742e+69	-1.937422e+48
std	2.456380e+49	1.903274e+63	4.202968e+58	2.011115e+72	5.867349e+50
min	-7.439020e+51	-5.565298e+65	-1.272847e+61	-6.090557e+74	-1.776896e+53
25%	1.017725e+03	2.311319e+02	3.239347e+01	1.179731e+02	6.489677e+01
50%	1.066347e+03	2.667104e+02	3.944687e+01	1.564521e+02	7.799157e+01
75%	1.106721e+03	2.993994e+02	4.607173e+01	1.978223e+02	9.086132e+01
max	7.978110e+44	1.500194e+65	1.555360e+55	3.117856e+55	1.710299e+40

	message
count	305
unique	2
top	warning
freq	304





Data processing

In [634]:
```
import plotly
import plotly.plotly as py
import plotly.graph_objs as go

import pandas as pd

from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from scipy import stats

from lifelines import KaplanMeierFitter
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.graphics.gofplots import qqplot
import itertools

import warnings
from pandas import read_csv
from pandas import datetime
from statsmodels.tsa.arima_model import ARIMA
from sklearn.metrics import mean_squared_error

import numpy as np
from sklearn.model_selection import train_test_split
import statsmodels.api as sm

plotly.tools.set_credentials_file(username='kempegowda.a', api_key='cF87
0W5z9vOzGeP0O5iv')
```

In [635]:
```
#creating a wrapper class so that its easier to access data(in the form
 of data frames) for all the given sensors
class RMS(object):
    """
    Wrapper function to wrap data for all sensors
    """

    def __init__(self, rms_path=None, alarm_path=None):
        """
        To initialize test/train data
        """
        if rms_path is not None:
            self.load_rms(rms_path)
        if alarm_path is not None:
            self.load_alarms(alarm_path)

    def load_rms(self, filename):
        """
        To load RMS data(train/test)
        """
        self.rms = pd.read_csv(filename, index_col="timestamp")
        self.rms_processed = pd.read_csv(filename, index_col="timestamp"
)
        self.rms_processed.reset_index(level=0, inplace=True)
        self.rms_processed = self.rms_processed.dropna(axis=0)

    def assign_cleaned_df(self, df):
        self.noise_free_rms = df

    def load_alarms(self, filename):
        """
        To load Alarm data
        """
        self.alarm = pd.read_csv(filename, header=None, names=["timestam
p", "message"], index_col="timestamp")
        self.alarm_processed = pd.read_csv(filename, header=None, names=
["timestamp", "message"], index_col="timestamp")
        self.alarm_processed.reset_index(level=0, inplace=True)
        self.alarm_processed = self.alarm_processed.dropna(axis=0)

    def plotly_rms_ts(self, df, title= None):
        """
        Plotly viz to generate RMS graph
        """
        rpm = go.Scatter(
            x=df.timestamp,
            y=df['rpm'],
            name = "RPM",
            line = dict(color = '#17BECF'),
            opacity = 0.8)
        motor_voltage = go.Scatter(
            x=df.timestamp,
            y=df['motor_voltage'],
            name = "Motor Voltage",
            line = dict(color = '#7F7F7F'),
            opacity = 0.8)
```

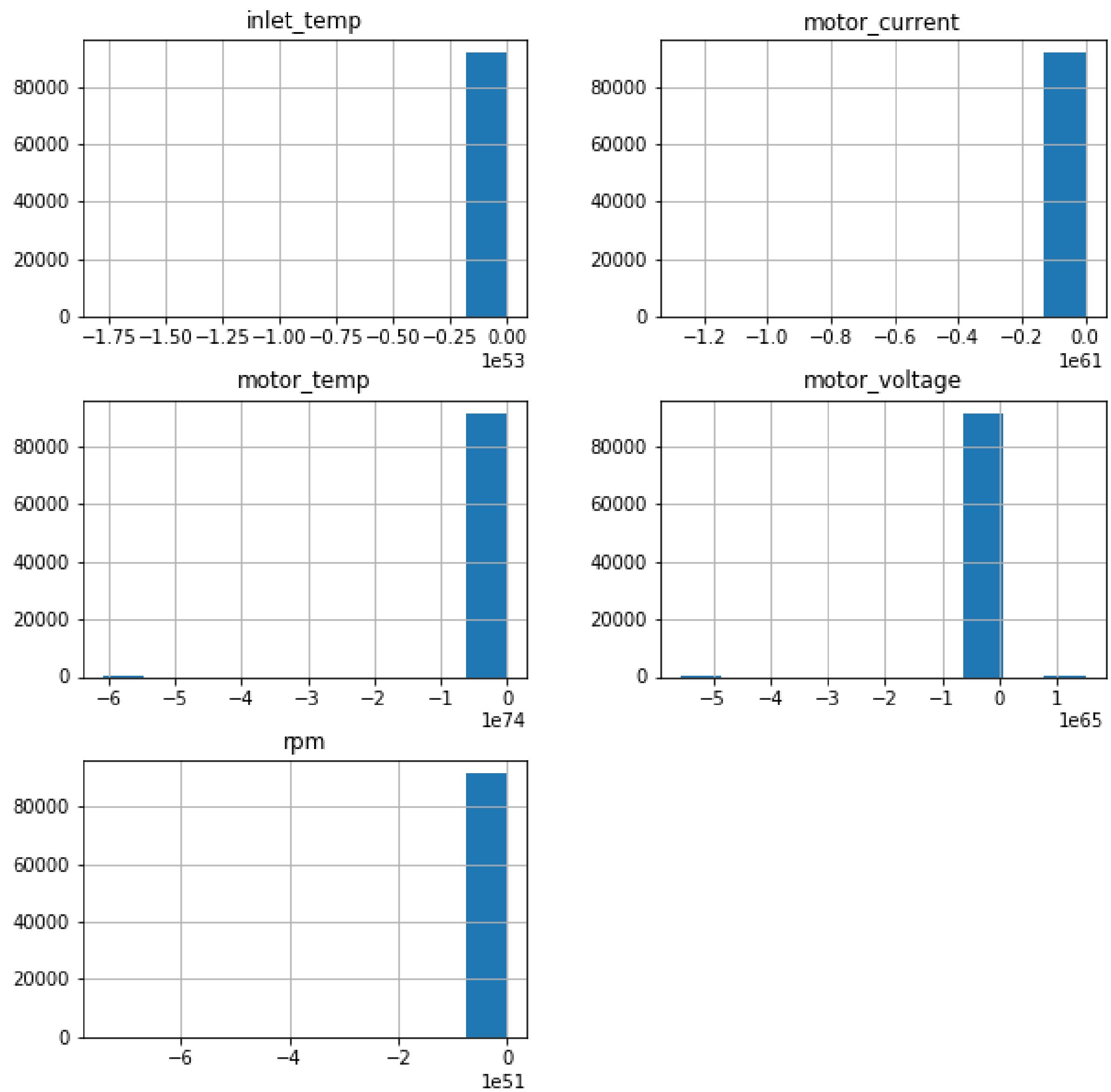
```
        motor_current = go.Scatter(
            x=df.timestamp,
            y=df['motor_current'],
            name = "Motor Current",
            line = dict(color = '#d62728'),
            opacity = 0.8)
        motor_temp = go.Scatter(
            x=df.timestamp,
            y=df['motor_temp'],
            name = "Motor Temperature",
            line = dict(color = '#e377c2'),
            opacity = 0.8)
        inlet_temp = go.Scatter(
            x=df.timestamp,
            y=df['inlet_temp'],
            name = "Inlet Temperature",
            line = dict(color = '#ff7f0e'),
            opacity = 0.8)
        motor_data = [rpm,motor_voltage, motor_current, motor_temp,inlet
_temp]
        layout = dict(
            title=title,
            xaxis=dict(
                rangeselector=dict(
                    buttons=list([
                        dict(count=1,
                             label='1m',
                             step='month',
                             stepmode='backward'),
                        dict(count=12,
                             label='30m',
                             step='month',
                             stepmode='backward'),
                        dict(step='all')
                    ])
                ),
                rangeslider=dict(
                    visible = True
                ),
                type='date'
            )
        )
        fig = dict(data=motor_data, layout=layout)
        return py.iplot(fig, filename = title)
```

In [636]:
```
class HelperFunction(object):
    """
    Helper function
    """

    def remove_outliers(self, sensor, custom_range = [0.05, 0.95]):
        """
        To remove outliers based on threshold
        """
        self.df = sensor.loc[:, sensor.columns != 'Index']
        self.quantile = self.df.quantile(custom_range)
        return self.outlier_lambda(custom_range)

    def outlier_lambda(self, custom_range):
        """
        Defining threshold values
        """
        clean_df = self.df.apply(
            lambda element: element[
                (element > self.quantile.loc[custom_range[0],element.nam
e]) &
                (element < self.quantile.loc[custom_range[1],element.nam
e])],
            axis=0)
        clean_df = clean_df.dropna(axis=0)
        return clean_df

    def generate_file_paths(self, iter_range,is_train=True):
        """
        To programmatically generate test and train file paths
        """
        ds = 'train' if is_train else 'test'
        paths_dict = {}
        for i in range(iter_range[0],iter_range[1]):
            paths_list=[]
            key = 'sensor'+str(i)
            unit_id = str(i).zfill(4)
            gen_rms_file_path = 'data/'+ ds +'/unit' + unit_id  +'_rms.c
sv'
            gen_alarm_file_path = 'data/'+ ds +'/unit'+ unit_id +'_alarm
s.csv'
            paths_list.append(gen_rms_file_path)
            paths_list.append(gen_alarm_file_path)
            paths_dict[key] = paths_list
        return paths_dict
```

Initial EDA suggests that data is skewed i.e data is prone to outliers

In [637]:
```
rms.hist(figsize=(10,10))
```

Out[637]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1a05d6198>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a14493c8>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1a14759e8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a14a3f60>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1a4e71550>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1a4e9eb00
>]],
      dtype=object)
```



To generate file paths for train or test data

In [638]:
```
units = [0,20]
helper = HelperFunction()
file_paths = helper.generate_file_paths(units)
file_paths
```

Out[638]:
```
{'sensor0': ['data/train/unit0000_rms.csv', 'data/train/unit0000_alarm
s.csv'],
 'sensor1': ['data/train/unit0001_rms.csv', 'data/train/unit0001_alarm
s.csv'],
 'sensor2': ['data/train/unit0002_rms.csv', 'data/train/unit0002_alarm
s.csv'],
 'sensor3': ['data/train/unit0003_rms.csv', 'data/train/unit0003_alarm
s.csv'],
 'sensor4': ['data/train/unit0004_rms.csv', 'data/train/unit0004_alarm
s.csv'],
 'sensor5': ['data/train/unit0005_rms.csv', 'data/train/unit0005_alarm
s.csv'],
 'sensor6': ['data/train/unit0006_rms.csv', 'data/train/unit0006_alarm
s.csv'],
 'sensor7': ['data/train/unit0007_rms.csv', 'data/train/unit0007_alarm
s.csv'],
 'sensor8': ['data/train/unit0008_rms.csv', 'data/train/unit0008_alarm
s.csv'],
 'sensor9': ['data/train/unit0009_rms.csv', 'data/train/unit0009_alarm
s.csv'],
 'sensor10': ['data/train/unit0010_rms.csv', 'data/train/unit0010_alarm
s.csv'],
 'sensor11': ['data/train/unit0011_rms.csv', 'data/train/unit0011_alarm
s.csv'],
 'sensor12': ['data/train/unit0012_rms.csv', 'data/train/unit0012_alarm
s.csv'],
 'sensor13': ['data/train/unit0013_rms.csv', 'data/train/unit0013_alarm
s.csv'],
 'sensor14': ['data/train/unit0014_rms.csv', 'data/train/unit0014_alarm
s.csv'],
 'sensor15': ['data/train/unit0015_rms.csv', 'data/train/unit0015_alarm
s.csv'],
 'sensor16': ['data/train/unit0016_rms.csv', 'data/train/unit0016_alarm
s.csv'],
 'sensor17': ['data/train/unit0017_rms.csv', 'data/train/unit0017_alarm
s.csv'],
 'sensor18': ['data/train/unit0018_rms.csv', 'data/train/unit0018_alarm
s.csv'],
 'sensor19': ['data/train/unit0019_rms.csv', 'data/train/unit0019_alarm
s.csv']}
```

In [639]:
```
helper.generate_file_paths([20,50], False)
```

Out[639]:
```
{'sensor20': ['data/test/unit0020_rms.csv', 'data/test/unit0020_alarms.
csv'],
 'sensor21': ['data/test/unit0021_rms.csv', 'data/test/unit0021_alarms.
csv'],
 'sensor22': ['data/test/unit0022_rms.csv', 'data/test/unit0022_alarms.
csv'],
 'sensor23': ['data/test/unit0023_rms.csv', 'data/test/unit0023_alarms.
csv'],
 'sensor24': ['data/test/unit0024_rms.csv', 'data/test/unit0024_alarms.
csv'],
 'sensor25': ['data/test/unit0025_rms.csv', 'data/test/unit0025_alarms.
csv'],
 'sensor26': ['data/test/unit0026_rms.csv', 'data/test/unit0026_alarms.
csv'],
 'sensor27': ['data/test/unit0027_rms.csv', 'data/test/unit0027_alarms.
csv'],
 'sensor28': ['data/test/unit0028_rms.csv', 'data/test/unit0028_alarms.
csv'],
 'sensor29': ['data/test/unit0029_rms.csv', 'data/test/unit0029_alarms.
csv'],
 'sensor30': ['data/test/unit0030_rms.csv', 'data/test/unit0030_alarms.
csv'],
 'sensor31': ['data/test/unit0031_rms.csv', 'data/test/unit0031_alarms.
csv'],
 'sensor32': ['data/test/unit0032_rms.csv', 'data/test/unit0032_alarms.
csv'],
 'sensor33': ['data/test/unit0033_rms.csv', 'data/test/unit0033_alarms.
csv'],
 'sensor34': ['data/test/unit0034_rms.csv', 'data/test/unit0034_alarms.
csv'],
 'sensor35': ['data/test/unit0035_rms.csv', 'data/test/unit0035_alarms.
csv'],
 'sensor36': ['data/test/unit0036_rms.csv', 'data/test/unit0036_alarms.
csv'],
 'sensor37': ['data/test/unit0037_rms.csv', 'data/test/unit0037_alarms.
csv'],
 'sensor38': ['data/test/unit0038_rms.csv', 'data/test/unit0038_alarms.
csv'],
 'sensor39': ['data/test/unit0039_rms.csv', 'data/test/unit0039_alarms.
csv'],
 'sensor40': ['data/test/unit0040_rms.csv', 'data/test/unit0040_alarms.
csv'],
 'sensor41': ['data/test/unit0041_rms.csv', 'data/test/unit0041_alarms.
csv'],
 'sensor42': ['data/test/unit0042_rms.csv', 'data/test/unit0042_alarms.
csv'],
 'sensor43': ['data/test/unit0043_rms.csv', 'data/test/unit0043_alarms.
csv'],
 'sensor44': ['data/test/unit0044_rms.csv', 'data/test/unit0044_alarms.
csv'],
 'sensor45': ['data/test/unit0045_rms.csv', 'data/test/unit0045_alarms.
csv'],
 'sensor46': ['data/test/unit0046_rms.csv', 'data/test/unit0046_alarms.
csv'],
 'sensor47': ['data/test/unit0047_rms.csv', 'data/test/unit0047_alarms.
csv'],
 'sensor48': ['data/test/unit0048_rms.csv', 'data/test/unit0048_alarms.
```

```
csv'],
 'sensor49': ['data/test/unit0049_rms.csv', 'data/test/unit0049_alarms.
csv']}
```

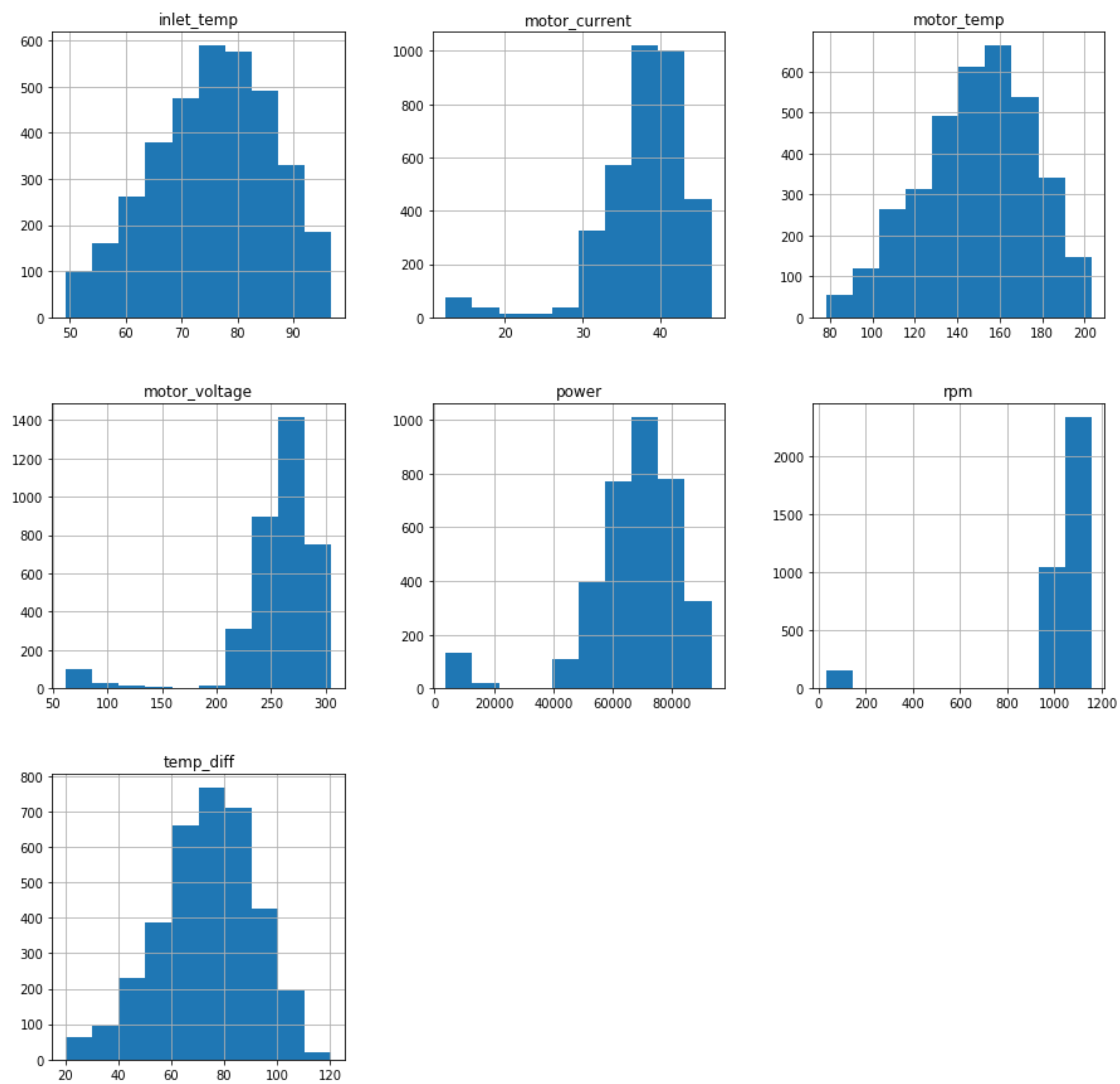
In [641]:
```
def get_test_or_train_sensors(units, file_paths = None):
    """
    To create test and train sensor objects along with 2 interaction ter
ms - power and temp diff
    Applying physics laws to generate 2 new columns:
        power(P=VI) and
        heat dissipation: temp_diff(sink temperatute-source)
    These columns are created to explore if the trends of power generate
d(and proper heat dissipation
    as improper heat management leads to overheating which leads to equi
pment failure) over the course
    of time affects the motor's life
    """
    if file_paths:
        sensors = []
#         import ipdb;ipdb.set_trace()
        for each_sensor in range(units[0], units[1]):
            sensor_var = 'sensor' + str(each_sensor)
            sensor_obj = RMS(rms_path=file_paths.get(sensor_var)[0],
                                alarm_path = file_paths.get(sensor_var)[1])
            rms_cleaned_df = helper.remove_outliers(sensor_obj.rms)
            sensor_obj.assign_cleaned_df(rms_cleaned_df)
            sensor_obj.noise_free_rms_processed = sensor_obj.noise_free_
rms.copy()
            sensor_obj.noise_free_rms_processed.reset_index(level=0, inp
lace=True)
            sensor_obj.noise_free_rms_processed['timestamp'] = pd.to_dat
etime(
                sensor_obj.noise_free_rms_processed['index'], format="%Y
-%m-%d %H:%M:%S")
            sensor_obj.noise_free_rms_processed['power'] = sensor_obj.no
ise_free_rms_processed['motor_voltage'] *  sensor_obj.noise_free_rms_pro
cessed['motor_voltage']
            sensor_obj.noise_free_rms_processed['temp_diff'] = sensor_ob
j.noise_free_rms_processed['motor_temp'] -  sensor_obj.noise_free_rms_pr
ocessed['inlet_temp']
            sensors.append(sensor_obj)
            del sensor_obj
        return sensors
```

In [642]:
```
#create trains sensors objects
sensors = get_test_or_train_sensors([0,20],helper.generate_file_paths(un
its))
```

In [643]:
```
#create test sensors objects
test_sensors = get_test_or_train_sensors([20,50], helper.generate_file_p
aths([20,50], False))
```

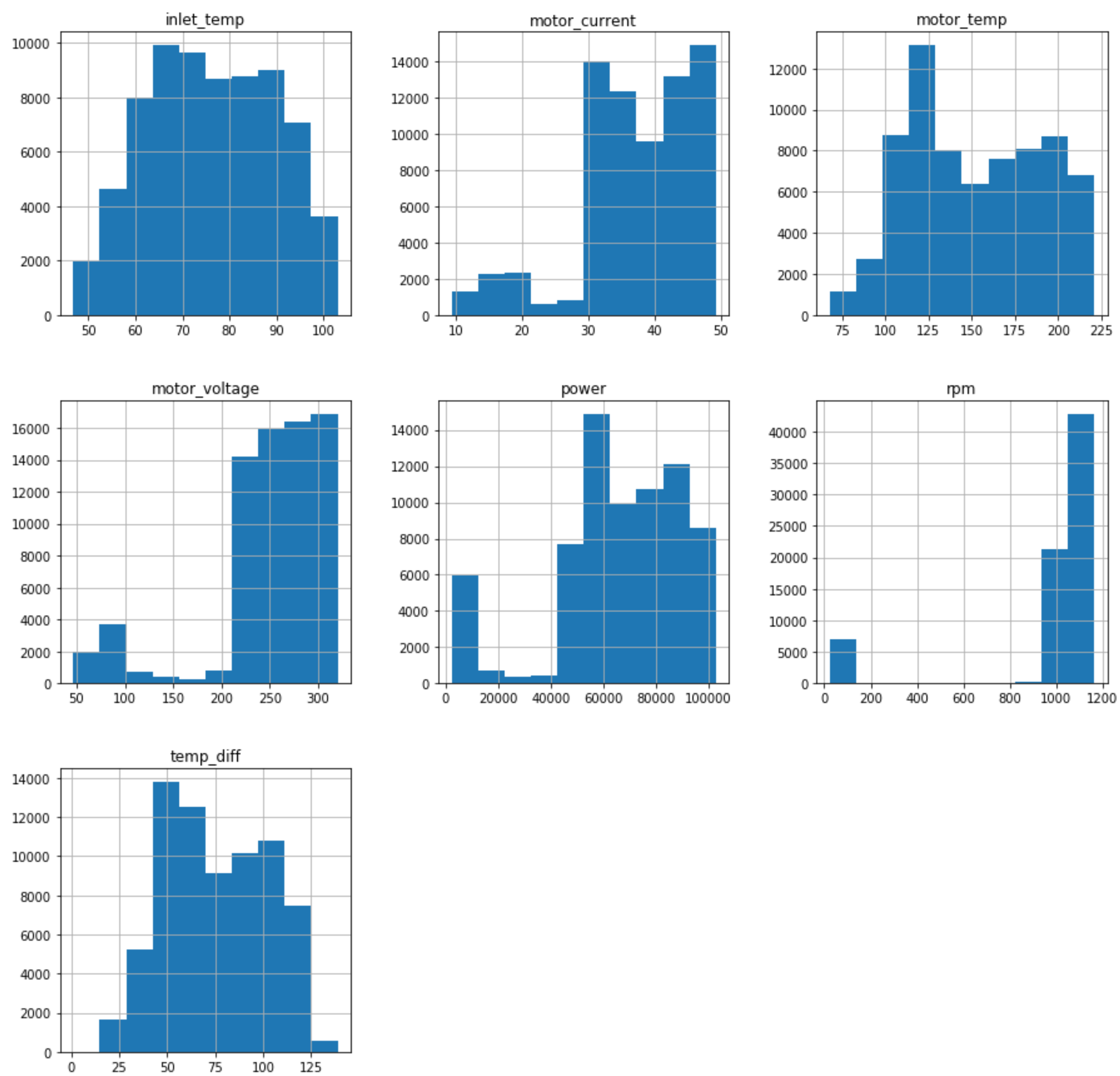
In [644]:
```
# after the outliers have been removed, the distribution looks much bett
er - test sensor
test_sensors[0].noise_free_rms_processed.hist(figsize=(15,15))
```

Out[644]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x15feb8080>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x191bd1588>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x15c18b588>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x191a173c8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x188c7c518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x18f825668>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x15c2467b8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x190f08940>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x190f08978
>]],
      dtype=object)
```



In [645]:
```
# after the outliers have been removed, the distribution looks much bett
er - train sensor
sensors[0].noise_free_rms_processed.hist(figsize=(15,15))
```

Out[645]:
```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x189fac8d0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x18ae02240>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x18add9780>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x166152d30>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x166015320>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x190c328d0>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x18bd9fe80>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x18a4be4a8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x18a4be4e0
>]],
      dtype=object)
```



In [646]:
```
#3d plots using Plotly library for better viz after noise have been remo
ved.
sensors[2].plotly_rms_ts(sensors[2].noise_free_rms_processed, title = "S
ensor2: Denoised")
```

Out[646]:

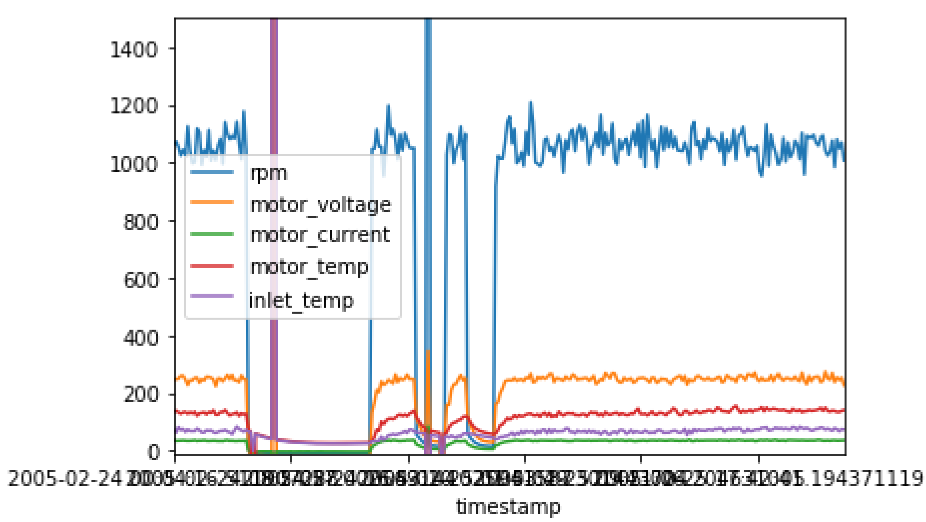
In [647]:
```
sensors[9].plotly_rms_ts(sensors[9].noise_free_rms_processed, title = "Sensor9: Denoised")
```

```
The draw time for this plot will be slow for clients without much RAM.
```

Out[647]:

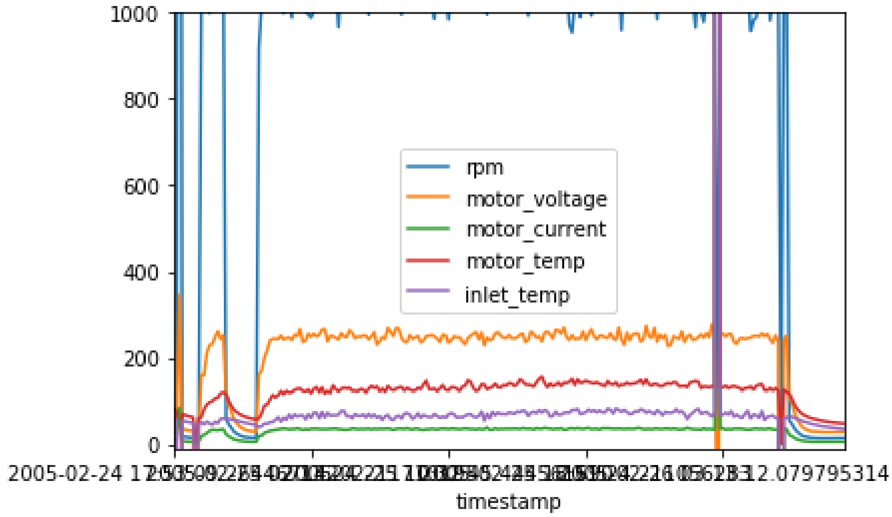
In [648]:
```
sensors[9].rms.loc["2005-02-24":"2005-02-26"].plot(ylim=(-10, 1500))
```

Out[648]: <matplotlib.axes._subplots.AxesSubplot at 0x15a2d75c0>



In [649]:
```
sensors[9].rms.loc["2005-02-24 17:52:05.042022256":"2005-02-26 10:39:35.
505043604"].plot(ylim=(-10, 1000))
```

Out[649]: <matplotlib.axes._subplots.AxesSubplot at 0x15a523898>



Trying to analyse the hidden signal as specified in the question that immediately led to the breakdown of the motor. In my perspective except for the fact that RPM took a steep dip to 0 frequently in the end stages there are no other visiable signals that could help us narrow it down. However, looking at the overall lifecycle of the motor, the RPM drops to zero(because of low voltage and current) did not make the motor faulty in any other earlier time point.

So, I'm not completely convinced that RPM going to 0 frequently within a time window led to the failure of the equipment. Generally speaking improper heat management or wild fluctuations of voltage and current does make the equipments go faulty. But this theory is not completely backed up by our data. The factors could be latent as well if we're to assume some assumptions.

In [594]:
```
sensors[19].plotly_rms_ts(sensors[19].noise_free_rms_processed, title =
"Sensor19: Denoised")
```

```
The draw time for this plot will be slow for all clients.
```

Out[594]:

In [595]:
```
sensors[1].plotly_rms_ts(sensors[11].noise_free_rms_processed, title =
"Sensor1: Denoised")
```

```
The draw time for this plot will be slow for all clients.
```

Out[595]:

Data generative process:

In [617]:
```
sensors[9].alarm.loc["2005-02-24 17:52:05.042022257":"2005-02-26 10:39:35.505043603"]
```

Out[617]:

	message
timestamp	
2005-02-24 17:52:05.042022257	warning
2005-02-24 18:04:57.867410258	warning
2005-02-24 18:26:51.050476258	warning
2005-02-26 00:03:41.086676034	warning
2005-02-26 07:35:13.111720603	warning
2005-02-26 08:20:47.090539603	warning
2005-02-26 09:08:29.938814603	warning
2005-02-26 09:50:05.329822603	warning
2005-02-26 10:05:48.206773603	warning
2005-02-26 10:39:35.505043603	warning

In [618]:
```
sensors[9].rms.loc["2005-02-24 17:11:33.983617938":"2005-02-24 19:33:06.055260093"]
```

Out[618]:

timestamp	rpm	motor_voltage	motor_current	motor_temp	inlet_temp
2005-02-24 17:11:33.983617938	1051.753106	246.098534	37.634945	1.388264e+02	6.710506e+01
2005-02-24 17:23:56.777062258	55.860294	111.510112	22.156585	1.168839e+02	6.461122e+01
2005-02-24 17:31:54.870145103	46.555404	92.910383	18.618010	1.044332e+02	6.343561e+01
2005-02-24 17:42:06.700977430	36.271451	72.634827	14.584470	9.178469e+01	6.183100e+01
2005-02-24 17:53:09.264467142	28.284597	56.754719	11.118397	8.228705e+01	5.979363e+01
2005-02-24 18:03:23.507335606	23.295827	46.809437	9.402220	7.664972e+01	5.803451e+01
2005-02-24 18:11:55.683468344	5648.392523	346.431998	84.190571	-4.011995e+12	-3.878493e+07
2005-02-24 18:23:56.893561947	18.312001	36.532352	7.209578	6.967994e+01	5.457424e+01
2005-02-24 18:32:12.140621523	17.380738	34.574573	6.962278	6.792990e+01	5.346782e+01
2005-02-24 18:41:08.798460262	16.780815	32.764557	6.591236	6.609839e+01	5.228995e+01
2005-02-24 18:52:47.147424966	15.860268	31.778324	6.434830	6.431053e+01	5.079972e+01
2005-02-24 19:03:46.924916494	15.663574	31.142226	6.196356	6.260098e+01	4.955087e+01
2005-02-24 19:13:12.202454240	5.346230	-4167.844964	0.058290	-7.527910e+00	-1.217358e+02
2005-02-24 19:23:38.870598037	15.320507	30.692176	6.065363	6.057625e+01	4.769478e+01
2005-02-24 19:33:06.055260093	1027.117731	160.067336	18.752917	6.348033e+01	5.097882e+01

To formulate a theory about the data generative process let's consider a subset of the data from unit 9 both from alarms and rms csv. I've chosen a random timestamp from alarms file and to tried to check for readings from 'rms' file that lead to a 'warning'

Looking at the data, it is clear that the readings are taken almost 15 minutes(on an average) apart to generate. The data logged in the alarms file is likely due to some business logic factoring in RMS sensor values.

The timestamp is "2005-02-24 17:52:05.042022257" (first index in the above alarms subset) is logged as a warning. Looking at the data (RPM value)in the RMS file, the previous 3 samples at timestamps(2005-02-24 17:23:56.777062258, 2005-02-24 17:31:54.870145103, 2005-02-24 17:42:06.700977430) were consitently low - (55.860294,46.555404,36.271451) respectively. When the next sampling value came in at 2005-02-24 17:53:09.264467142 the system is likely to flag this motor/unit and has logged a warning.

Further, even the next reading from the unit at 2005-02-24 18:03:23.507335606 read a low RPM value. The system has flagged and logged a warning again.

The next reading at 2005-02-24 18:11:55.683468344 shows that the RPM value has been restored to a healthy value and the system unflags this unit and doesn't log any message.

Looking further down the timeline the same set of behavior repeats at 2005-02-24 18:26:51.050476258 where the pevious 3 reading from the unit shows a low RPM value.

In [597]:
```
sensors[9].plotly_rms_ts(sensors[9].noise_free_rms_processed, title = "S
ensor9: Denoised")
```

```
The draw time for this plot will be slow for clients without much RAM.
```

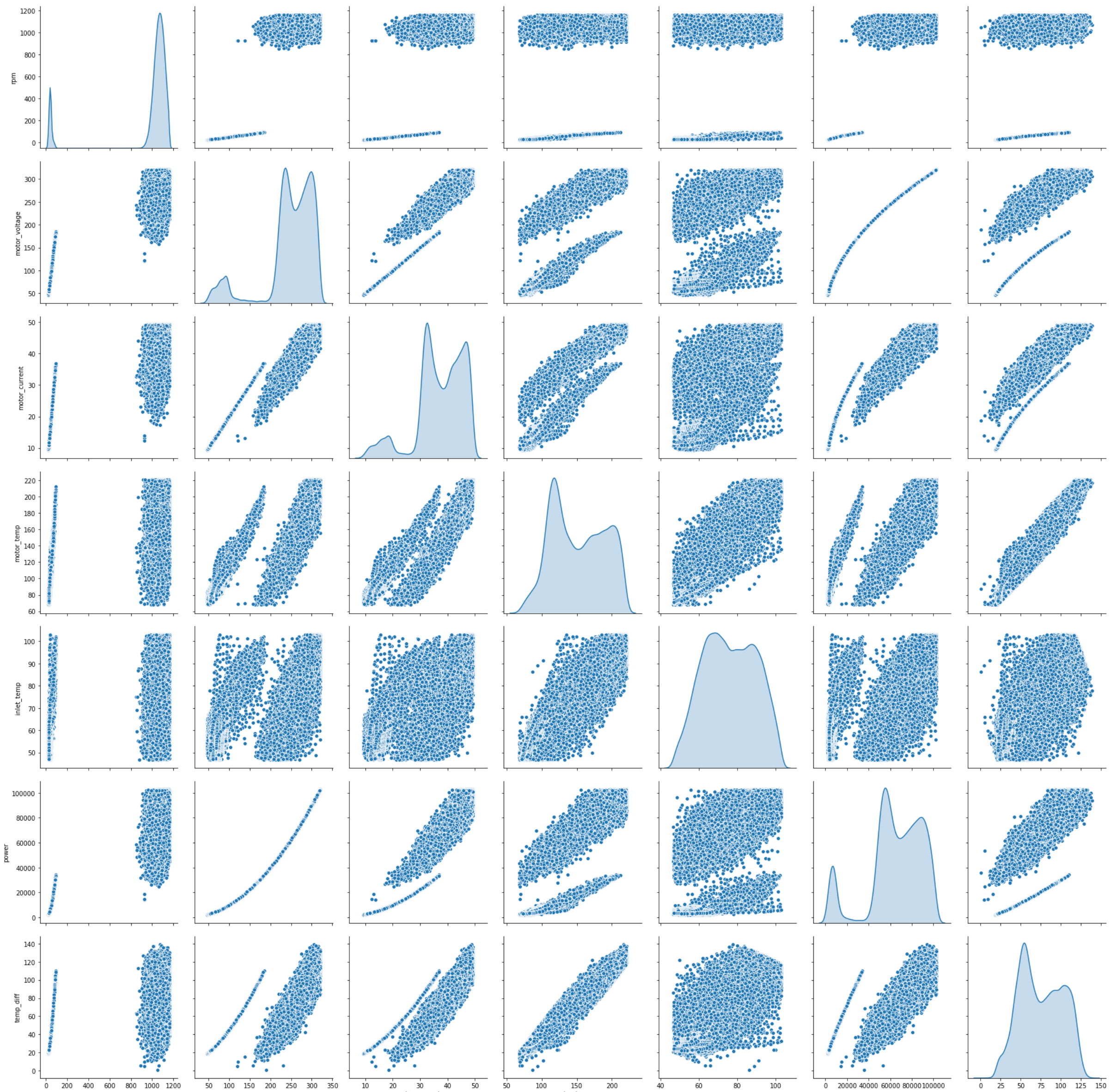
Out[597]:

In [14]:
```
def plot_heatmaps(sensor):
    """
    To create correlation maps
    """
    cols = ['rpm', 'motor_voltage', 'motor_current', 'motor_temp', 'inle
t_temp']
    f = plt.figure(figsize=(10, 10))
    plt.matshow(sensor.noise_free_rms_processed[cols].corr(), fignum=f.n
umber)
    plt.xticks(range(len(cols)), cols, fontsize=14, rotation=90)
    plt.yticks(range(len(cols)), cols, fontsize=14)
    cb = plt.colorbar()
    cb.ax.tick_params(labelsize=14)
```

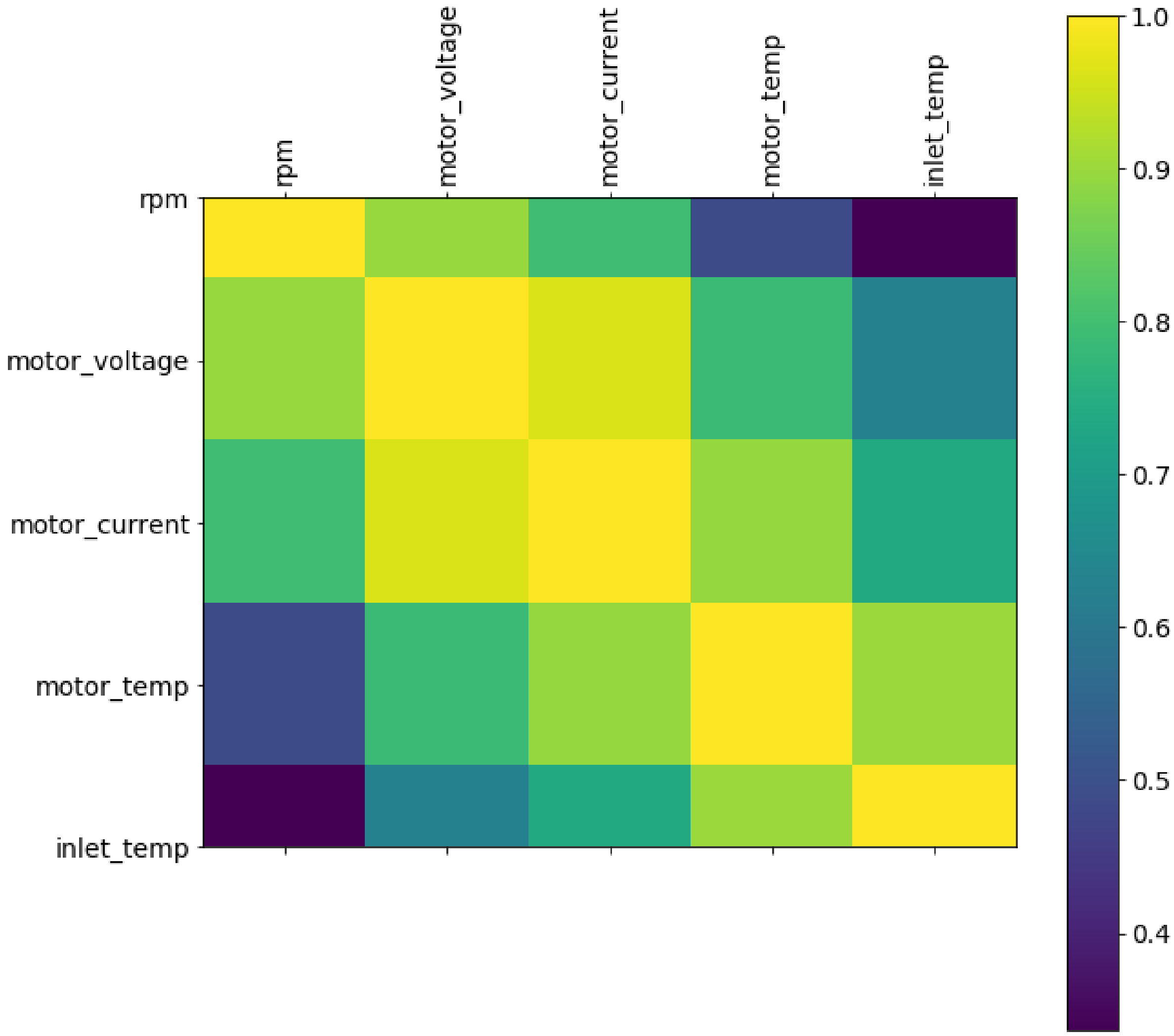
It is implicit from the data that Motor's speed(RPM) is dependant on voltage and current supplied. Let's statistically verify by making use of correlation plots and pairplots

In [15]:
```
# Pairplot to visualize correlation between terms
sns.pairplot(sensors[0].noise_free_rms_processed,  palette="Set2", diag_
kind="kde", height=3.5)
```

Out[15]: <seaborn.axisgrid.PairGrid at 0x133f50470>



In [16]:
```
plot_heatmaps(sensors[1])
```



In [17]:
```
#subsetting the dataframe for quick processing of vizs
frames = []
for i in sensors:
    frames.append(i.noise_free_rms_processed)
result = pd.concat(frames)
result.head()

temp_df = result[:30000]
```

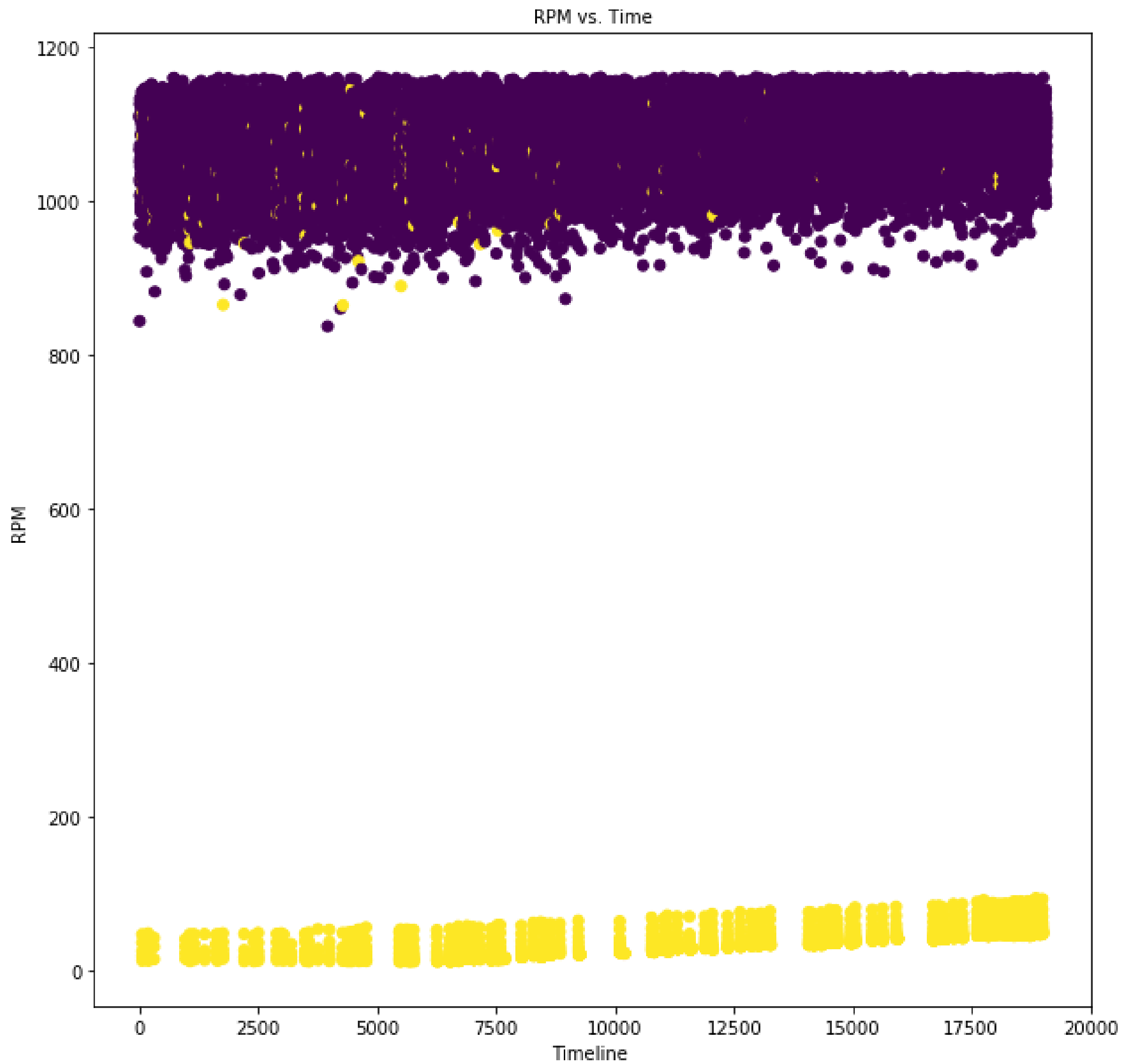
In [18]:
```
result.shape
```

Out[18]:
```
(775374, 9)
```

In [384]:
```
#Sensor 19 which had the highest working life
def plot_clusters(data, col):
    """
    Helper function to cluster based on predictions and facet based on t
he labels.
    This will help us to clear infer if there are clear distinctions in
 the data
    """
    plt.figure(figsize=(10,10))
    cols = ['rpm','motor_voltage','motor_current','motor_temp','inlet_te
mp','power','temp_diff']
    model = KMeans(n_clusters=2)
    model.fit(data[cols],data[col])
    plt.xlabel('Timeline ',fontsize = 10)
    plt.ylabel('{}'.format(col.upper()),fontsize = 10)
    plt.title('{} vs. Time'.format(col.upper()),fontsize = 10)
    return plt.scatter(range(len(data['rpm'])),data[col],c=model.labels_
)
```

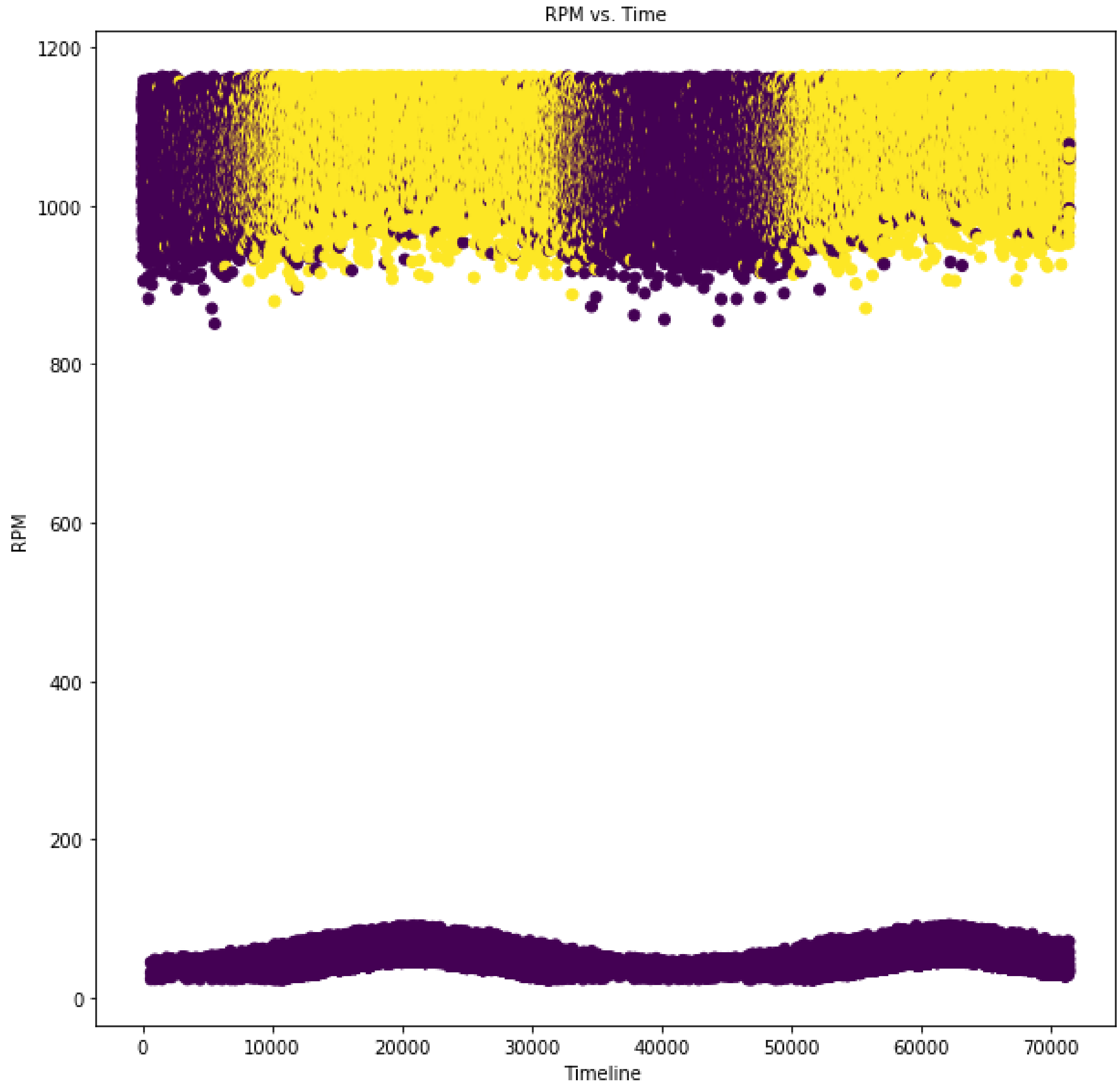
In [385]:
```
plot_clusters(sensors[19].noise_free_rms_processed, 'rpm')
```

Out[385]: <matplotlib.collections.PathCollection at 0x16a213208>



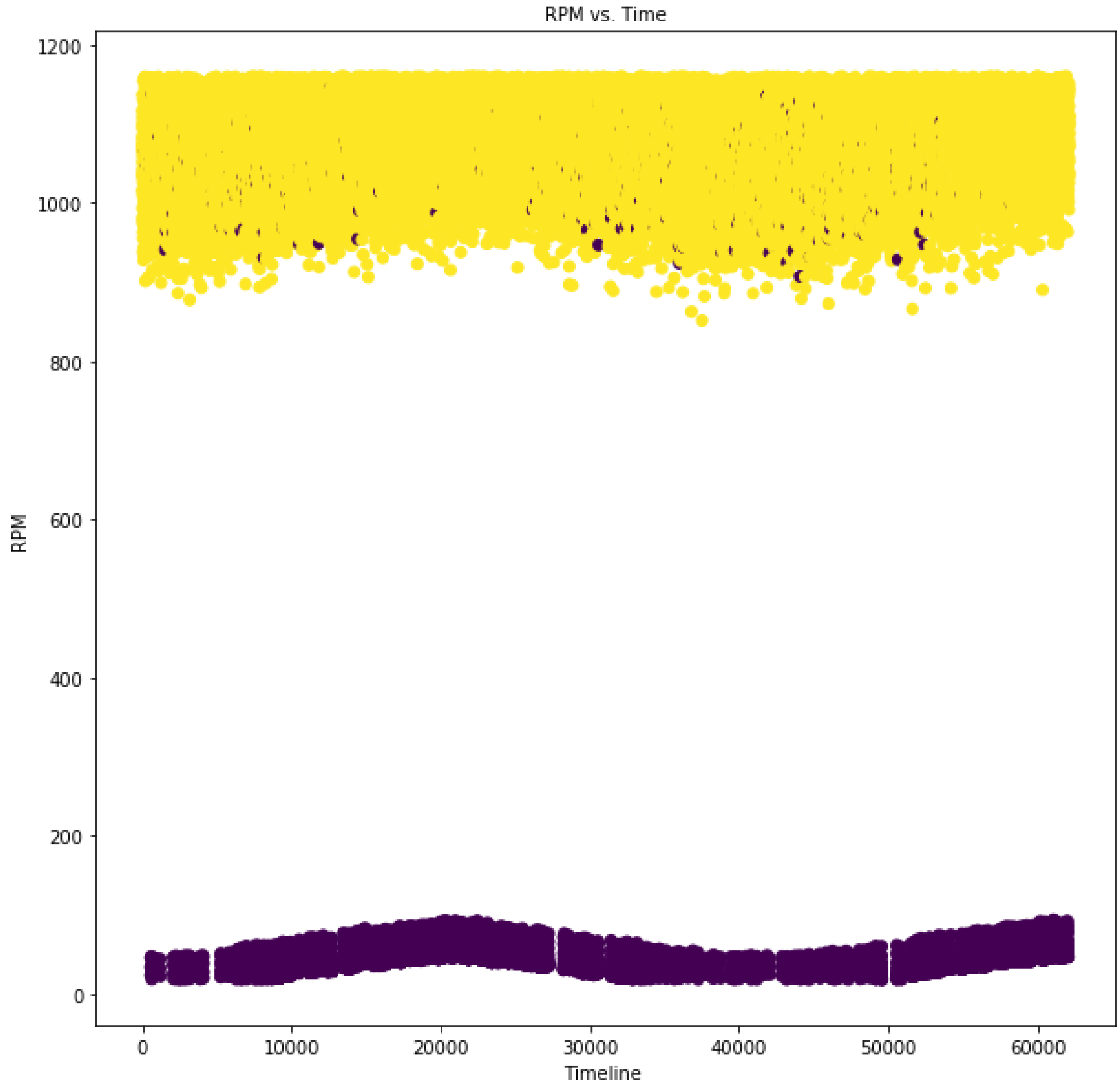
In [386]:
```
plot_clusters(sensors[0].noise_free_rms_processed, 'rpm')
```

Out[386]: <matplotlib.collections.PathCollection at 0x16a206320>



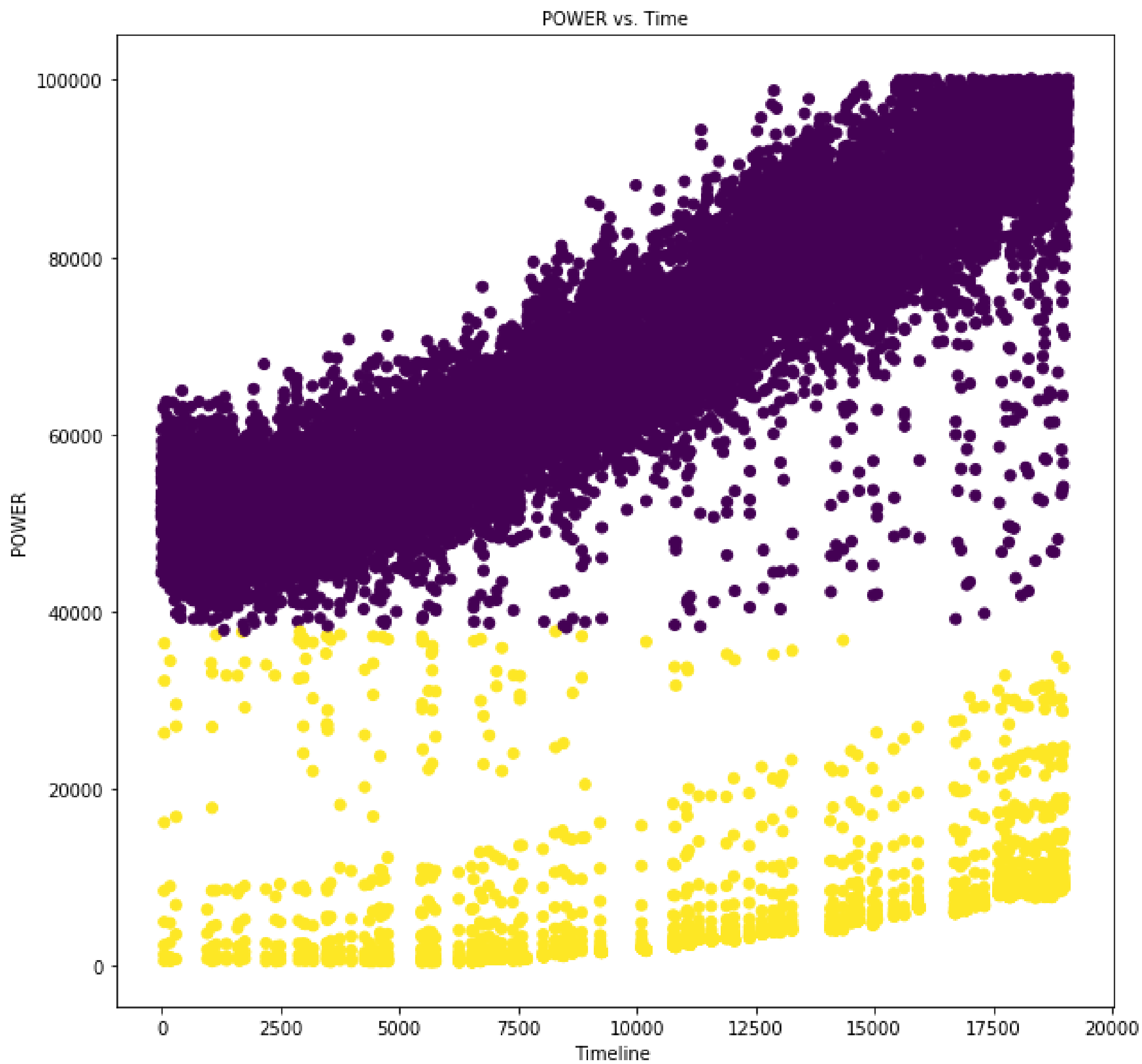
In [387]:
```
plot_clusters(sensors[12].noise_free_rms_processed, 'rpm')
```

Out[387]:
```
<matplotlib.collections.PathCollection at 0x16d05c0f0>
```



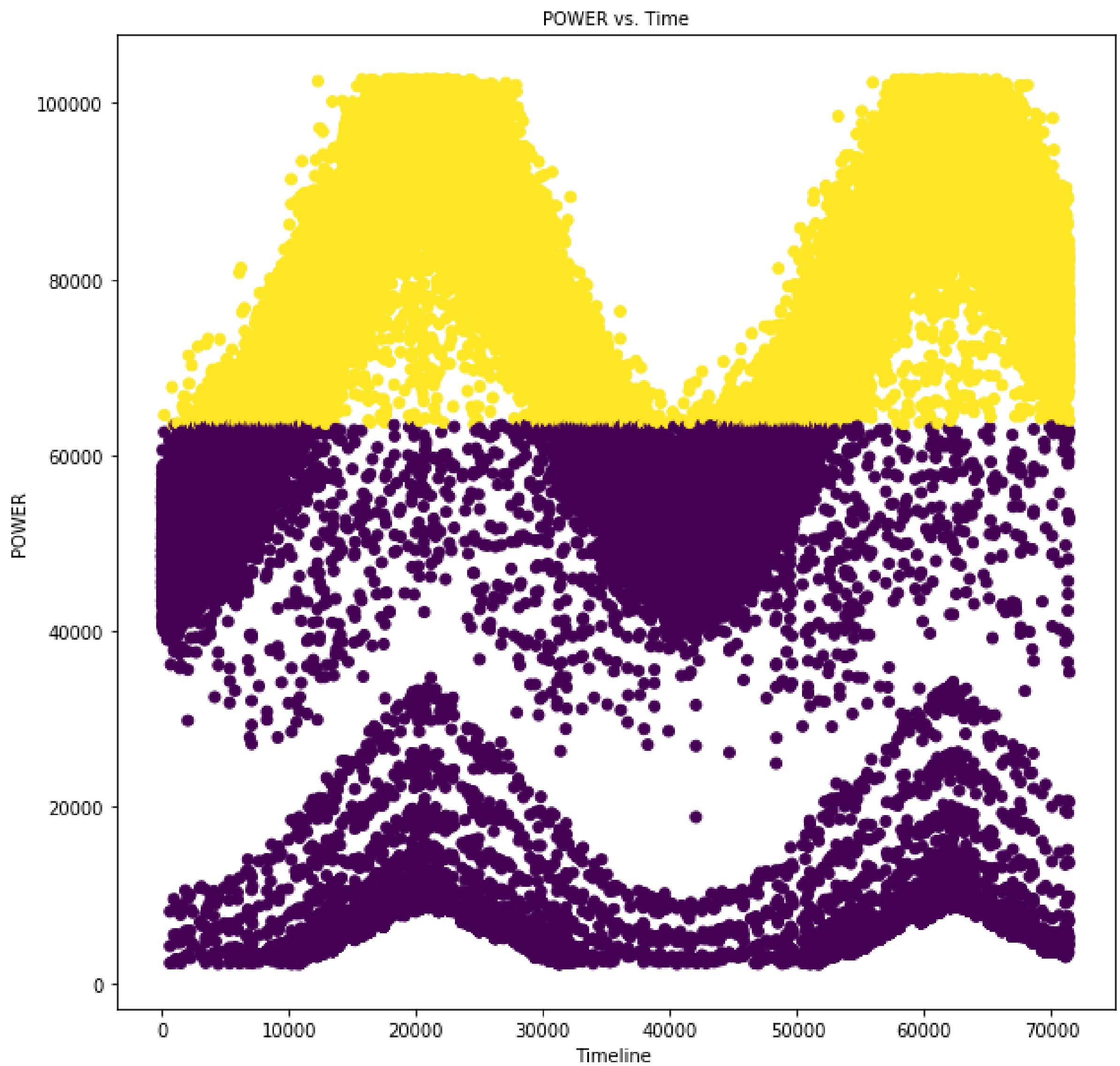
In [388]:
```
plot_clusters(sensors[19].noise_free_rms_processed, 'power')
```

Out[388]:
```
<matplotlib.collections.PathCollection at 0x16d36fc50>
```



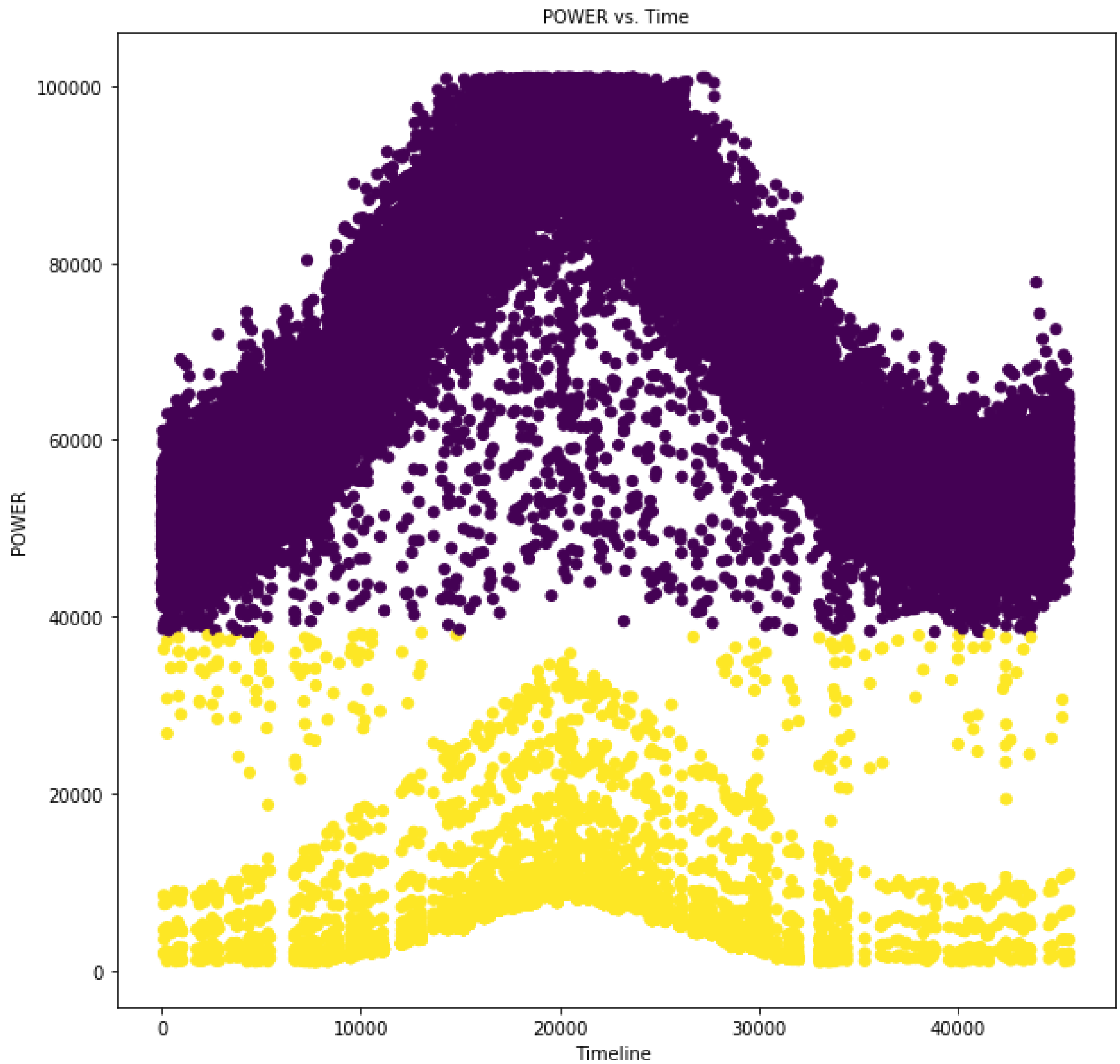
In [389]:
```
plot_clusters(sensors[0].noise_free_rms_processed, 'power')
```

Out[389]:
```
<matplotlib.collections.PathCollection at 0x16dcbfe48>
```



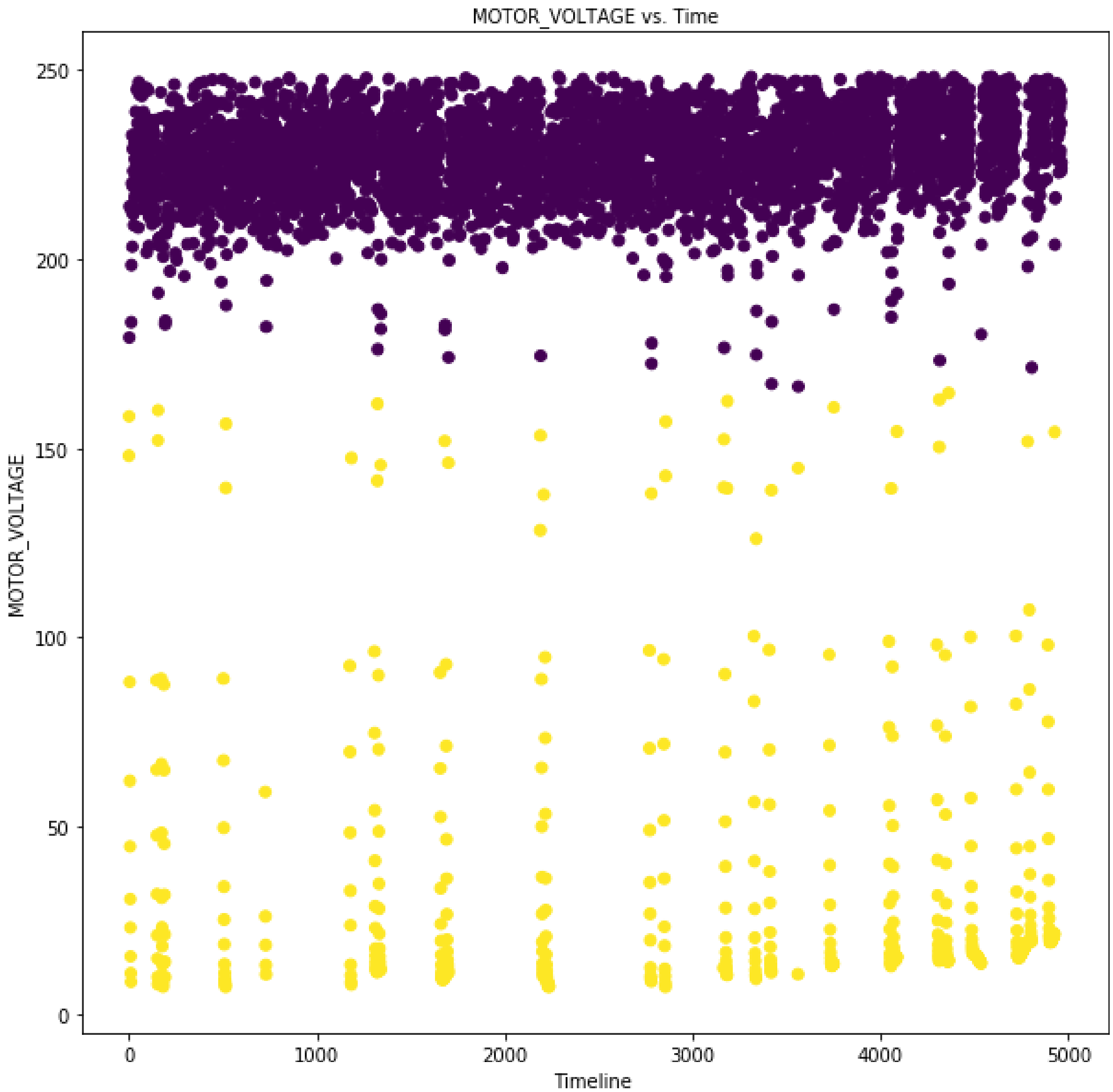
In [390]:
```
plot_clusters(sensors[11].noise_free_rms_processed, 'power')
```

Out[390]: <matplotlib.collections.PathCollection at 0x170284da0>



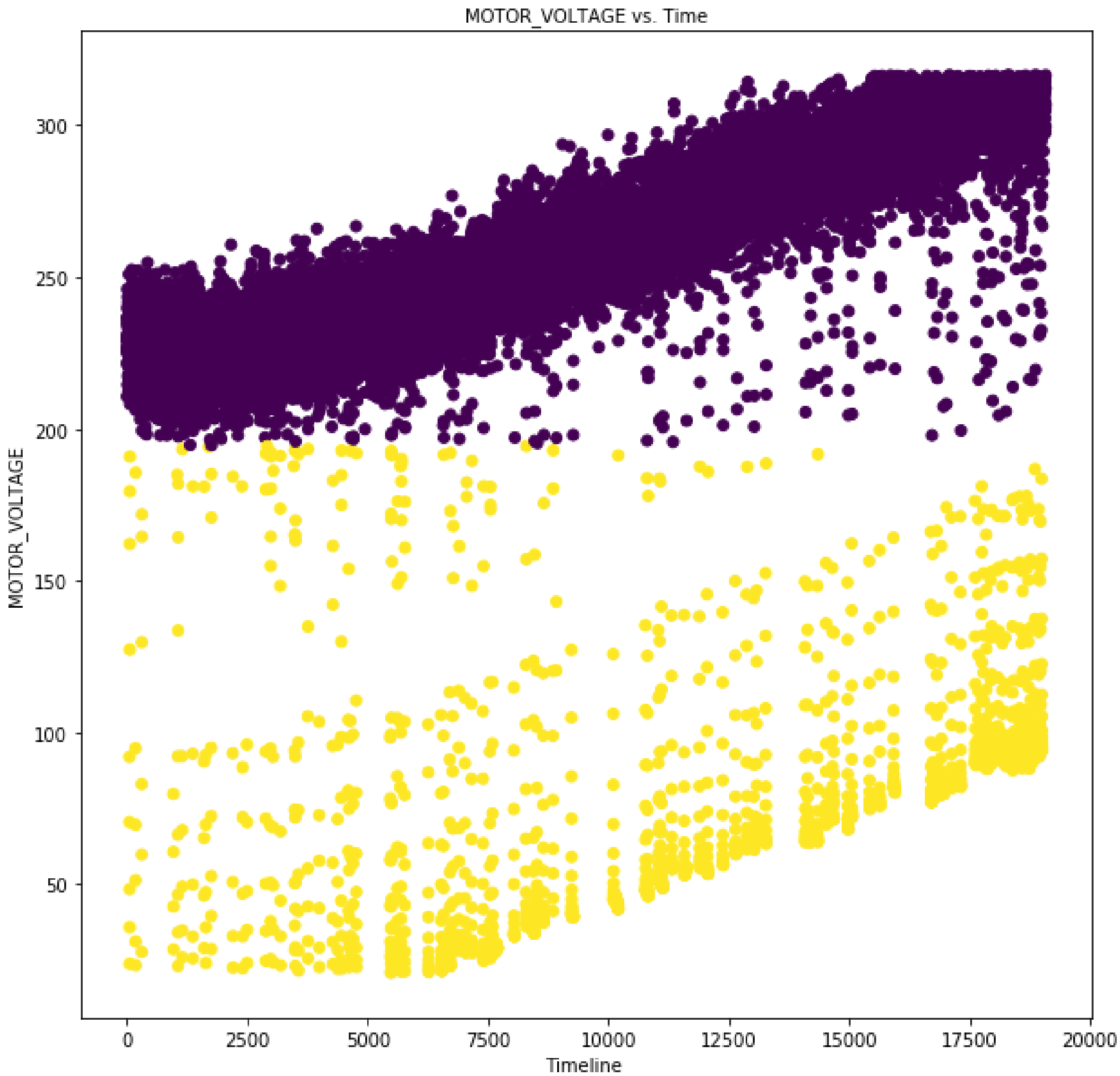
In [391]:
```
plot_clusters(sensors[15].noise_free_rms_processed, 'motor_voltage')
```

Out[391]:
```
<matplotlib.collections.PathCollection at 0x1717e2748>
```



In [392]:
```
plot_clusters(sensors[19].noise_free_rms_processed, 'motor_voltage')
```

Out[392]: <matplotlib.collections.PathCollection at 0x16e6ab978>



3D plots for interactive graph that show clear clustering

In [393]:
```
def create_3d_plot(x, y, z):
    trace1 = go.Scatter3d(x=x,y=y,z=z,mode='markers',marker=dict(size=12
,colorscale='Viridis',opacity=0.8))
    data = [trace1]
    layout = go.Layout(margin=dict(l=0,r=0,b=0,t=0))
    fig = go.Figure(data=data, layout=layout)
    return py.iplot(fig, filename='3d-scatter-colorscale')

create_3d_plot(x=temp_df['rpm'],y=temp_df['power'],z=temp_df['temp_diff'
])
```

Out[393]:

In [394]:
```
create_3d_plot(x=temp_df['rpm'],y=temp_df['motor_current'],z=temp_df['temp_diff'])
```

Out[394]:

Survival modeling

Given that we're required to find an event where the sensor might fail, survival analysis seems to be a good avenue to explore

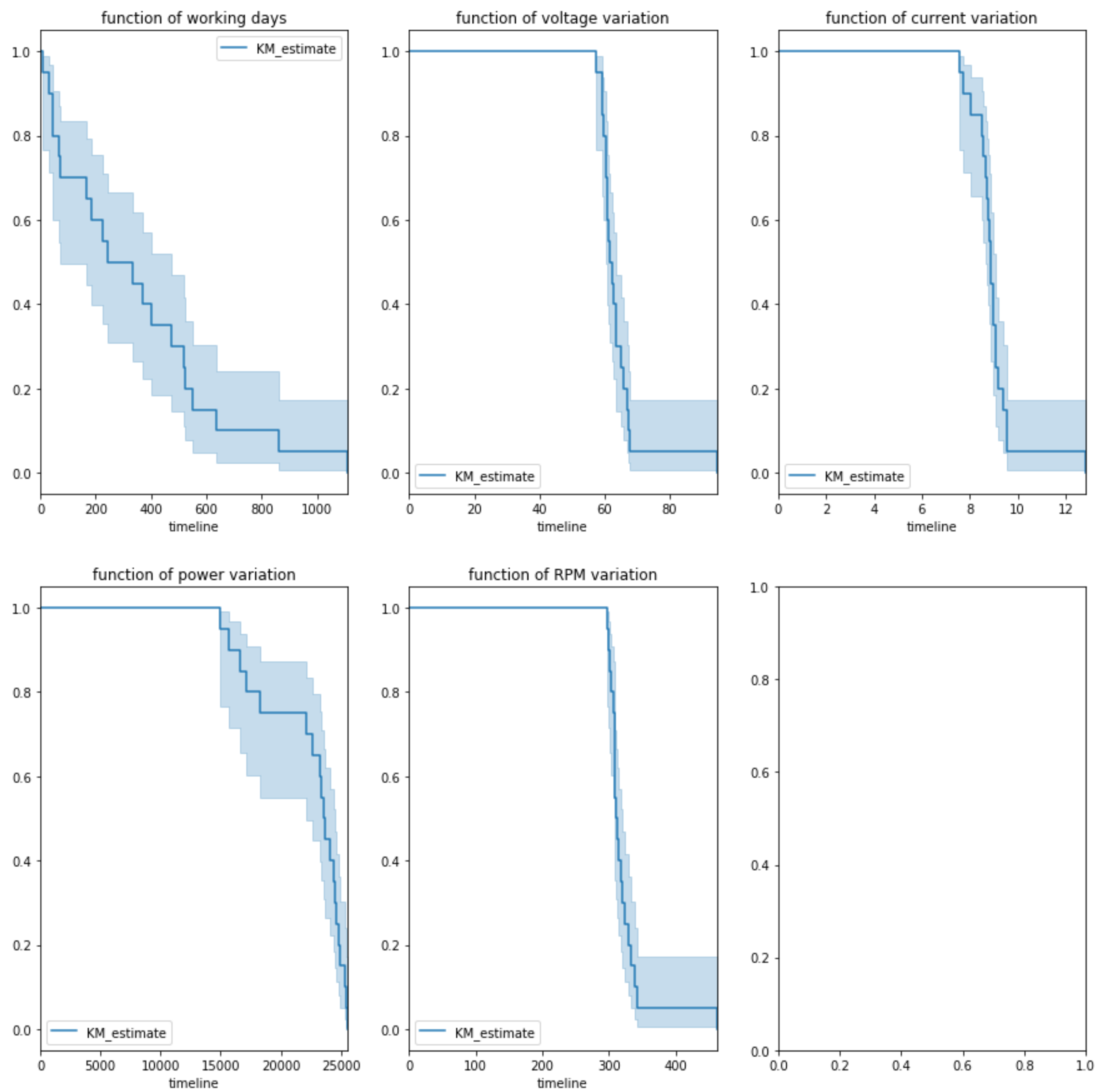
References:

https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html (https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html) http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf (http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf) https://towardsdatascience.com/survival-analysis-intuition-implementation-in-python-504fde4fcf8e (https://towardsdatascience.com/survival-analysis-intuition-implementation-in-python-504fde4fcf8e)

In [131]:
```
def get_working_days(sensors):
    """
    Extracting params on which we're interested to perform survival anal
ysis
    """
    working_life = []
    v_std, i_std, p_std, rpm_std = [], [], [], []
    for each_s in sensors:
        desc = each_s.noise_free_rms_processed.describe()
        working_life.append(
            (each_s.noise_free_rms_processed['timestamp'].max()-
             each_s.noise_free_rms_processed['timestamp'].min()).days)
        v_std.append(desc['motor_voltage']['std'])
        i_std.append(desc['motor_current']['std'])
        p_std.append(desc['power']['std'])
        rpm_std.append(desc['rpm']['std'])
    return working_life, v_std, i_std, p_std, rpm_std
sensor_life, v_std, i_std, p_std, rpm_std = get_working_days(sensors)
```

In [146]:

```
#https://lifelines.readthedocs.io/en/latest/lifelines.fitters.html?highl
ight=KaplanMeierFitter#module-lifelines.fitters.kaplan_meier_fitter
from lifelines import KaplanMeierFitter
fig, axes = plt.subplots(2, 3, figsize=(15, 15))
axes = axes.reshape(6,)
titles = ["function of working days",
          "function of voltage variation",
          "function of current variation",
          "function of power variation",
          "function of RPM variation"]
for i, model in enumerate([sensor_life, v_std, i_std, p_std, rpm_std]):
    axes[i].set_title(titles[i])
    KaplanMeierFitter(alpha=0.1).fit(durations = model, event_observed =
[1]*len(model)).plot(ax=axes[i],)
plt.show()
```



Observations -

Figure 1 - as a function of working days: It is clear that the survival function has a negative relation as working life increases. if the motor is old more chances it'll fail.

Figure 2 - as a function of voltage variation: Improper input to an electric motor can have ramifications on its output and lifecycle. The graph shows that any voltage variation more tha 60V(units in micro, milli, kilo or Mega) tends to decrease the lifecycle of the motor

Figure 3 - as a function of current variation: Similarly like motor voltage, 8 or more units(milli, micro amps) variation from its mean value would not be ideal.

Figure 4 - as a function of power: This is trivial given above two points as P = VI

Figure 5 - as a function of RPM: RPM is the ouput we're able to measure given the inputs. It's ideal that the rpm doesnt wildly fluctuate. We don't not have much control over this as the line voltage can fluctuate randomly. One suggestion would to regulate the voltage/current before it's fed to the motor. However, in a pratical/ industrial setting(assumption) this will be taken care of. The only other thing that would cause our RPM to deviate from it's intended level would be exogenous noises or error in the measurement itself.
As per the survival function, +- 300revs/m tends to decrease the lifecycle of the sensor.

We can also investigate hazard function as The hazard function, used for regression in survivalanalysis, can lend more insight into the failure mechanism

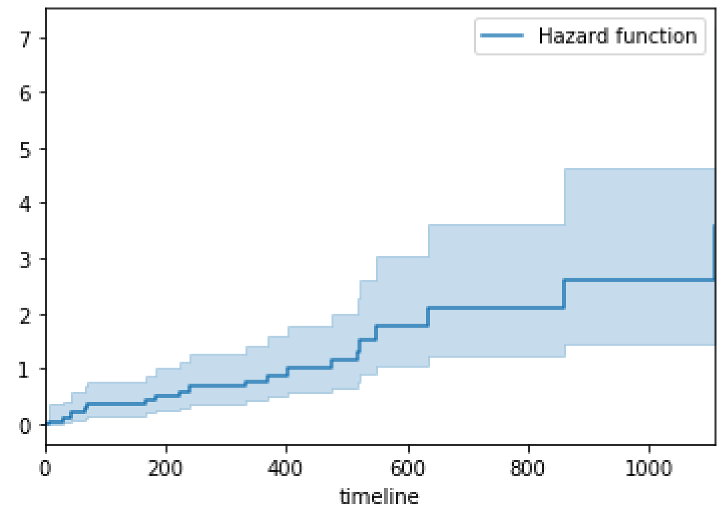
Reference:

[http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf (http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf)](http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf)

[https://lifelines.readthedocs.io/en/latest/lifelines.fitters.html?highlight=KaplanMeierFitter#module-lifelines.fitters.nelson_aalen_fitter (https://lifelines.readthedocs.io/en/latest/lifelines.fitters.html?highlight=KaplanMeierFitter#module-lifelines.fitters.nelson_aalen_fitter)](https://lifelines.readthedocs.io/en/latest/lifelines.fitters.html?highlight=KaplanMeierFitter#module-lifelines.fitters.nelson_aalen_fitter)

In [139]:
```
from lifelines import NelsonAalenFitter

fitter = NelsonAalenFitter()
fitter.fit(sensor_life, event_observed=[1]*len(sensor_life), label="Haza
rd function")
fitter.plot(show_censors=True)
```

Out[139]: <matplotlib.axes._subplots.AxesSubplot at 0x1596f7ba8>



ARIMA Approach to forecast motor failure mainly forecasting for the rpm signal

In [79]:
```
def evaluate_arima_model(X, arima_order):
    """
    ARIMA model
    """
#     import ipdb;ipdb.set_trace()
    train_size = int(len(X) * 0.66)
    train, test = train_test_split(X, test_size=0.34)
    history = [x for x in train]
    predictions = list()
    for t in range(len(test)):
        model = ARIMA(history, order=arima_order)
        model_fit = model.fit(disp=0)
        yhat = model_fit.forecast()[0]
        predictions.append(yhat)
        history.append(test.iloc[t])
    error = mean_squared_error(test, predictions)
    return error

# evaluate combinations of p, d and q values for an ARIMA model
def evaluate_models(dataset, p_values, d_values, q_values):
    """
    Crude grid search method to find p,d,q values
    """
    best_score, best_cfg = float("inf"), None
    for p in p_values:
        for d in d_values:
            for q in q_values:
                order = (p,d,q)
                try:
                    mse = evaluate_arima_model(dataset, order)
                    if mse < best_score:
                        best_score, best_cfg = mse, order
                        print('ARIMA%s MSE=%.3f' % (order,mse))
                except Exception as e:
                    print(e.args)
                    continue
    print('Best ARIMA%s MSE=%.3f' % (best_cfg, best_score))
# evaluate parameters
p_values = [0,1,2, 3, 4, 5]
d_values = range(0, 4)
q_values = range(0, 4)
```

In [80]:
```
warnings.filterwarnings("ignore")
evaluate_models(sensors[0].noise_free_rms_processed['rpm'][:50], p_value
s, d_values, q_values)
```

```
(0, 0, 0)
ARIMA(0, 0, 0) MSE=2643.894
(0, 0, 1)
(0, 0, 2)
(0, 0, 3)
(0, 1, 0)
(0, 1, 1)
(0, 1, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(0, 1, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(0, 2, 0)
(0, 2, 1)
(0, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(0, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(0, 3, 0)
('d > 2 is not supported',)
(0, 3, 1)
('d > 2 is not supported',)
(0, 3, 2)
('d > 2 is not supported',)
(0, 3, 3)
('d > 2 is not supported',)
(1, 0, 0)
ARIMA(1, 0, 0) MSE=2550.884
(1, 0, 1)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(1, 0, 2)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(1, 0, 3)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(1, 1, 0)
(1, 1, 1)
(1, 1, 2)
(1, 1, 3)
ARIMA(1, 1, 3) MSE=2387.909
(1, 2, 0)
(1, 2, 1)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
```

```
(1, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(1, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(1, 3, 0)
('d > 2 is not supported',)
(1, 3, 1)
('d > 2 is not supported',)
(1, 3, 2)
('d > 2 is not supported',)
(1, 3, 3)
('d > 2 is not supported',)
(2, 0, 0)
(2, 0, 1)
(2, 0, 2)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(2, 0, 3)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(2, 1, 0)
(2, 1, 1)
(2, 1, 2)
(2, 1, 3)
("Input contains NaN, infinity or a value too large for dtype('float6
4').",)
(2, 2, 0)
(2, 2, 1)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(2, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(2, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(2, 3, 0)
('d > 2 is not supported',)
(2, 3, 1)
('d > 2 is not supported',)
(2, 3, 2)
('d > 2 is not supported',)
(2, 3, 3)
('d > 2 is not supported',)
(3, 0, 0)
(3, 0, 1)
(3, 0, 2)
(3, 0, 3)
```

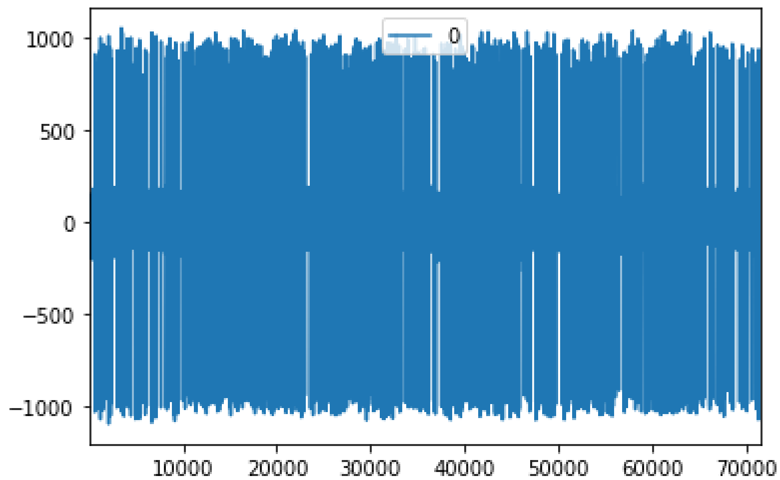
```
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(3, 1, 0)
(3, 1, 1)
(3, 1, 2)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(3, 1, 3)
('SVD did not converge',)
(3, 2, 0)
(3, 2, 1)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(3, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(3, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(3, 3, 0)
('d > 2 is not supported',)
(3, 3, 1)
('d > 2 is not supported',)
(3, 3, 2)
('d > 2 is not supported',)
(3, 3, 3)
('d > 2 is not supported',)
(4, 0, 0)
(4, 0, 1)
(4, 0, 2)
(4, 0, 3)
("Input contains NaN, infinity or a value too large for dtype('float6
4').",)
(4, 1, 0)
(4, 1, 1)
(4, 1, 2)
(4, 1, 3)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(4, 2, 0)
(4, 2, 1)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(4, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(4, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
```

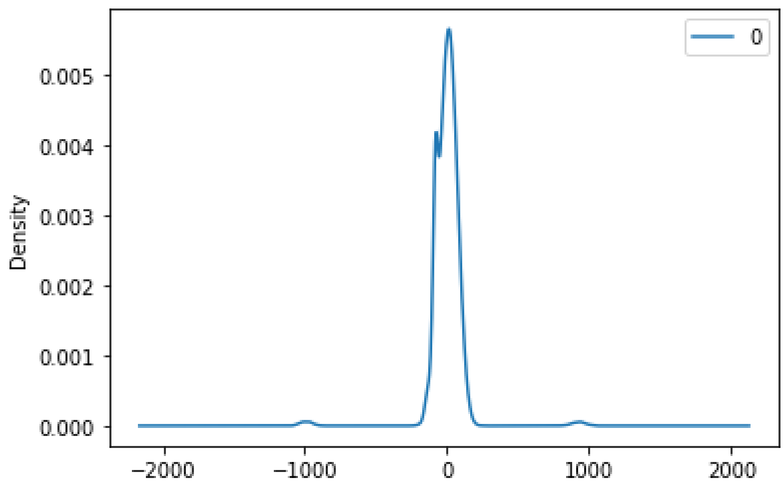
```
our own start_params.',)
(4, 3, 0)
('d > 2 is not supported',)
(4, 3, 1)
('d > 2 is not supported',)
(4, 3, 2)
('d > 2 is not supported',)
(4, 3, 3)
('d > 2 is not supported',)
(5, 0, 0)
(5, 0, 1)
(5, 0, 2)
(5, 0, 3)
("Input contains NaN, infinity or a value too large for dtype('float6
4').",)
(5, 1, 0)
(5, 1, 1)
(5, 1, 2)
('The computed initial AR coefficients are not stationary\nYou should i
nduce stationarity, choose a different model order, or you can\npass yo
ur own start_params.',)
(5, 1, 3)
(5, 2, 0)
(5, 2, 1)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(5, 2, 2)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(5, 2, 3)
('The computed initial MA coefficients are not invertible\nYou should i
nduce invertibility, choose a different model order, or you can\npass y
our own start_params.',)
(5, 3, 0)
('d > 2 is not supported',)
(5, 3, 1)
('d > 2 is not supported',)
(5, 3, 2)
('d > 2 is not supported',)
(5, 3, 3)
('d > 2 is not supported',)
Best ARIMA(1, 1, 3) MSE=2387.909
```

Evaluate ARIMA's performance

In [566]:
```
cols = ['rpm', 'motor_voltage', 'motor_current', 'motor_temp','inlet_tem
p',  'power', 'temp_diff']
data = pd.Series(sensors[0].noise_free_rms_processed['rpm'])
model = ARIMA(data,order=(1, 1, 3))
model_fit = model.fit(disp=0)
print(model_fit.summary())
# plot residual errors
residuals = pd.DataFrame(model_fit.resid)
residuals.plot()
plt.show()
residuals.plot(kind='kde')
plt.show()
print(residuals.describe())
```

```
                                 ARIMA Model Results
==============================================================================
=======
Dep. Variable:                  D.rpm   No. Observations:
71406
Model:                 ARIMA(1, 1, 3)   Log Likelihood                  -448
691.357
Method:                       css-mle   S.D. of innovations
129.632
Date:                Sun, 21 Jul 2019   AIC                              897
394.713
Time:                        15:23:31   BIC                              897
449.770
Sample:                             1   HQIC                             897
411.678

==============================================================================
========
                  coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
--------
const          -0.0006      0.005     -0.115      0.909      -0.010
0.009
ar.L1.D.rpm     0.9067      0.002    450.961      0.000       0.903
0.911
ma.L1.D.rpm    -1.0505      0.004   -251.217      0.000      -1.059
-1.042
ma.L2.D.rpm     0.0792      0.005     15.352      0.000       0.069
0.089
ma.L3.D.rpm    -0.0278      0.004     -6.712      0.000      -0.036
-0.020
                                    Roots
==============================================================================
======
                  Real          Imaginary           Modulus          Fre
quency
------------------------------------------------------------------------------
------
AR.1            1.1029           +0.0000j            1.1029
0.0000
MA.1            1.0010           -0.0000j            1.0010            -
0.0000
MA.2            0.9244           -5.9222j            5.9939            -
0.2254
MA.3            0.9244           +5.9222j            5.9939
0.2254
------------------------------------------------------------------------------
------
```



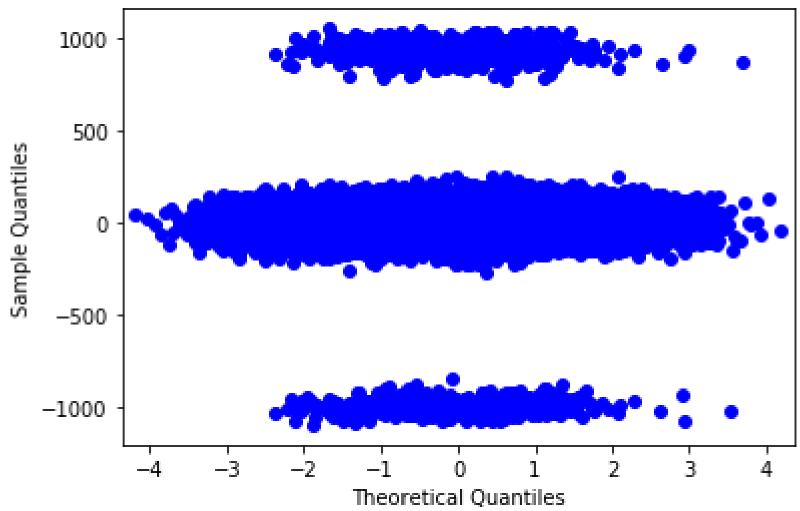


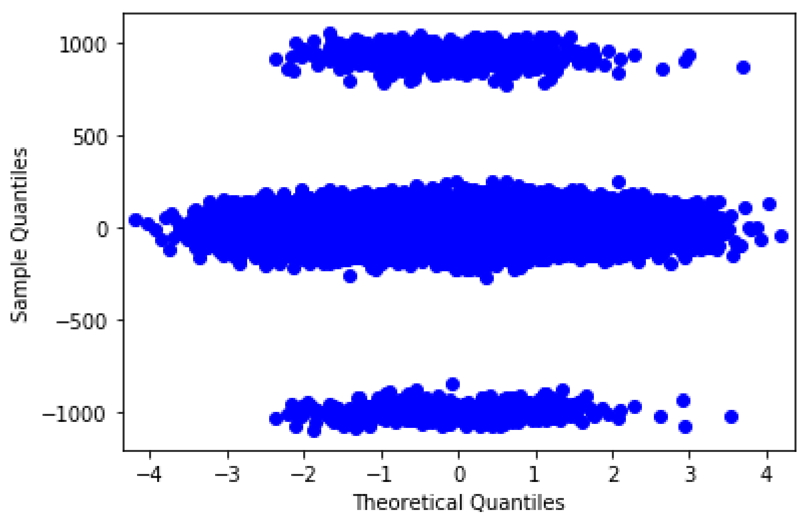

```
                  0
count  71406.000000
mean      -0.010540
std      129.634389
min    -1093.120322
25%      -49.303901
50%        2.156555
75%       47.047988
max     1051.594558
```

Residuals clearly show that our model did not quite capture the underlying relationship as the residuals are not random and still show the trend of our input. MSE wouldn't tell us completely if ourn model is a good fit. We need to evaluate using qqplot

In [567]:
```
qqplot(residuals)
```

Out[567]:

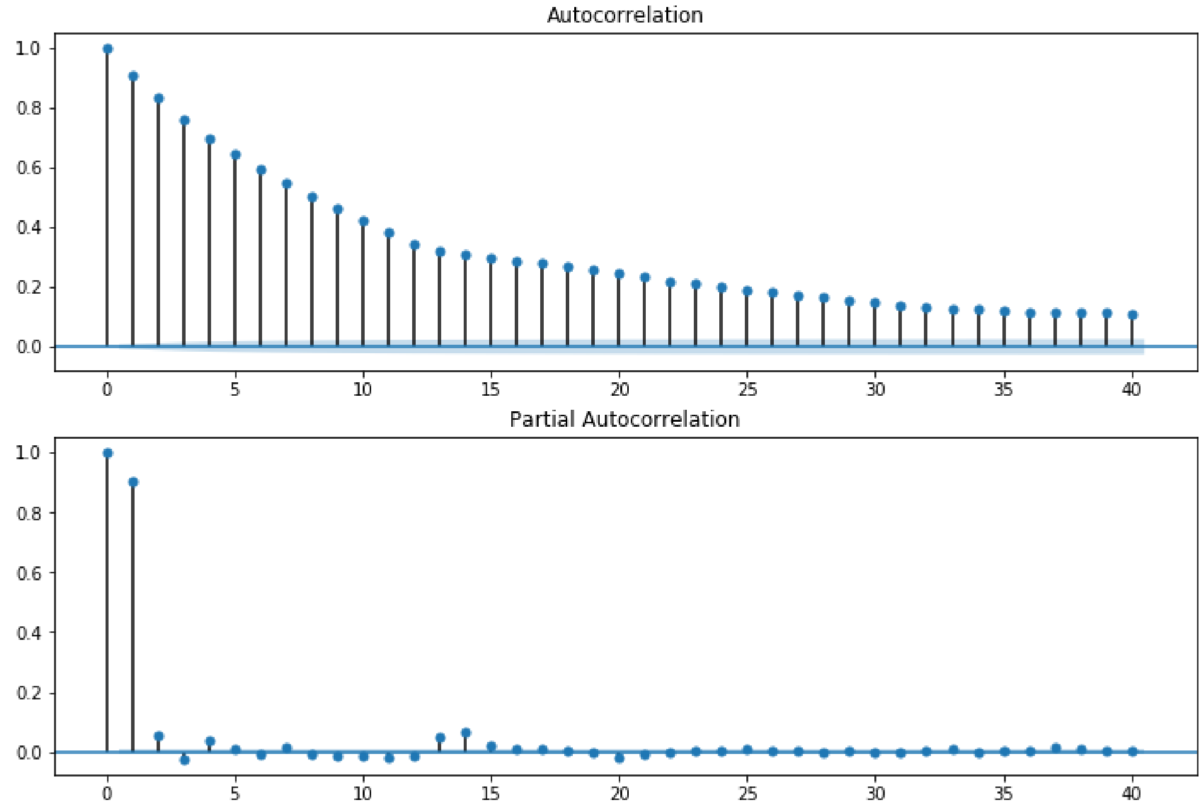




Since qqplot is not normal/linear in nature, this confirms ARIMA isn't a good fit or we need to optimize our p,d,q values more.

Exploring Auto correlation and partial auto correlation in RPM as it can identify the extent of the lag in the signal and help us predicting the future value of RPM in the given autoregressive model. Reference: [https://www.mathworks.com/help/econ/autocorrelation-and-partial-autocorrelation.html (https://www.mathworks.com/help/econ/autocorrelation-and-partial-autocorrelation.html)](https://www.mathworks.com/help/econ/autocorrelation-and-partial-autocorrelation.html)

In [207]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(sensors[0].noise_free_rms['rpm'].squeeze
(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(sensors[0].noise_free_rms['rpm'], lags=4
0, ax=ax2)
```



In [208]:
```
#Baseline model for ARMA
arma_mod20 = sm.tsa.ARMA(sensors[0].noise_free_rms['rpm'], (2,0)).fit(di
sp=False)
print(arma_mod20.params)
```

```
const         968.202945
ar.L1.rpm       0.854482
ar.L2.rpm       0.057416
dtype: float64
```

In [209]:
```
#Increasing the variance by accounting for higher order terms
arma_mod30 = sm.tsa.ARMA(sensors[0].noise_free_rms['rpm'], (3,0)).fit(di
sp=False)
print(arma_mod30.params)
```

```
const         968.202945
ar.L1.rpm       0.855840
ar.L2.rpm       0.077630
ar.L3.rpm      -0.023656
dtype: float64
```

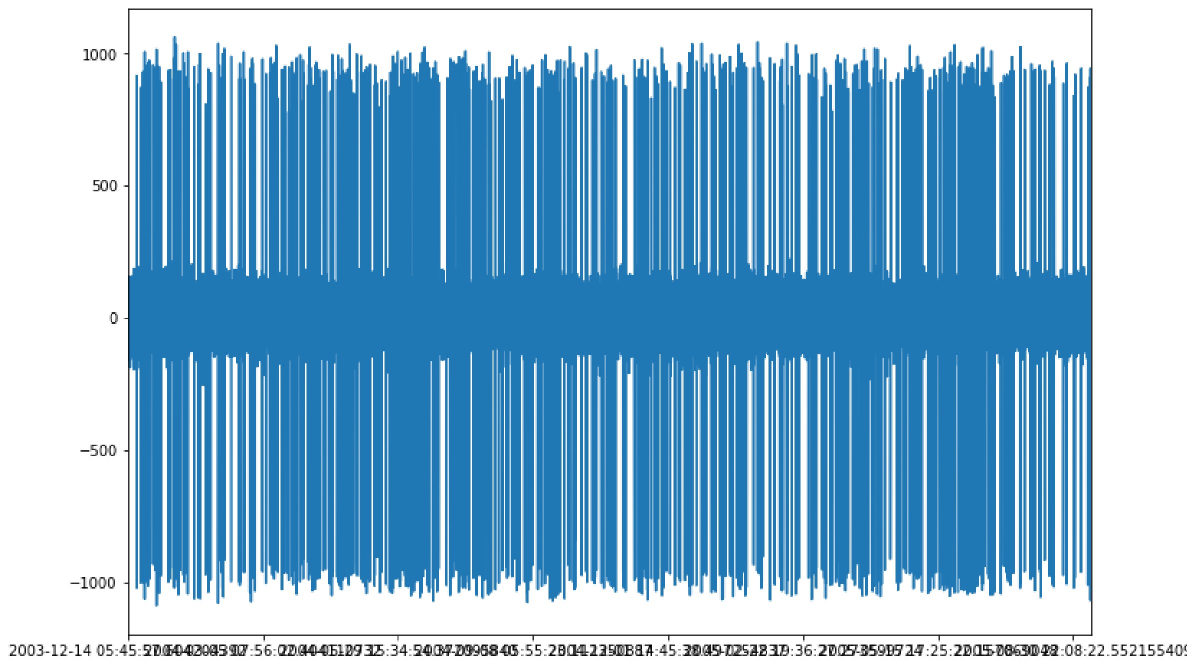
In [210]:
```
print("--- arma_mod20 models params --")
print(arma_mod20.aic, arma_mod20.bic, arma_mod20.hqic)
print("=============================")
print("--- arma_mod30 models params --")
print(arma_mod30.aic, arma_mod30.bic, arma_mod30.hqic)
print("=============================")
```

```
--- arma_mod20 models params --
897537.0276610606 897573.7322657915 897548.337918255
=============================
--- arma_mod30 models params --
897499.0554035543 897544.936159468 897513.1932250474
=============================
```

In [212]:
```
sm.stats.durbin_watson(arma_mod30.resid.values)
```

Out[212]: 1.9981344724949541

In [213]:
```
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)
ax = arma_mod30.resid.plot(ax=ax)
```

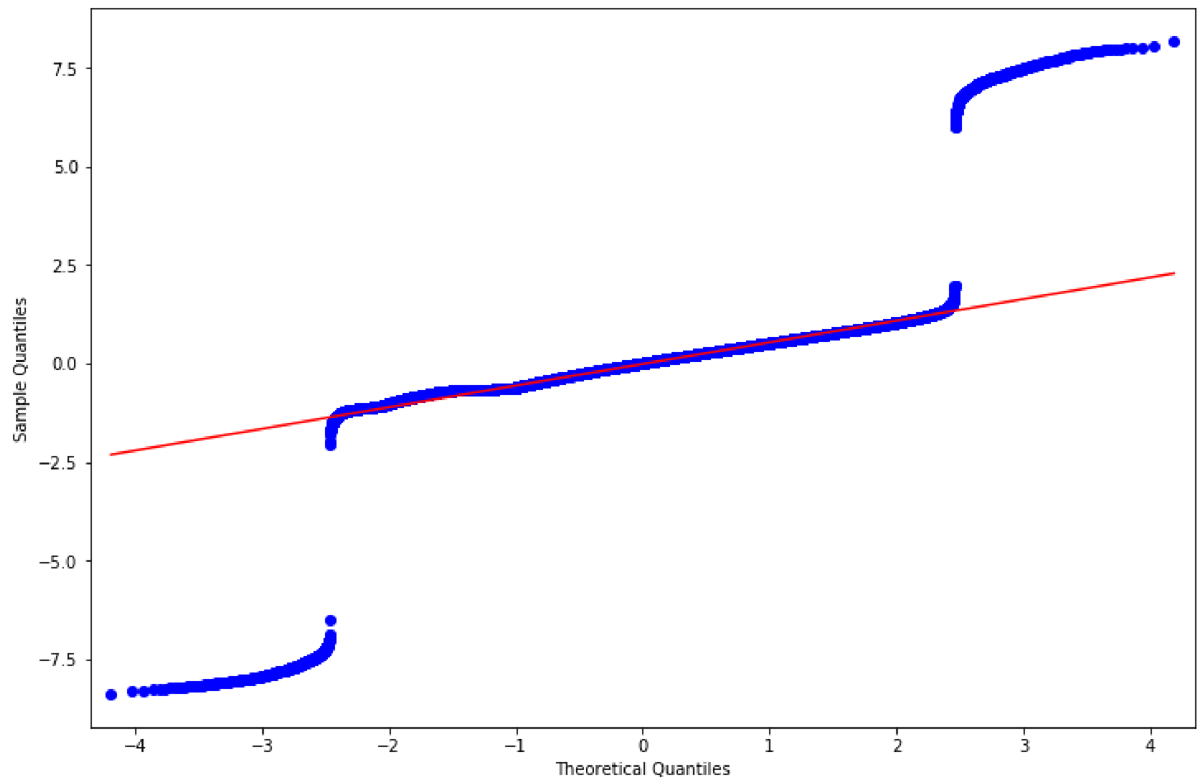


Residual plot doesn't give much information about how good our model is as the residual plot is pretty hard to understand. Let's plot QQplot to understand more

In [214]:
```
resid = arma_mod30.resid
stats.normaltest(resid)
```

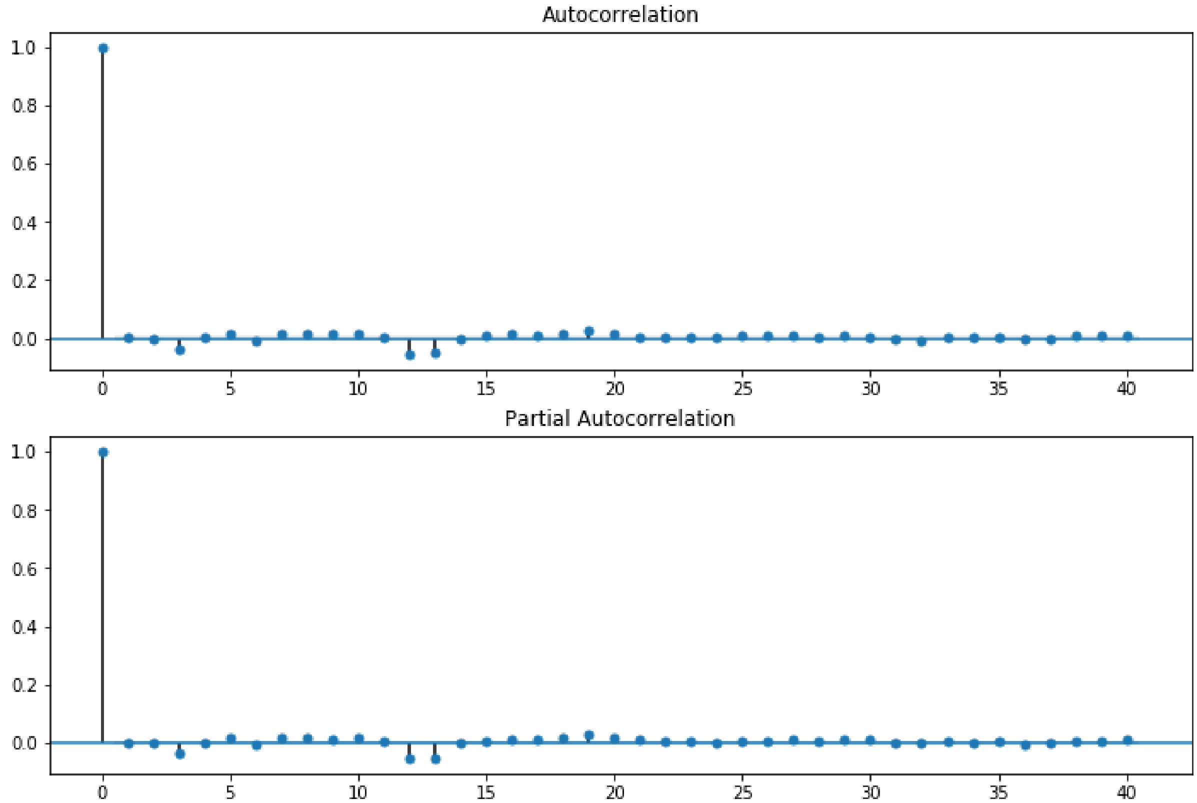
Out[214]: NormaltestResult(statistic=24960.880538976213, pvalue=0.0)

In [215]:
```
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)
fig = qqplot(resid, line='q', ax=ax, fit=True)
```



The above QQPlot is much better than the ARIMA model. We've already seen the different clusters at the exteremes before in our EDA. We can fix up a logistic regression model to account for the outliers we're seeing at the extreme ends. The majority of our data are in good accord with the quantiles presented by the graph

In [216]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(resid.values.squeeze(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(resid, lags=40, ax=ax2)
```



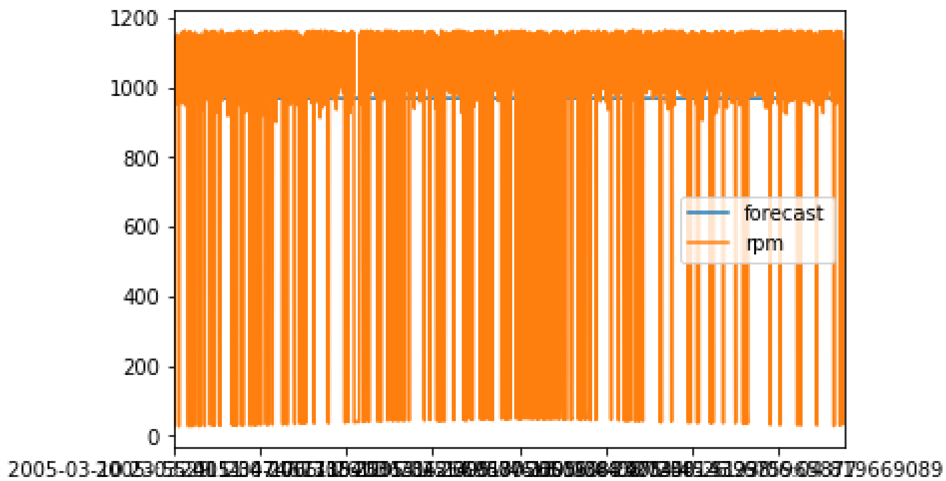
In [652]:
```
#OBTAINING PREDICTIONS FROM THE ARMA MODEL: Incomplete at the moment as
 I needed more time to research and debug more
# on this. ARMA model is not accepting any index values outside of the v
alues used in training. This sort of defeats the
# purpose of forecasting.

# fig, ax = plt.subplots(figsize=(12, 8))
# ax = sensors[0].noise_free_rms.loc['2005-09-10 17:36:47.236452969':].p
lot(ax=ax)
# fig = arma_mod30.plot_predict(start = '2005-09-10 17:36:47.236452969',
dynamic=True, ax=ax, plot_insample=True)
# arma_mod30.predict(start='2005-09-10 17:36:47.236452969', end = '2005-
09-11')
```

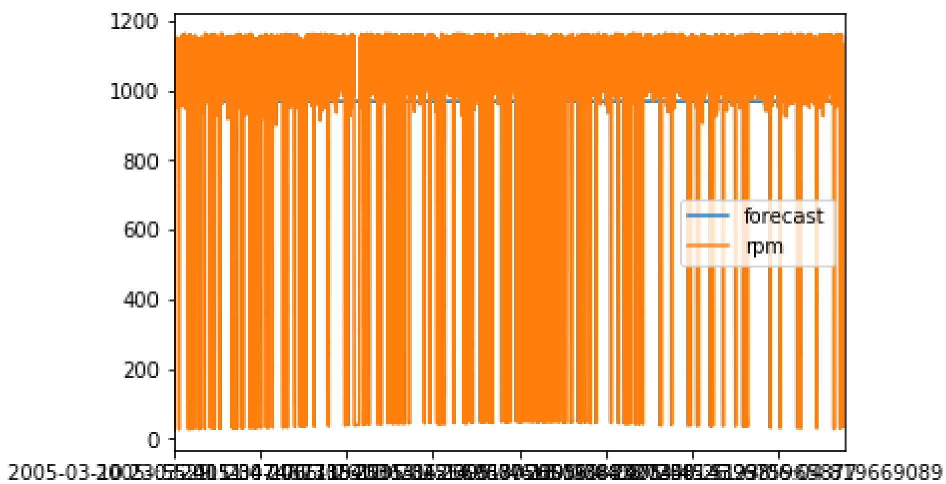
In [263]:
```
def mean_forecast_err(y, yhat):
    return y.sub(yhat).mean()
mean_forecast_err(sensors[0].noise_free_rms['rpm'], predict_sunspots)
```

Out[263]: 0.0013226780413007394

In [262]:
```
arma_mod30.plot_predict(start='2005-03-10 23:55:41.234746771', dynamic=True)
```

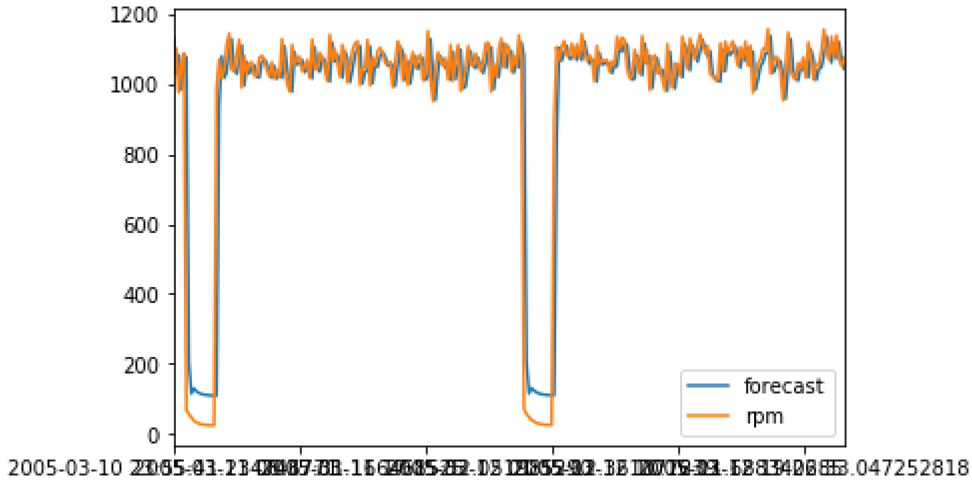
Out[262]:

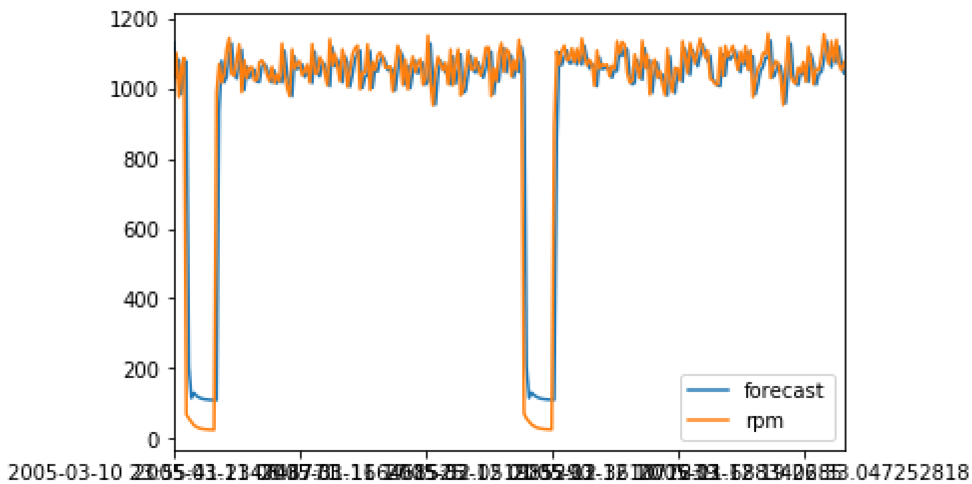




In [250]:
```
#Predicting values within a window. Predictions and forecast seem to be
 work well(Overfitted)
arma_mod30.plot_predict(start = '2005-03-10 23:55:41.234746771', end =
'2005-03-12 22:05:32.183090259')
```

Out[250]:





In [547]:
```
sensors[0].noise_free_rms_processed.columns
```

Out[547]:
```
Index(['index', 'rpm', 'motor_voltage', 'motor_current', 'motor_temp',
       'inlet_temp', 'timestamp', 'power', 'temp_diff'],
      dtype='object')
```

LSTM is another valid option for timeseries modeling(Given the nature of the data Sequential modeling might be helpful)

In [626]:
```
X = sensors[0].noise_free_rms_processed[['motor_voltage', 'motor_curren
t','motor_temp','inlet_temp']]
y = sensors[0].noise_free_rms_processed['rpm']
X_train, X_valid, y_train, y_valid = train_test_split(X,y, test_size =
0.20, random_state = 19)
```

In [627]:
```
def preprocess(X, y, window = 4):
    """
    Based on the theory discussed in the data generative process I'm
    trying to model by looking at a few reading in the past(window varia
ble)
    """
    pred, target = [], []
    for i in range(window, len(X)):
        pred.append(X.iloc[i-window:i].values)
        target.append(y.iloc[i])
    X_p, y_p = np.array(pred), np.array(target)
    X_p = np.reshape(X_p, (X_p.shape[0], X_p.shape[1], len(X.columns)))
    return X_p, y_p
```

In [628]:
```
X_train, y_train = preprocess(X_train, y_train)
X_valid, y_valid = preprocess(X_valid, y_valid)
```

In [629]:
```
import keras
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import Dropout

model = Sequential()
model.add(LSTM(units=450, return_sequences=False, input_shape=(4, len(X.
columns))))
model.add(Dropout(rate =0.2))
model.add(Dense(1, activation='relu'))

model.compile(optimizer='adam', loss = 'mean_squared_error')
model.fit(X_train, y_train, epochs=100, batch_size=200)
```

```
Epoch 1/100
57121/57121 [==============================] - 24s 427us/step - loss: 9
66277.3052
Epoch 2/100
57121/57121 [==============================] - 21s 360us/step - loss: 8
72349.3085
Epoch 3/100
57121/57121 [==============================] - 21s 359us/step - loss: 7
87818.0308
Epoch 4/100
57121/57121 [==============================] - 21s 366us/step - loss: 7
10386.3449
Epoch 5/100
57121/57121 [==============================] - 21s 363us/step - loss: 6
39242.0313
Epoch 6/100
57121/57121 [==============================] - 21s 368us/step - loss: 5
73917.7848
Epoch 7/100
57121/57121 [==============================] - 22s 379us/step - loss: 5
14326.3787
Epoch 8/100
57121/57121 [==============================] - 21s 374us/step - loss: 4
59745.5922
Epoch 9/100
57121/57121 [==============================] - 21s 370us/step - loss: 4
09979.0790
Epoch 10/100
57121/57121 [==============================] - 21s 372us/step - loss: 3
64943.9944
Epoch 11/100
57121/57121 [==============================] - 22s 383us/step - loss: 3
24336.9610
Epoch 12/100
57121/57121 [==============================] - 21s 375us/step - loss: 2
87888.0755
Epoch 13/100
57121/57121 [==============================] - 23s 396us/step - loss: 2
55431.2369
Epoch 14/100
57121/57121 [==============================] - 22s 385us/step - loss: 2
26952.8845
Epoch 15/100
57121/57121 [==============================] - 22s 379us/step - loss: 2
01741.0955
Epoch 16/100
57121/57121 [==============================] - 22s 381us/step - loss: 1
79869.1692
Epoch 17/100
57121/57121 [==============================] - 22s 383us/step - loss: 1
61407.6340
Epoch 18/100
57121/57121 [==============================] - 23s 406us/step - loss: 1
45766.2743
Epoch 19/100
57121/57121 [==============================] - 22s 388us/step - loss: 1
32831.7361
```

```
Epoch 20/100
57121/57121 [==============================] - 22s 389us/step - loss: 1
22479.5173
Epoch 21/100
57121/57121 [==============================] - 22s 390us/step - loss: 1
14187.6864
Epoch 22/100
57121/57121 [==============================] - 23s 401us/step - loss: 1
07955.5054
Epoch 23/100
57121/57121 [==============================] - 23s 411us/step - loss: 1
03425.5918
Epoch 24/100
57121/57121 [==============================] - 23s 410us/step - loss: 9
9973.4889
Epoch 25/100
57121/57121 [==============================] - 23s 402us/step - loss: 9
7848.7349
Epoch 26/100
57121/57121 [==============================] - 23s 407us/step - loss: 9
6444.4580
Epoch 27/100
57121/57121 [==============================] - 23s 405us/step - loss: 9
5715.2585
Epoch 28/100
57121/57121 [==============================] - 26s 447us/step - loss: 9
5283.9409
Epoch 29/100
57121/57121 [==============================] - 26s 461us/step - loss: 9
5029.7510
Epoch 30/100
57121/57121 [==============================] - 24s 424us/step - loss: 9
5089.1216
Epoch 31/100
57121/57121 [==============================] - 23s 410us/step - loss: 9
5027.7811
Epoch 32/100
57121/57121 [==============================] - 23s 411us/step - loss: 9
4860.1005
Epoch 33/100
57121/57121 [==============================] - 24s 420us/step - loss: 9
4978.8507
Epoch 34/100
57121/57121 [==============================] - 24s 416us/step - loss: 9
4999.4053
Epoch 35/100
57121/57121 [==============================] - 24s 417us/step - loss: 9
5101.9430
Epoch 36/100
57121/57121 [==============================] - 24s 415us/step - loss: 9
4838.0568
Epoch 37/100
57121/57121 [==============================] - 25s 438us/step - loss: 9
5072.5356
Epoch 38/100
57121/57121 [==============================] - 24s 423us/step - loss: 9
5024.4334
```

```
Epoch 39/100
57121/57121 [==============================] - 24s 428us/step - loss: 9
5001.8687
Epoch 40/100
57121/57121 [==============================] - 24s 418us/step - loss: 9
4820.0626
Epoch 41/100
57121/57121 [==============================] - 24s 417us/step - loss: 9
4998.6766
Epoch 42/100
57121/57121 [==============================] - 24s 417us/step - loss: 9
4890.8096
Epoch 43/100
57121/57121 [==============================] - 24s 427us/step - loss: 9
4970.0429
Epoch 44/100
57121/57121 [==============================] - 24s 426us/step - loss: 9
4901.6458
Epoch 45/100
57121/57121 [==============================] - 24s 418us/step - loss: 9
5000.0381
Epoch 46/100
57121/57121 [==============================] - 24s 419us/step - loss: 9
4837.4412
Epoch 47/100
57121/57121 [==============================] - 24s 418us/step - loss: 9
5066.2683
Epoch 48/100
57121/57121 [==============================] - 25s 439us/step - loss: 9
5098.5963
Epoch 49/100
57121/57121 [==============================] - 25s 439us/step - loss: 9
4905.9010
Epoch 50/100
57121/57121 [==============================] - 25s 431us/step - loss: 9
4951.3668
Epoch 51/100
57121/57121 [==============================] - 24s 427us/step - loss: 9
4855.4767
Epoch 52/100
57121/57121 [==============================] - 25s 432us/step - loss: 9
5127.9001
Epoch 53/100
57121/57121 [==============================] - 25s 436us/step - loss: 9
5042.5641
Epoch 54/100
57121/57121 [==============================] - 23s 408us/step - loss: 9
4995.6292
Epoch 55/100
57121/57121 [==============================] - 24s 418us/step - loss: 9
4955.2875
Epoch 56/100
57121/57121 [==============================] - 24s 415us/step - loss: 9
4935.0993
Epoch 57/100
57121/57121 [==============================] - 24s 415us/step - loss: 9
4962.6559
```

```
Epoch 58/100
57121/57121 [==============================] - 25s 432us/step - loss: 9
4922.0258
Epoch 59/100
57121/57121 [==============================] - 24s 418us/step - loss: 9
5117.1851
Epoch 60/100
57121/57121 [==============================] - 24s 428us/step - loss: 9
5149.8232
Epoch 61/100
57121/57121 [==============================] - 24s 421us/step - loss: 9
5107.9799
Epoch 62/100
57121/57121 [==============================] - 24s 428us/step - loss: 9
4872.5520
Epoch 63/100
57121/57121 [==============================] - 25s 437us/step - loss: 9
5088.8702
Epoch 64/100
57121/57121 [==============================] - 25s 437us/step - loss: 9
4861.8157
Epoch 65/100
57121/57121 [==============================] - 24s 424us/step - loss: 9
5072.6171
Epoch 66/100
57121/57121 [==============================] - 24s 426us/step - loss: 9
5201.2834
Epoch 67/100
57121/57121 [==============================] - 24s 425us/step - loss: 9
5023.1407
Epoch 68/100
57121/57121 [==============================] - 26s 446us/step - loss: 9
5214.7189
Epoch 69/100
57121/57121 [==============================] - 25s 434us/step - loss: 9
5012.7454
Epoch 70/100
57121/57121 [==============================] - 25s 436us/step - loss: 9
5106.5353
Epoch 71/100
57121/57121 [==============================] - 26s 448us/step - loss: 9
5135.5900
Epoch 72/100
57121/57121 [==============================] - 25s 432us/step - loss: 9
5279.0495
Epoch 73/100
57121/57121 [==============================] - 26s 448us/step - loss: 9
4917.5997
Epoch 74/100
57121/57121 [==============================] - 25s 438us/step - loss: 9
5232.5315
Epoch 75/100
57121/57121 [==============================] - 25s 431us/step - loss: 9
5015.0652
Epoch 76/100
57121/57121 [==============================] - 25s 437us/step - loss: 9
4854.6005
```

```
Epoch 77/100
57121/57121 [==============================] - 25s 438us/step - loss: 9
4953.0028
Epoch 78/100
57121/57121 [==============================] - 26s 448us/step - loss: 9
5029.5378
Epoch 79/100
57121/57121 [==============================] - 25s 431us/step - loss: 9
4871.1418
Epoch 80/100
57121/57121 [==============================] - 25s 438us/step - loss: 9
5008.2008
Epoch 81/100
57121/57121 [==============================] - 25s 440us/step - loss: 9
5085.0403
Epoch 82/100
57121/57121 [==============================] - 25s 446us/step - loss: 9
4799.9806
Epoch 83/100
57121/57121 [==============================] - 25s 440us/step - loss: 9
5197.3412
Epoch 84/100
57121/57121 [==============================] - 25s 433us/step - loss: 9
4948.6549
Epoch 85/100
57121/57121 [==============================] - 25s 444us/step - loss: 9
4976.5620
Epoch 86/100
57121/57121 [==============================] - 25s 443us/step - loss: 9
4832.2005
Epoch 87/100
57121/57121 [==============================] - 26s 451us/step - loss: 9
5062.1879
Epoch 88/100
57121/57121 [==============================] - 23s 407us/step - loss: 9
5015.9867
Epoch 89/100
57121/57121 [==============================] - 23s 403us/step - loss: 9
4938.0120
Epoch 90/100
57121/57121 [==============================] - 23s 401us/step - loss: 9
4988.5093
Epoch 91/100
57121/57121 [==============================] - 23s 399us/step - loss: 9
4884.1087
Epoch 92/100
57121/57121 [==============================] - 24s 414us/step - loss: 9
4972.0548
Epoch 93/100
57121/57121 [==============================] - 23s 408us/step - loss: 9
4999.3632
Epoch 94/100
57121/57121 [==============================] - 23s 399us/step - loss: 9
5022.4908
Epoch 95/100
57121/57121 [==============================] - 23s 400us/step - loss: 9
5167.5681
```

```
Epoch 96/100
57121/57121 [==============================] - 23s 398us/step - loss: 9
4875.0762
Epoch 97/100
57121/57121 [==============================] - 39s 686us/step - loss: 9
5002.4025
Epoch 98/100
57121/57121 [==============================] - 18s 317us/step - loss: 9
5129.7565
Epoch 99/100
57121/57121 [==============================] - 18s 312us/step - loss: 9
5041.1676
Epoch 100/100
57121/57121 [==============================] - 46s 810us/step - loss: 9
4886.2004
```

```
Out[629]: <keras.callbacks.History at 0x1a0963588>
```

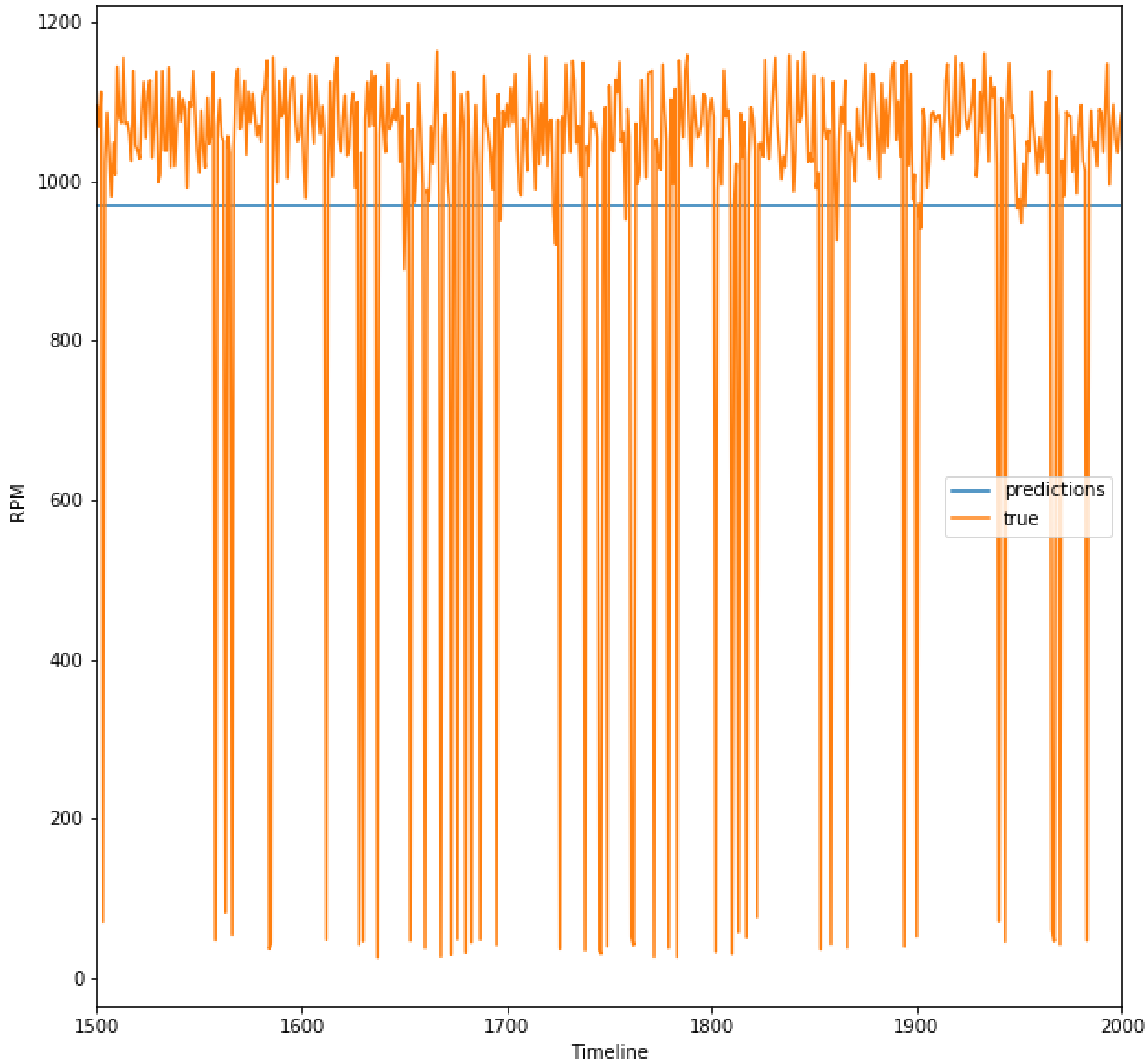
In [552]:
```
model.get_config()
```

Out[552]:


```
{'name': 'sequential_31',
 'layers': [{'class_name': 'LSTM',
   'config': {'name': 'lstm_25',
    'trainable': True,
    'batch_input_shape': (None, 25, 4),
    'dtype': 'float32',
    'return_sequences': False,
    'return_state': False,
    'go_backwards': False,
    'stateful': False,
    'unroll': False,
    'units': 600,
    'activation': 'tanh',
    'recurrent_activation': 'hard_sigmoid',
    'use_bias': True,
    'kernel_initializer': {'class_name': 'VarianceScaling',
     'config': {'scale': 1.0,
      'mode': 'fan_avg',
      'distribution': 'uniform',
      'seed': None}},
    'recurrent_initializer': {'class_name': 'Orthogonal',
     'config': {'gain': 1.0, 'seed': None}},
    'bias_initializer': {'class_name': 'Zeros', 'config': {}},
    'unit_forget_bias': True,
    'kernel_regularizer': None,
    'recurrent_regularizer': None,
    'bias_regularizer': None,
    'activity_regularizer': None,
    'kernel_constraint': None,
    'recurrent_constraint': None,
    'bias_constraint': None,
    'dropout': 0.0,
    'recurrent_dropout': 0.0,
    'implementation': 1}},
  {'class_name': 'Dropout',
   'config': {'name': 'dropout_25',
    'trainable': True,
    'rate': 0.2,
    'noise_shape': None,
    'seed': None}},
  {'class_name': 'Dense',
   'config': {'name': 'dense_40',
    'trainable': True,
    'units': 1,
    'activation': 'relu',
    'use_bias': True,
    'kernel_initializer': {'class_name': 'VarianceScaling',
     'config': {'scale': 1.0,
      'mode': 'fan_avg',
      'distribution': 'uniform',
      'seed': None}},
    'bias_initializer': {'class_name': 'Zeros', 'config': {}},
    'kernel_regularizer': None,
    'bias_regularizer': None,
    'activity_regularizer': None,
    'kernel_constraint': None,
    'bias_constraint': None}}]}
```

In [630]:
```
predictions = model.predict(X_valid)
```

In [632]:
```
plt.figure(figsize=(10,10))
plt.plot(predictions)
plt.plot(y_valid)
plt.ylabel(' RPM')
plt.xlabel('Timeline')
plt.legend(['predictions', 'true'])
plt.xlim((1500, 2000))
plt.show()
```



In [633]:
```
predictions
```

Out[633]:
```
array([[968.5545],
       [968.5545],
       [968.5545],
       ...,
       [968.5545],
       [968.5545],
       [968.5545]], dtype=float32)
```

The predictions i'm obtaining is not fairly accurate and the model needs considerable hyper parameter tuning. Since, my predictions are not accurate from the LSTM model and I'm facing some techincal difficulties generating predictions from ARMA(I need to learn more on the theoretical aspects to solve this), I won't be able to complete the 2nd part of deliverables(predicting which of the sensors would fail).

The idea would be something like below:

1. Once the model has been trained on the train data(data for all 20 sensors present in 'sensors' variable)
2. Preprocess the test data ('test_sensors' var to convert it into appropriate input and dimensions dictated by the LSTM model)
3. Generate predictions for each motor.
4. Put a certain threshold to classify it as a binary classificatin problem.

preds = model.predict(test_sensor[20].noise_free_rms) final_preds = (preds > 0.7)

References

https://stats.stackexchange.com/questions/71802/variable-selection-in-time-series-forecasting (https://stats.stackexchange.com/questions/71802/variable-selection-in-time-series-forecasting)

https://www.researchgate.net/post/How_can_I_make_a_time-series_stationary (https://www.researchgate.net/post/How_can_I_make_a_time-series_stationary)

http://people.duke.edu/~rnau/Slides_on_ARIMA_models--Robert_Nau.pdf (http://people.duke.edu/~rnau/Slides_on_ARIMA_models--Robert_Nau.pdf)

https://www.researchgate.net/post/p_value_of_0000 (https://www.researchgate.net/post/p_value_of_0000)

http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf (http://www.stat.columbia.edu/~madigan/W2025/notes/survival.pdf)

http://ceur-ws.org/Vol-1649/123.pdf (http://ceur-ws.org/Vol-1649/123.pdf)