Data Analytics Case Study

Berk Karahan

Outline

1. Environment and choice of software
2. Data preprocessing
3. Exploration & Feature Selection
4. Modeling
5. Model Blending

Environment and choice of software

- For the case study, Python(3.6.6) is selected from analysis to modeling.
- Libraries used in this case study;
    - Pandas(DataFrame / Series with common data munging tasks)
    - Numpy(Most basic fundemental nd-array package and linear algebra)
    - Matplotlib(2D plotting library)
    - scikit-learn(Machine-learning in Python.)
    - XGBoost(Highly efficient and performant gradient boosting library)
    - mlxtend(Extension and helpers for Python's data analysis capabilities.)
    - Custom classes and functions used in this project.

Custom classes and functions

- readexcel_set_df_names - to read train / test data from *.xlsx sheets while setting index as date/time.
- SimpleAnomalies(class) - Simple time-series anomaly detection based on rolling mean and rolling standard deviation or rolling standard deviation.
- resample - Custom two step resampling function for IoT series mentioned in this post.
- time_series_train_test_split and single_ts_split - Train / test splitting for timeseries.
- plotModelResults and plotCoefficients - plotting results and anomalies for time-series exploratory models.
- fit_model_cv - A custom function to train n models concurrently using threading, where n equals number of cv-splits. Scikit-learn's special cv generator is used.
- iqr_filter_outliers - Simple anomaly filtering based on interquartile range.

Custom classes and functions

- There are also other custom classes in the helpers file. However, they will not be mentioned since they are discarded during trials.

Data preprocessing

- To begin with, my first aim was to make frequency of training set strictly 8hrs. Followed by checking any NaN values(there may not be any observations within that resampled 8 hr bucket).
  - The built-in resample method for pandas resamples data given rule(mean/sum/median).

```
#resample and check for missing values
tr = tr.resample(rule='8H', base=00).mean()
tr.isna().sum()
```

- NaN's are interpolated inplace using time method.

```
#interpolate missing values.
tr.interpolate(method='time', inplace=True)
```

Data preprocessing

- The custom resample function for two step resampling(first upsampling to seconds using interpolation and mean and then upsampling to desired rate of 15 minutes using forward fill.)

```
from helpers.funcs import resample
```

```
tr_res = pd.DataFrame()
```

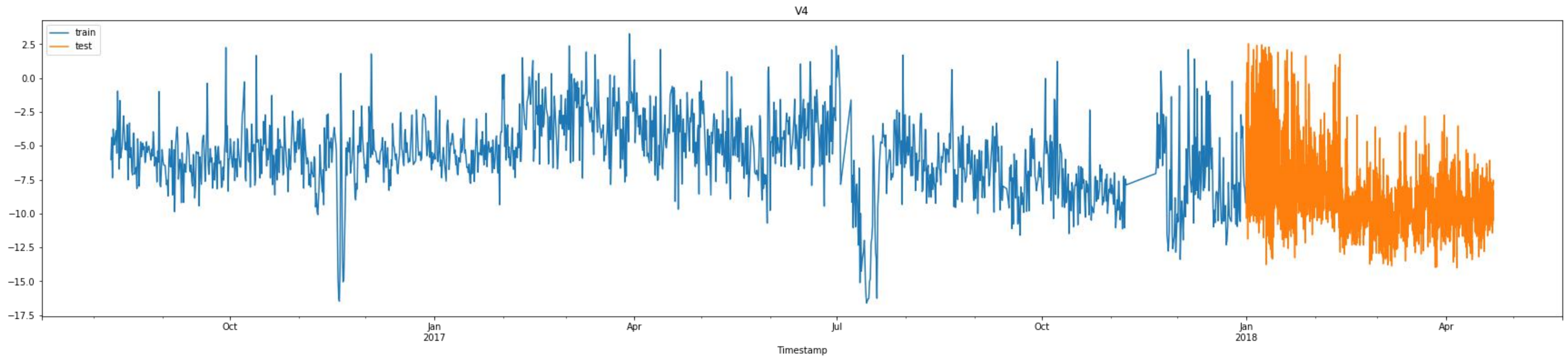
```
for v in tr.columns.values:
    tr_res[v] = resample(tr[v])
```

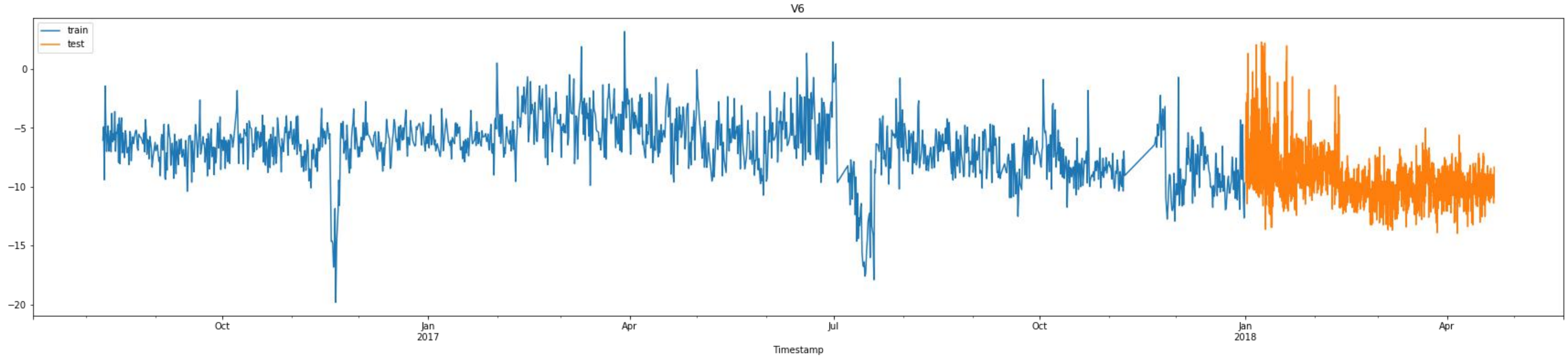
Data preprocessing

- The new resampled train set showed similar distributions to test set.

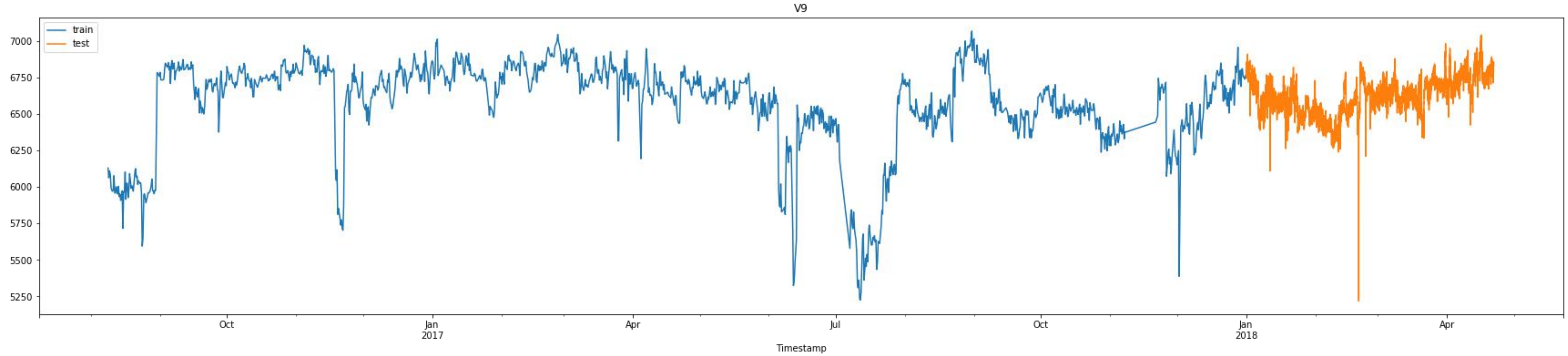
- Next slides show train-test set series plots for some of the variables(sensors).

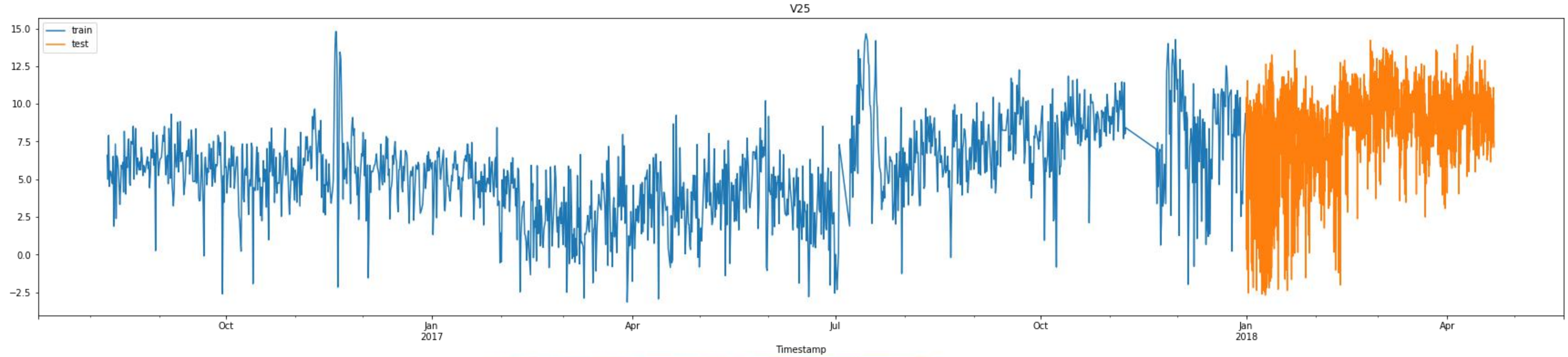
Data preprocessing





Data preprocessing

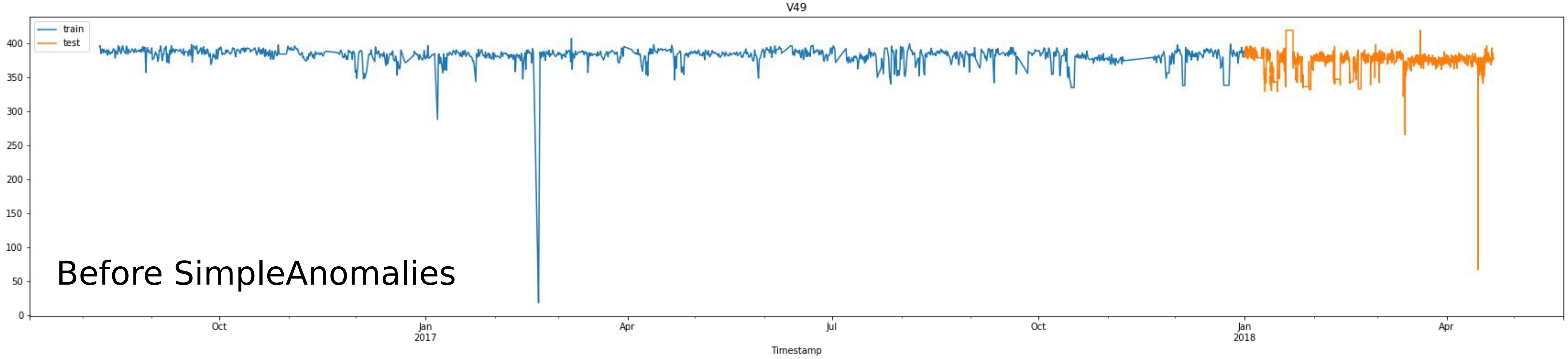


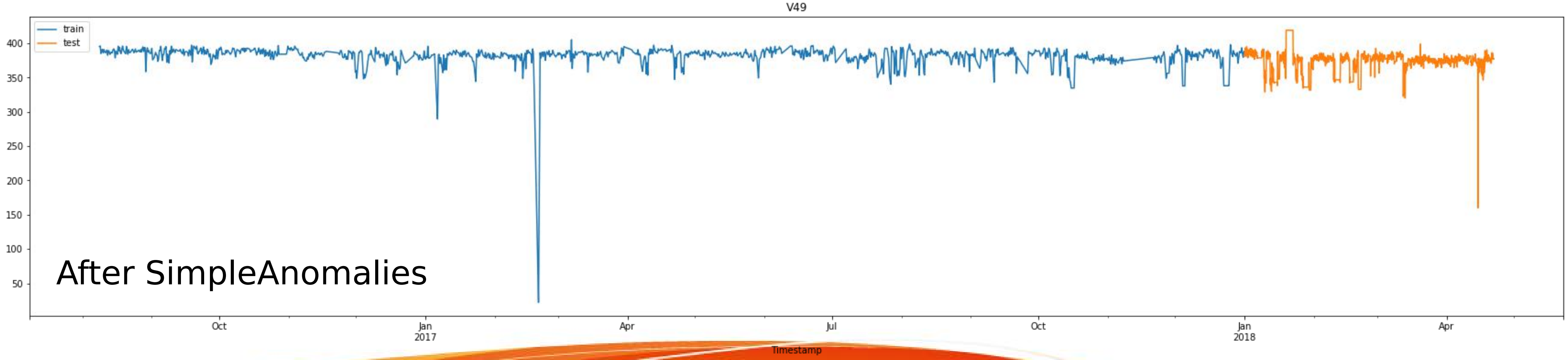


Data preprocessing

- For the rest of the plots please refer to the Jupyter notebook:
  - resample_rollingmean_tr_ts
- The anomalies in theses series are first treated with the SimpleAnomalies class. However, it didn't succeed on catching some of the obvious anomalies.

Data preprocessing

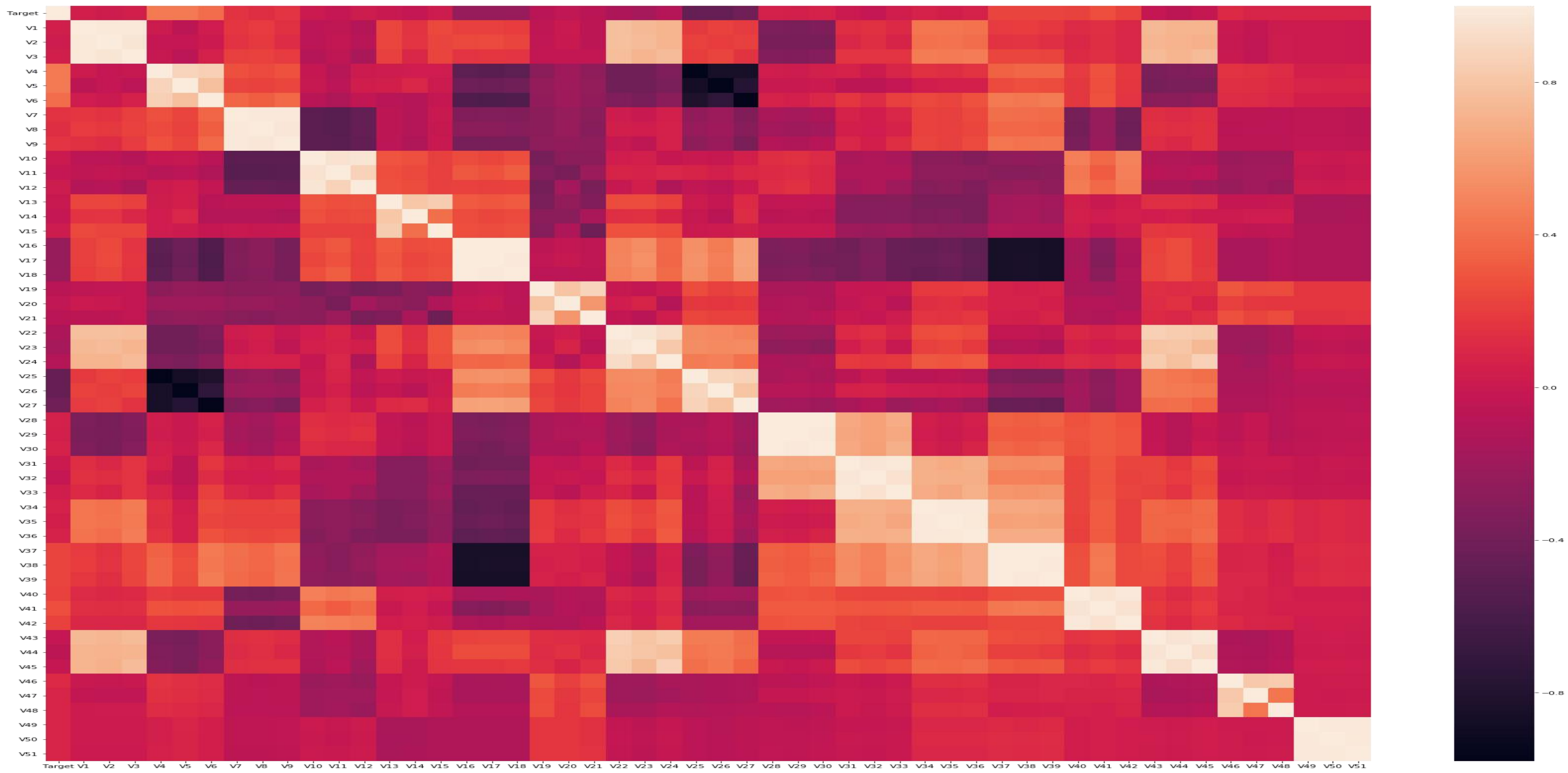




Exploration & Feature Selection

- I have started this stage by first visualising the correlation plot. However, due to the dimensionality of this dataset, it was not easy to gather valuable insights from the plot.

Exploration & Feature Selection - Correlation Plot



Exploration & Feature Selection

- Thus, I went on with fitting a linear model. The choice of linear model was Lasso. Lasso model is basically a linear model with L1 regularization. L1 regularization enables model weights to reach zero, hence pointing out the obvious ones that do not affect the target variable in any way.
- Lasso model is also trained on CV using TimeSeriesSplit cv-split generator.
- Next slides show model fitting and visualises the coefficients from the trained lasso model.

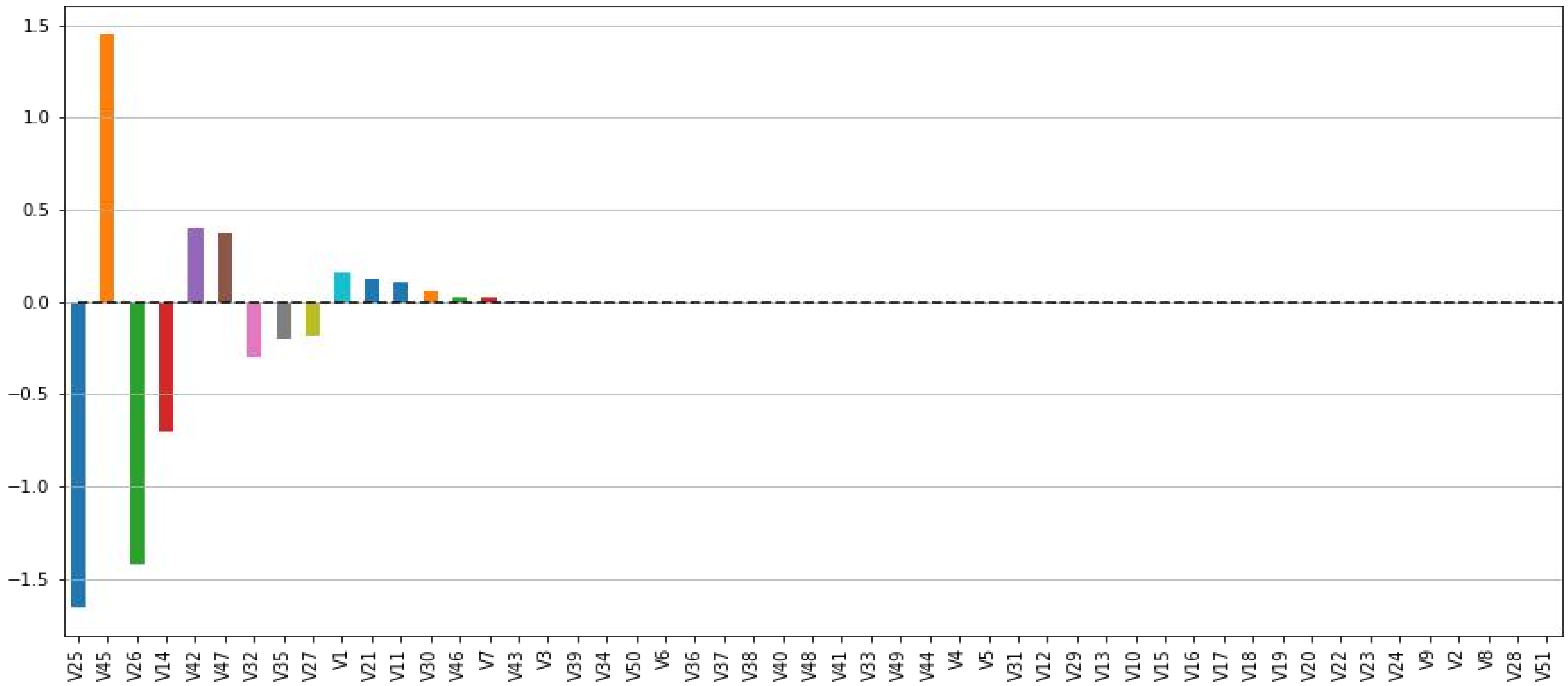
Fitting LassoCV

```
x_tr, x_ts, y_tr, y_ts = TSSplit(tr.drop('Target',1),tr.Target, test_size=0.3)
scaler = StandardScaler()
x_tr_sc = scaler.fit_transform(x_tr)
x_ts_sc = scaler.fit_transform(x_ts)
tscv = TimeSeriesSplit(n_splits=5)
lasso = LassoCV(cv=tscv, random_state=NB_Seed, max_iter=2000)
```

```
lasso.fit(x_tr_sc, y_tr)
```

Lasso model is trained on the scaled time-series cross validation folds of the train split.

Feature Importances from LassoCV



Exploration & Feature Selection

- After dropping unnecessary variables(with feature importances = 0). I have made another attempt for dealing with anomalies using IsolationForest from scikit-learn.
  - IsolationForest is a supervised learning algorithm which groups target variables as outlier(-1) or not(1).
- 5 models are trained using time-series cross validation. I have made a decision to classify a point as an outlier if sum of the predictions are less than or equal to -1.

Exploration & Feature Selection

```
x_tr, x_val, y_tr, y_val = TSSplit(trnz.drop('Target',1),trnz.Target, test_size=0.3)
```

```
isfo = IsolationForest(n_estimators=1000,n_jobs=-1,random_state=NB_Seed)
x_ = x_tr.values
y_ = y_tr.values
```

```
mlist = fit_model_cv(isfo, x_, y_)
```

Fitting IsolationForest on 5 time-series cv.

```
xval = trnz.drop('Target',1)
```

```
outliers = pd.DataFrame()
mdlpreds = list()
for i, m in enumerate(mlist):
    colnm = "model_" + str(i)
    mdlpreds.append(colnm)
    outliers[colnm] = m.predict(xval)
```

```
outliers.index = trnz.index
outliers['sum_score'] = outliers.apply(lambda row:np.sum(row), axis=1)
```

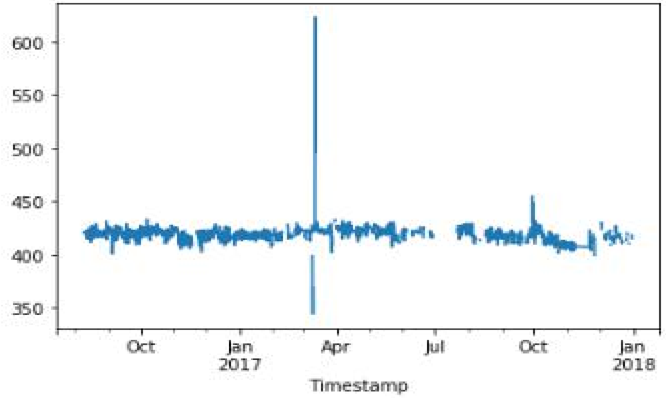
```
outliers['outlier'] = outliers.apply(lambda row: 1 if row.sum_score >= 4 else 0, axis=1)
```

```
trnz = trnz.join(outliers.outlier)
```

```
trnz[trnz.outlier == 1] = np.nan
```

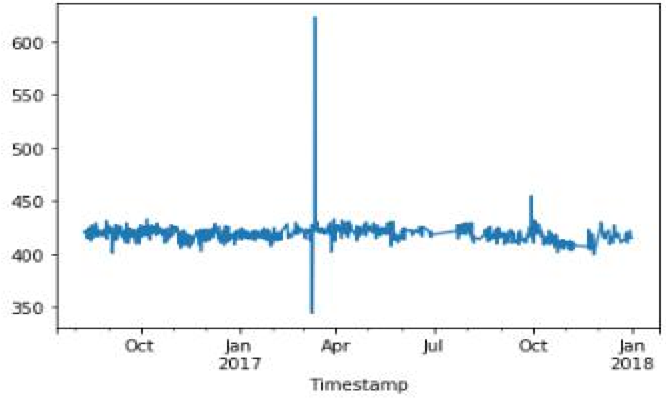
Classifying points as outliers.

Exploration & Feature Selection




```
trnz.Target.interpolate(method='time').plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9be5f3e080>
```



However, the forest method didn't really help to remove anomalies. So I went forward with a simple unsupervised static method. Constructing lower and upper bounds using quartiles.

```
#IQR Outlier drop
trg = trnz.Target
Q1 = trg.quantile(0.25)
Q3 = trg.quantile(0.75)
IQR = Q3 - Q1
LB = Q1 - 3*IQR
UB = Q3 + 3*IQR
bl = (((trg < LB) | (trg > UB)))
```

```
trnz[bl] = np.nan
```

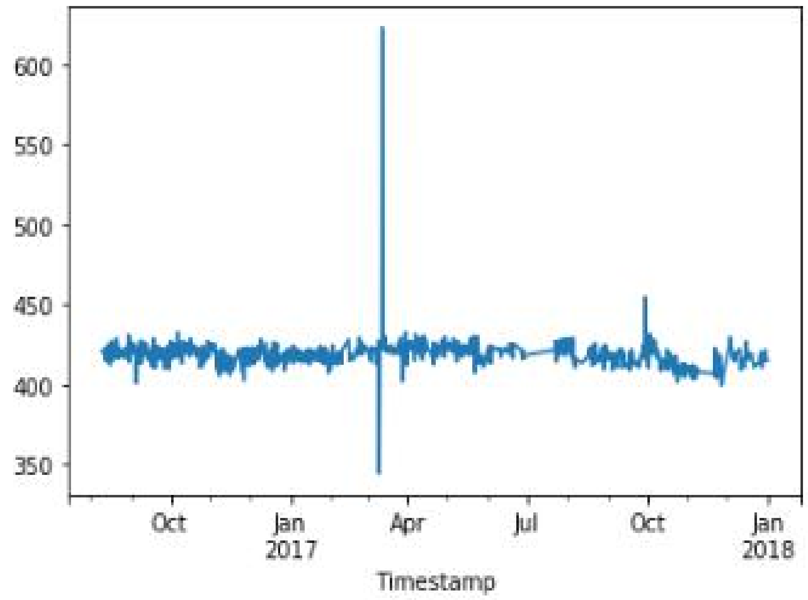
```
trnz.Target.plot()
```

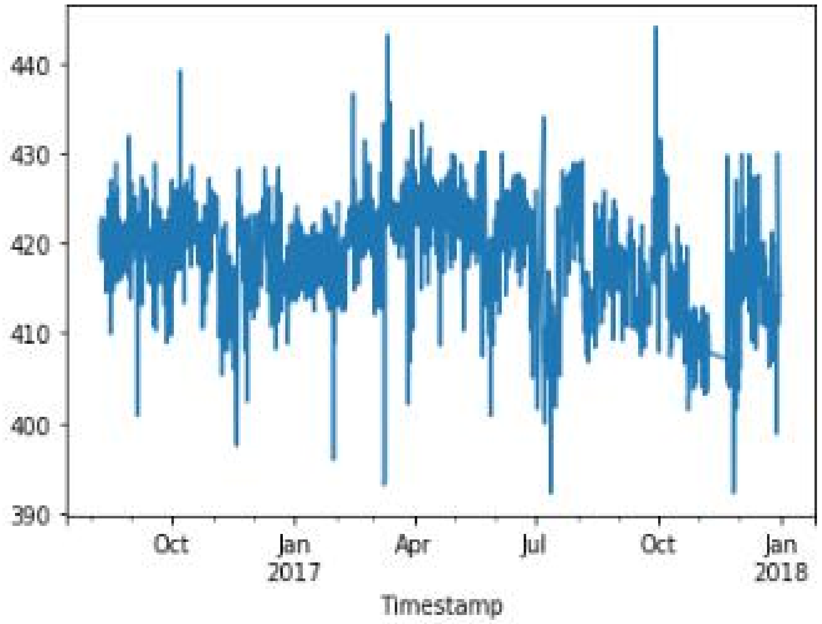
IQR was both simple and effective. Next slide shows target variable series before and after transformation.

Target variable before/after IQR transformation

```
trnz.Target.interpolate(method='time').plot()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f9bfa8aa4e0>
```





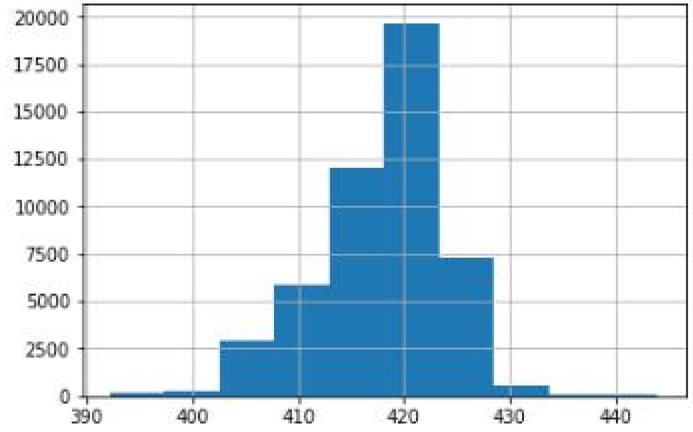
Finally, after this step, data was ready for modeling.

Exploration & Feature Selection

- Due to target variable being approximately normally distributed, No transformation is applied to the target variable.

```
trnz.Target.hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1c2abaa780>
```

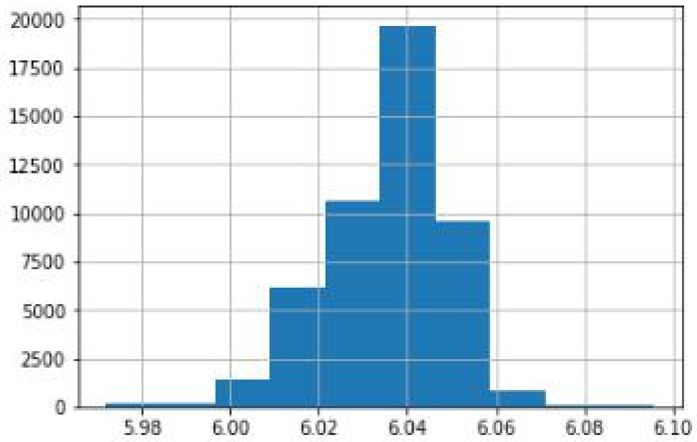


Log transformation had almost no effect on distribution

```
logtarg = np.log(trnz.Target.copy())
```

```
pd.Series(data=logtarg, index=trnz.index).hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1c2ab2d710>
```



Exploration & Feature Selection

- The following plots will examine the timeseries for stationarity with Augmented Dickey-Fuller test. Which is basically hypothesis testing;
  - Null(H0): If failed to be rejected, suggests that time series has a unit root in time making it time dependant.
  - Alternate(H1): If H0 is rejected, it suggests time series doesn't have a unit root in time, making series not time dependant.
- Differencing window will be hourly, per 6 hours, per 12 hours and daily.

Exploration & Feature Selection

```python
def difference_n(timeseries, n):
    timeseries = timeseries - timeseries.shift(n)
    return timeseries.dropna()
```

```python
#hourly freq series
hourly = tr.Target.copy().resample('H', base=00).interpolate(how='time')
#6hrly freq series
hourly_6 = tr.Target.copy().resample('6H', base=00).interpolate(how='time')
#12hrly freq series
hourly_12 = tr.Target.copy().resample('12H', base=00).interpolate(how='time')
#daily freq series
daily = tr.Target.copy().resample('D').interpolate(how='time')
```

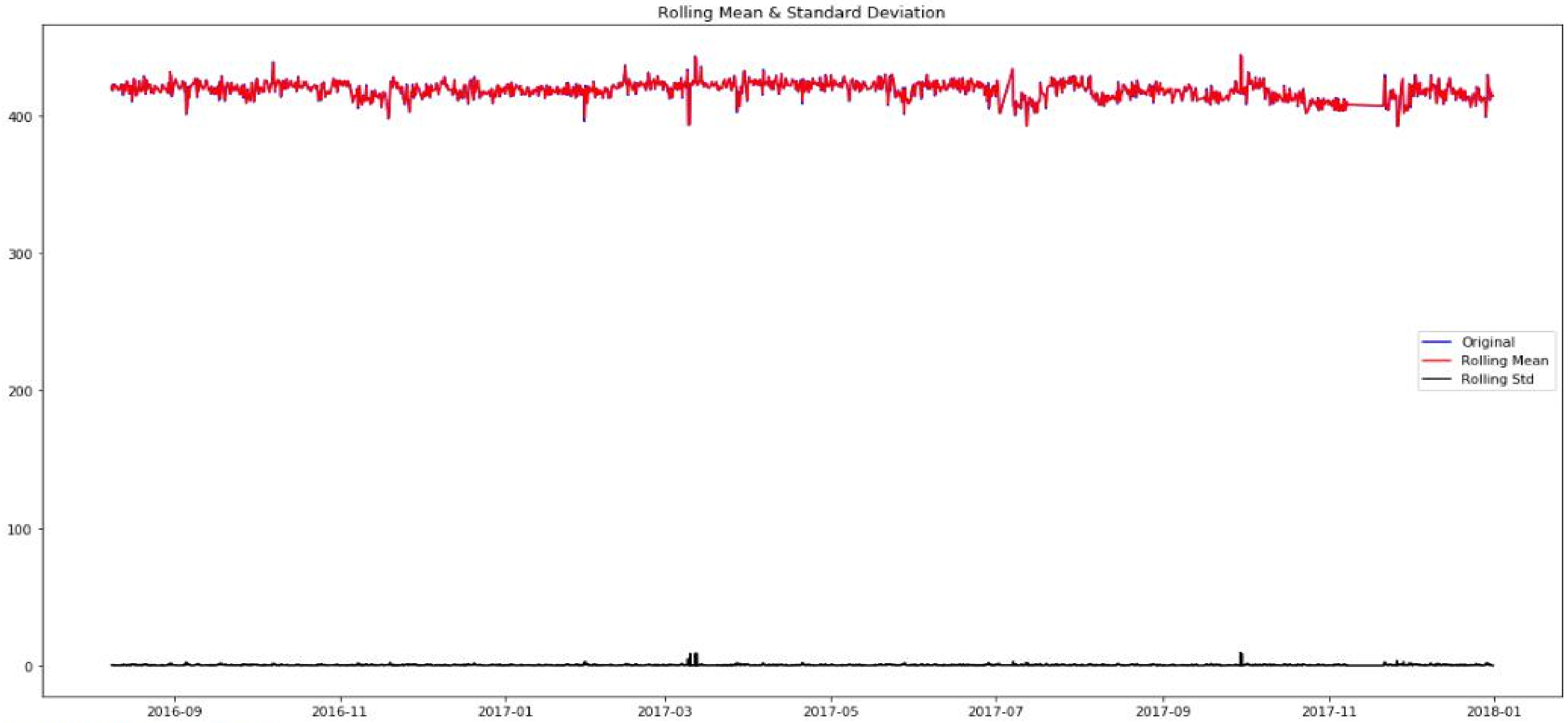
Differencing.

```python
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries, window=12):

    #Determing rolling statistics
    rolmean = timeseries.rolling(window=window).mean()
    rolstd = timeseries.rolling(window=window).std()

    #Plot rolling statistics:
    fig = plt.figure(figsize=(20, 10))
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show()

    #Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
```

Function used for making and plotting augmented dickey-fuller test.

Exploration & Feature Selection

```
#stationary test of initial timeseries
test_stationarity(tr.Target)
```

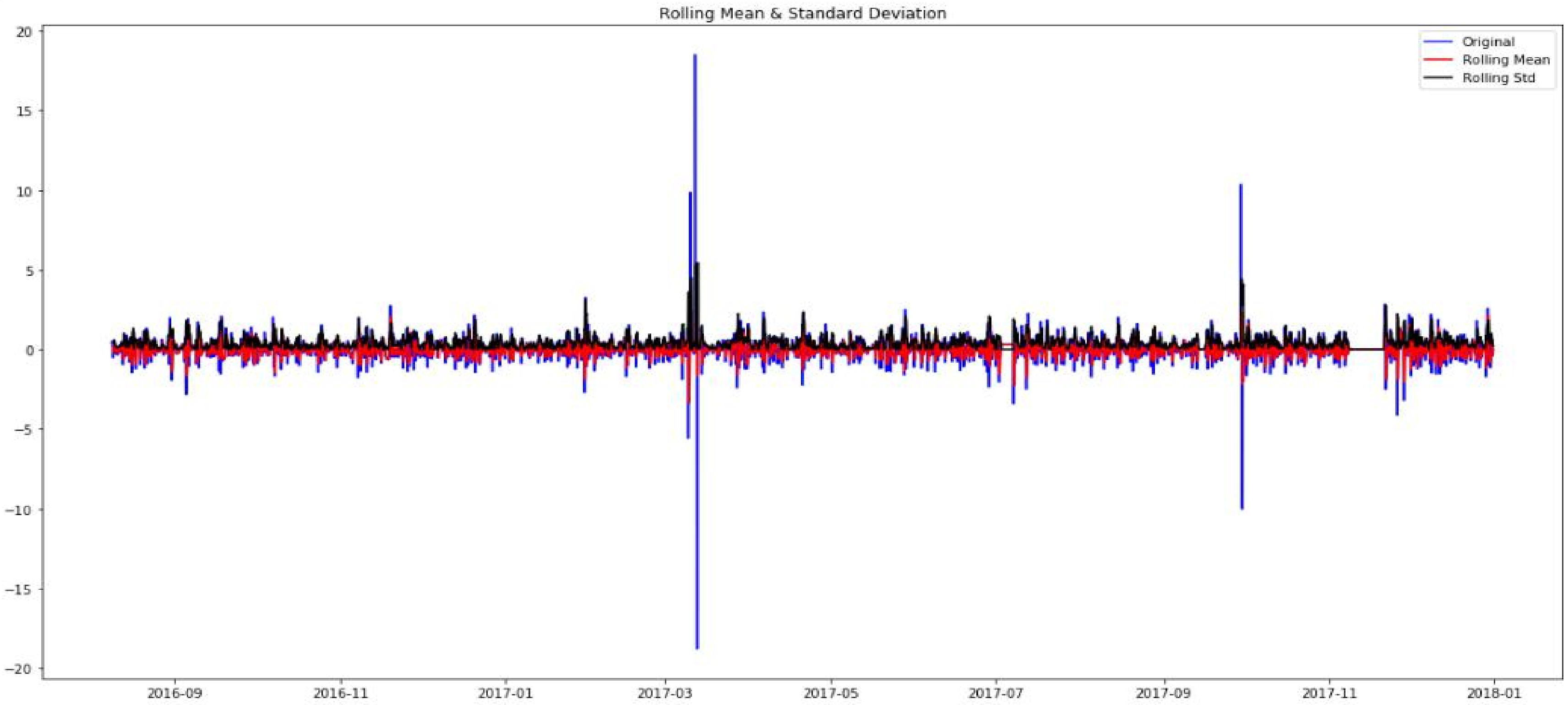



```
Results of Dickey-Fuller Test:
Test Statistic                 -1.154172e+01
p-value                         3.621034e-21
#Lags Used                      5.600000e+01
Number of Observations Used     4.890400e+04
Critical Value (1%)            -3.430484e+00
Critical Value (5%)            -2.861599e+00
Critical Value (10%)           -2.566801e+00
dtype: float64
```

Stationarity of the initial series.

Exploration & Feature Selection

```
#hourly stationarization check, hourly differencing might work. Check others.
fd = difference_n(hourly,1)
test_stationarity(fd)
```

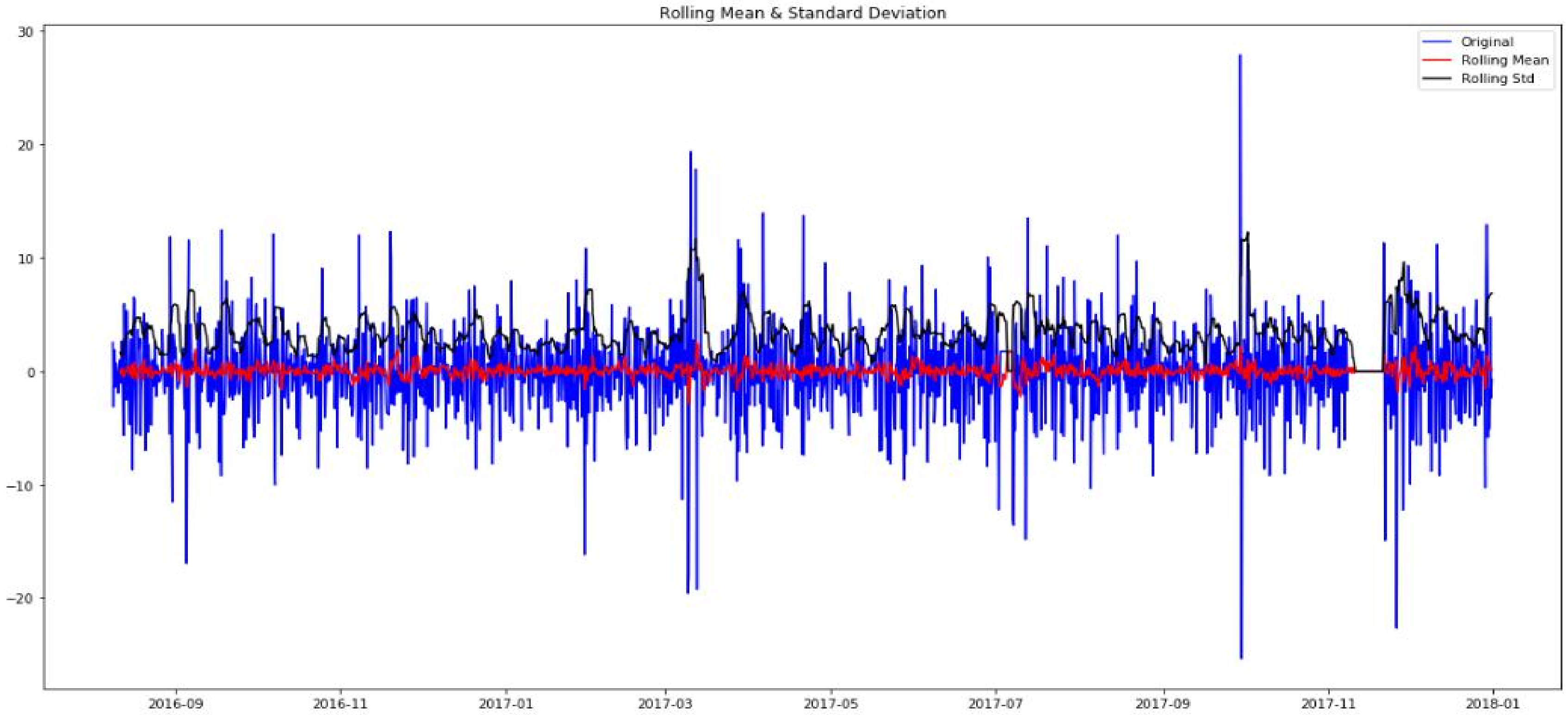



```
Results of Dickey-Fuller Test:
Test Statistic                  -23.204016
p-value                           0.000000
#Lags Used                       40.000000
Number of Observations Used   12199.000000
Critical Value (1%)              -3.430886
Critical Value (5%)              -2.861777
Critical Value (10%)             -2.566896
dtype: float64
```

Stationarity of hourly differenced series.

Exploration & Feature Selection

```
fd = difference_n(hourly_6,1)
test_stationarity(fd)
```

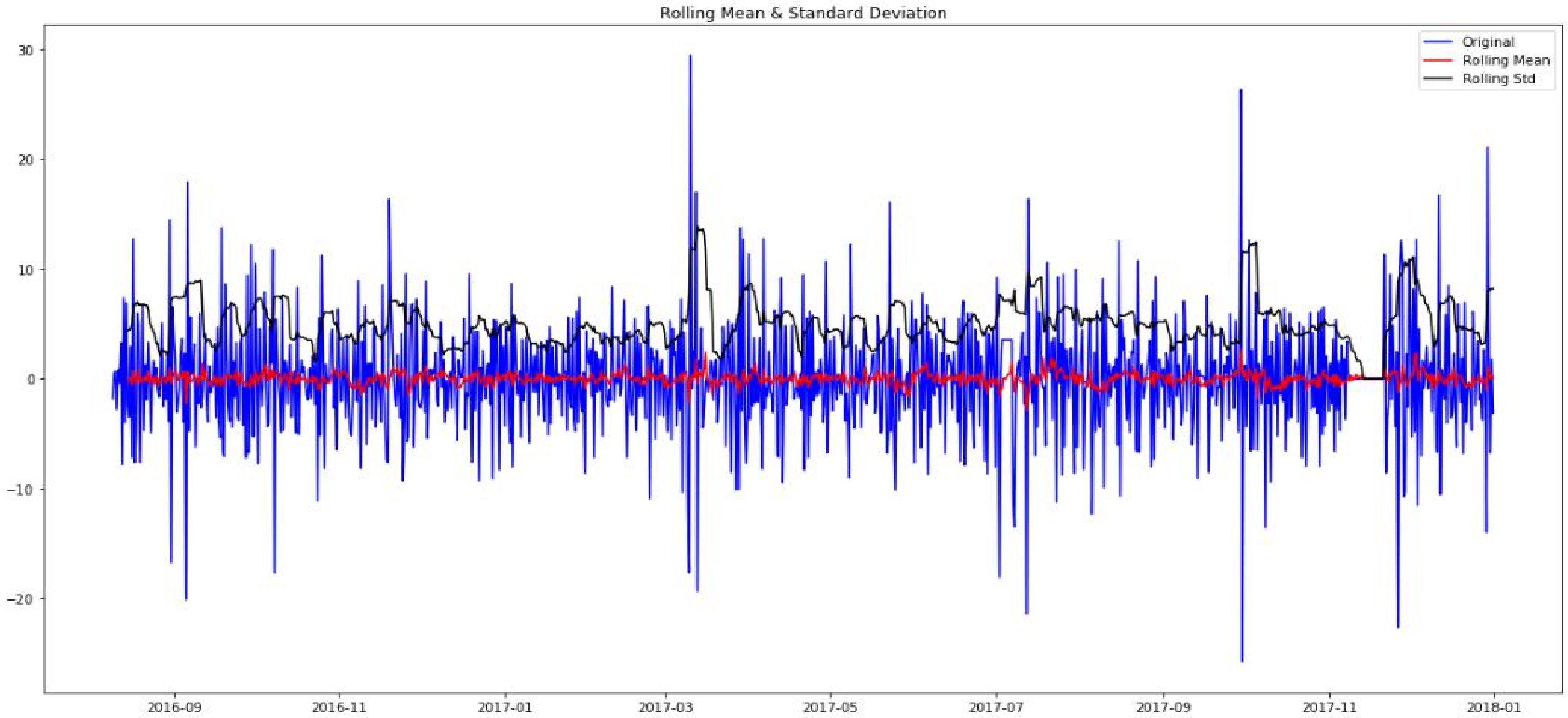



```
Results of Dickey-Fuller Test:
Test Statistic                -1.240638e+01
p-value                        4.442951e-23
#Lags Used                     2.600000e+01
Number of Observations Used    2.012000e+03
Critical Value (1%)           -3.433604e+00
Critical Value (5%)           -2.862978e+00
Critical Value (10%)          -2.567535e+00
dtype: float64
```

Stationarity of 6 hourly differenced series.

Exploration & Feature Selection

```
fd = difference_n(hourly_12,1)
test_stationarity(fd)
```

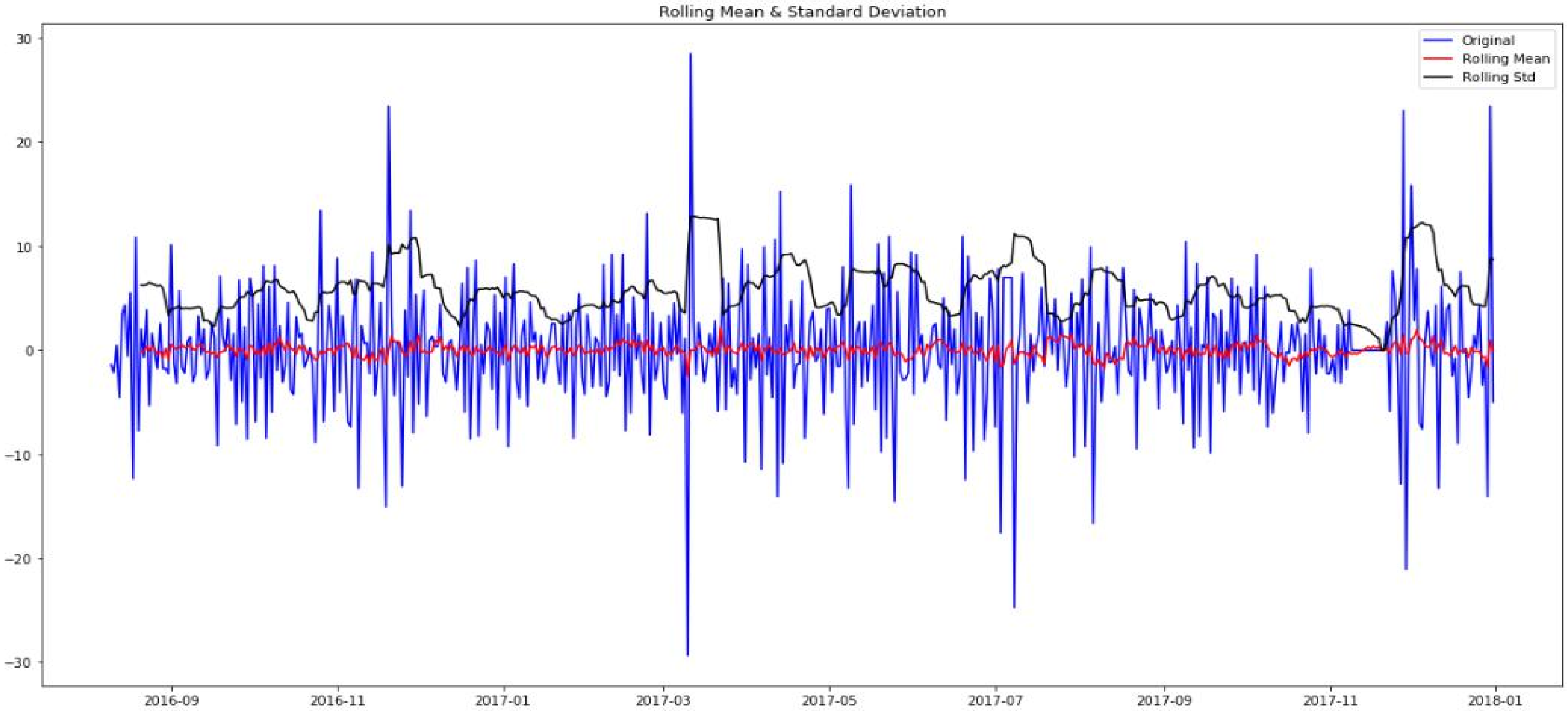



```
Results of Dickey-Fuller Test:
Test Statistic                -1.498697e+01
p-value                        1.140069e-27
#Lags Used                     9.000000e+00
Number of Observations Used    1.009000e+03
Critical Value (1%)           -3.436848e+00
Critical Value (5%)           -2.864409e+00
Critical Value (10%)          -2.568297e+00
dtype: float64
```

Stationarity of 12 hourly differenced series.

Exploration & Feature Selection

```
fd = difference_n(daily,1)
test_stationarity(fd)
```




```
Results of Dickey-Fuller Test:
Test Statistic                 -1.451797e+01
p-value                         5.584400e-27
#Lags Used                      5.000000e+00
Number of Observations Used     5.030000e+02
Critical Value (1%)            -3.443418e+00
Critical Value (5%)            -2.867303e+00
Critical Value (10%)           -2.569840e+00
dtype: float64
```

Stationarity of daily differenced series.

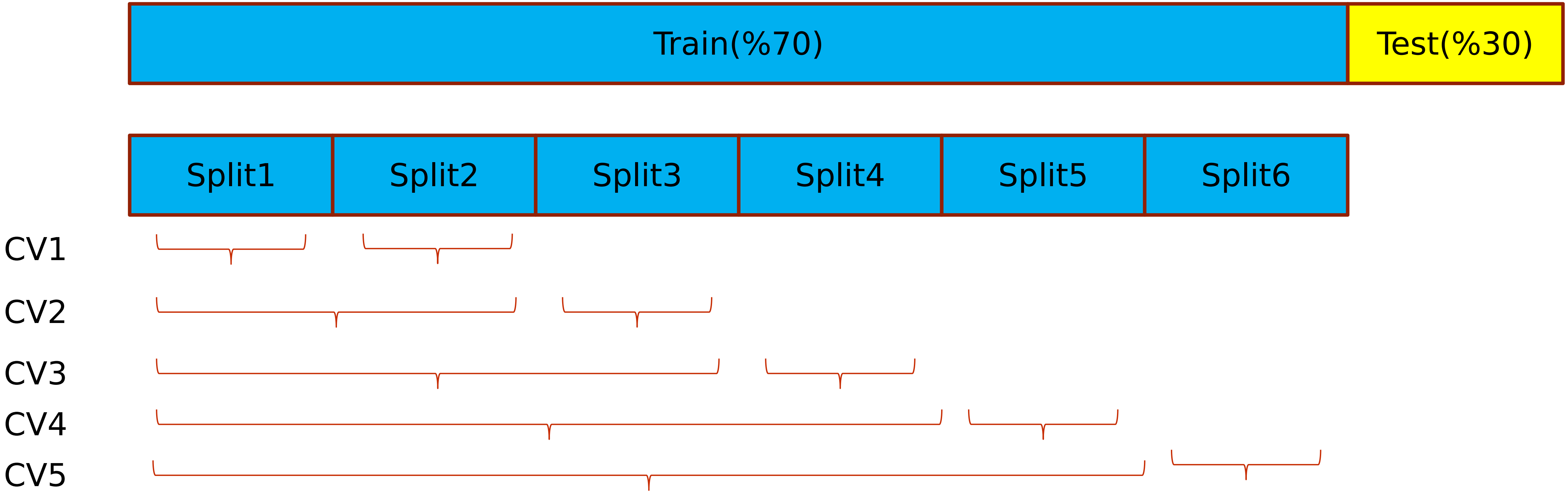
Exploration & Feature Selection

- Among these differentiations clearly hourly differencing was the one which made time series stationary (p-value ~ 0, so we reject H0 and accept H1).
- Although a valuable differencing method is found, I choose to go forward with standard machine learning models rather than time-series methods because the prediction horizon is too long.

Modeling

- Model choices can be separated into different categories. For each different model type, 5 models are trained on each time-series cross validation fold and predictions are aggregated(mean).
  - Linear models
  - Forest models
  - Gradient boosting(XGBoost)
  - Model stacking(with mlxtend)

General model fitting strategy



Modeling - Linear models

- OLS Linear Regression - Most simple linear model with ordinary least squares.
- ElasticNet - Combination of L1 and L2 regularization with ratio.
- HuberRegressor - A regressor that is robust to outliers.
- Ridge - OLS regression with L2 regularization.
- PolynomialRegression - Pipelined regressor with PolynomialFeatures and LinearRegression.
    - Following slides will show metrics of these regression models.

Linear Modeling - Linear Regression

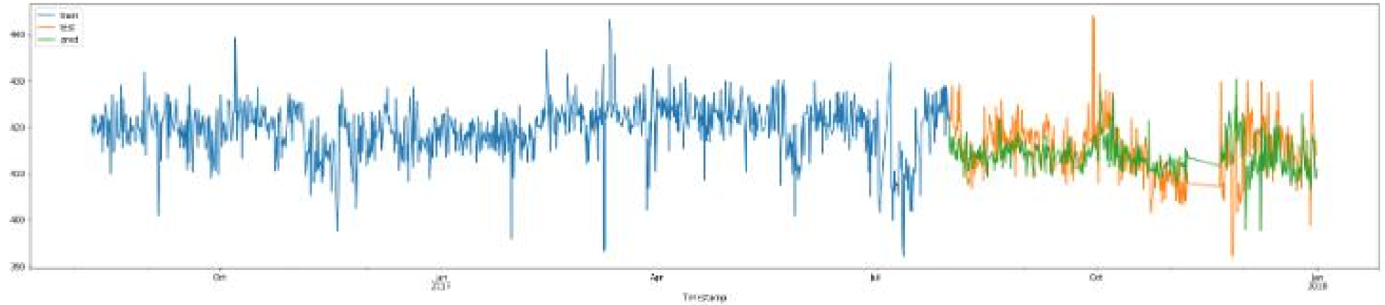
```
reg = LinearRegression(n_jobs=-1);
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 20.762008593407625 validation: 31.84106438732449
RMSE train: 4.556534713288995 validation: 5.6427887065992905
MAE train: 3.3423344888584015 validation: 4.349993182432847
R2 train: 0.23178800040953862 validation: 0.10747463227028142
```



Linear Modeling - ElasticNet

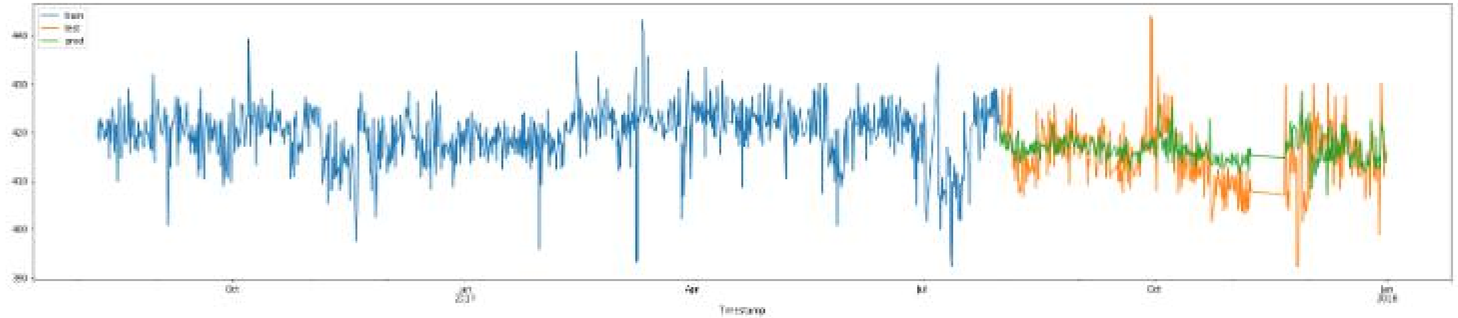
```
reg = ElasticNet(random_state=NB_SEED,l1_ratio=0.2)
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 19.165142077691936 test: 31.884745681355128
RMSE train: 4.377801055060855 test: 5.646657921404052
MAE train: 3.2195322176899395 test: 4.350501863933419
R2 train: 0.2908734214369876 test: 0.10625021770475818
```



Linear Modeling - HuberRegressor

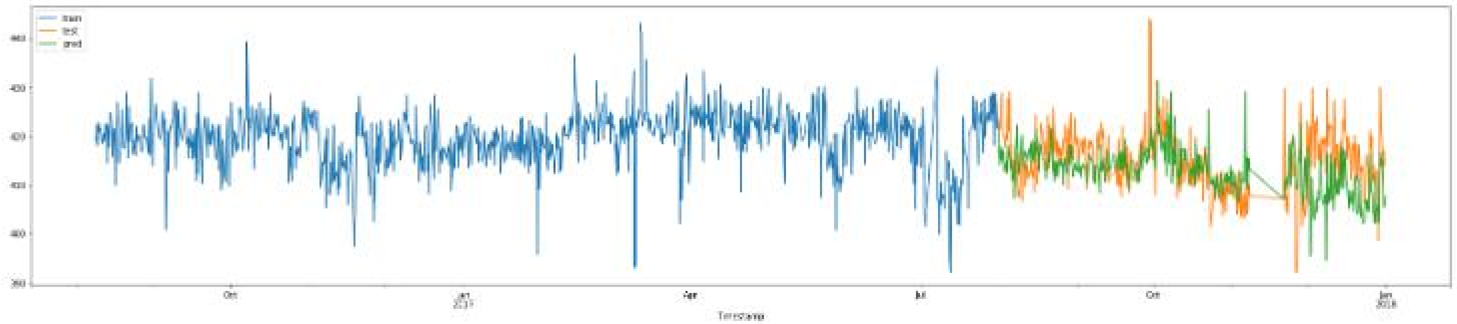
```
reg = HuberRegressor()
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 23.57208907384183 test: 39.1375152535731
RMSE train: 4.855109584122879 test: 6.255998341877426
MAE train: 3.5640090998750167 test: 4.762532161136678
R2 train: 0.12781262947312888 test: -0.09704954485216444
```



Linear Modeling - RidgeRegression

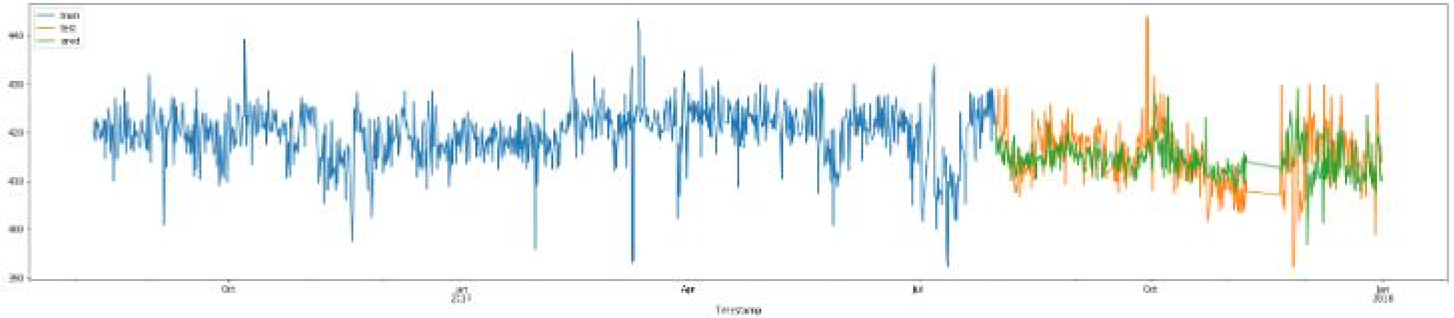
```
reg = Ridge(random_state=NB_SEED)
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 19.661427054335082 test: 30.827917333611634
RMSE train: 4.4341207757948 test: 5.552289377690219
MAE train: 3.247075027034066 test: 4.247547643728879
R2 train: 0.2725104546479846 test: 0.13587379115767484
```



Linear Modeling - Polynomial, degree 2

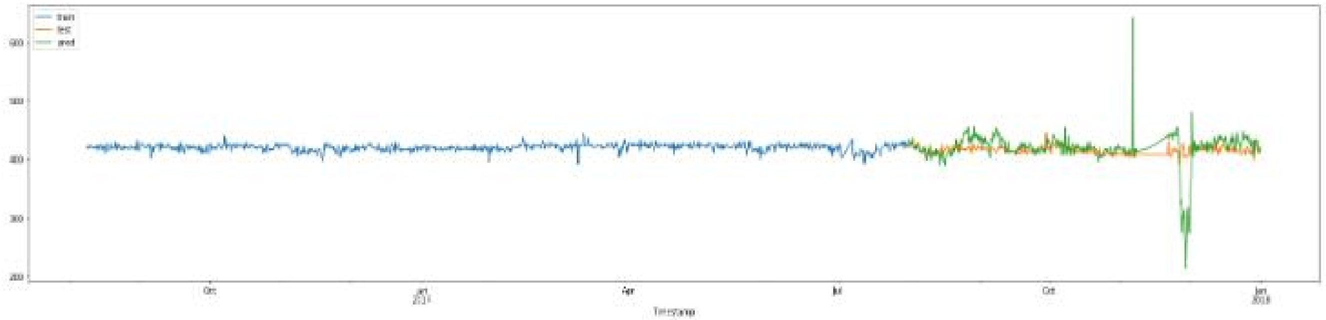
```
reg = Pipeline([('poly', PolynomialFeatures(degree=2)),
                ('linear', LinearRegression(fit_intercept=False))])
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 259.1647586027206 validation: 681.9826888540643
RMSE train: 16.09859492635058 validation: 26.114798273279163
MAE train: 8.209783874621618 validation: 14.23222917875196
R2 train: -8.589316781845067 validation: -18.11641026664568
```



Since polynomial features did significantly worse than other linear models. I didn't fit degree 3 polynomial regression.

Linear Modeling - Support Vector Regression

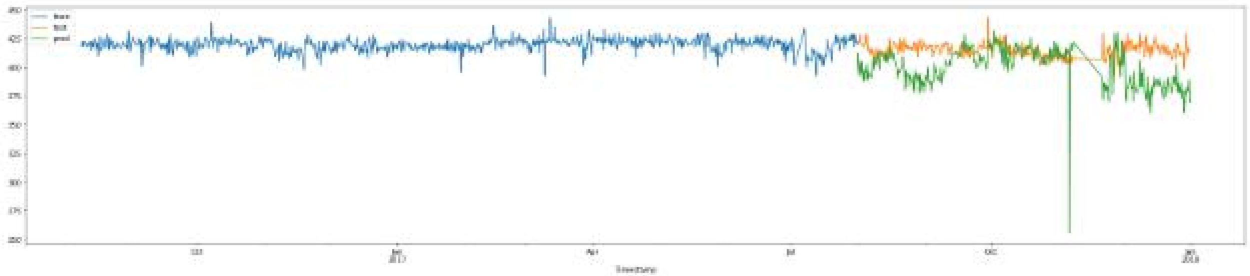
```
reg = SVR(kernel='linear')
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 245.93075575220624 validation: 469.8343834766614
RMSE train: 15.682179560003968 validation: 21.67566339184712
MAE train: 12.351329511911159 validation: 16.6266684150875
R2 train: -8.099647405848007 validation: -12.169757794011002
```



Linear Modeling - Support Vector Regression

```
reg = SVR(kernel='rbf')
```

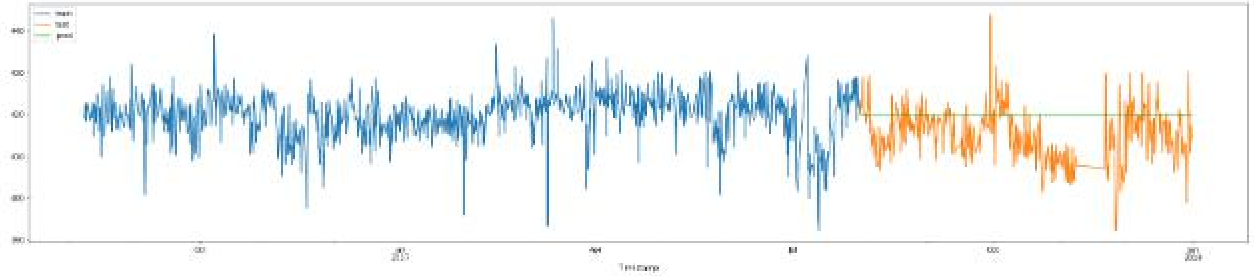
```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 21.691329887034975 validation: 62.6751371743421
RMSE train: 4.65739518261388 validation: 7.9167630490208625
MAE train: 3.0482375437328515 validation: 6.401612811480505
R2 train: 0.19740232110362876 validation: -0.7568241178617405
```

SVR with gaussian kernel. This model might be able to generalize the trend of the timeseries in general.



Modeling - Forest models

- RandomForestRegressor - Ensemble of decision trees.
- AdaBoostRegressor - Collection of weak regressors by weighting both regressors and data points.
- ExtraTreesRegressor - Different than random forest in terms of further randomizing the tree building.
    - Following slides will show metrics of these regression models.

Modeling - RandomForestRegressor

```
reg = RandomForestRegressor(n_estimators=200,random_state=NB_SEED)
```

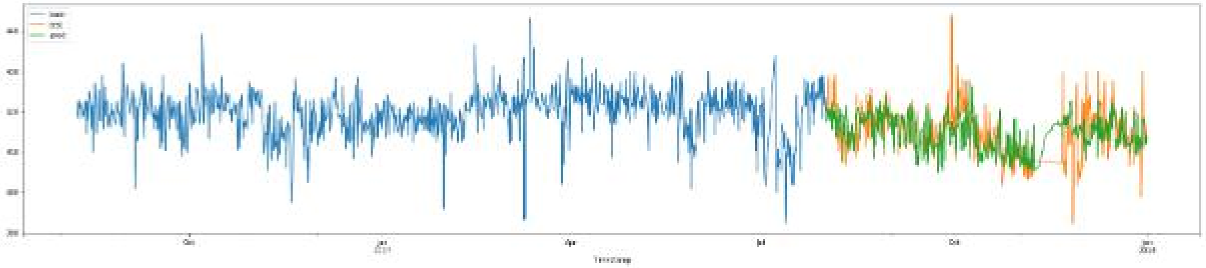
```
mdls=fit_model_cv(reg, x_tr.values,y_tr.values)
```

```
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 10.324655057647398 test: 28.809204236000728
RMSE train: 3.213200127232569 test: 5.36742063155113
MAE train: 2.0892493082202166 test: 3.993245800494347
R2 train: 0.6179789700387898 test: 0.19245960838629816
```



Modeling - ExtraTreesRegressor

```python
reg = ExtraTreesRegressor(random_state=NB_SEED)
```

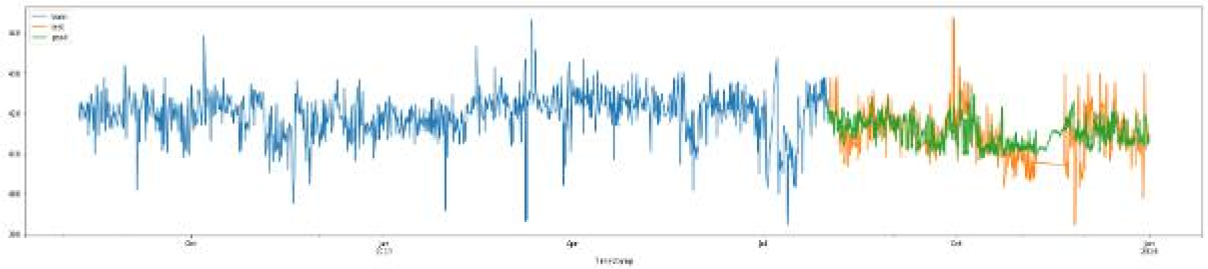
```python
mdls=fit_model_cv(reg, x_tr.values,y_tr.values)
```

```python
#mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```python
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```python
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 9.125084851994004 test: 26.838317984456175
RMSE train: 3.0207755381679724 test: 5.180571202527399
MAE train: 1.9080680421269713 test: 3.9544813541069
R2 train: 0.6623640892428507 test: 0.24770480857858534
```



Modeling - AdaBoostRegressor

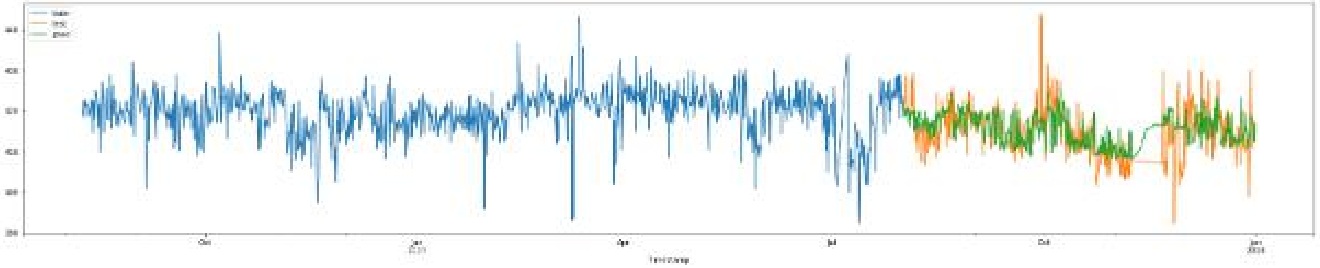
```
reg = AdaBoostRegressor(random_state=NB_SEED)
```

```
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 16.915487318784393 test: 26.332326832688093
RMSE train: 4.112844188488593 test: 5.131503369645984
MAE train: 3.1711599742424608 test: 3.793189517609457
R2 train: 0.37411256339925913 test: 0.2618880636766646
```



Modeling - XGBoost models

- XGBoost is an extremely fast and accurate gradient boosting library. It sequentially builds trees based on optimisation, adding trees which boost the objective in each step.
- XGBoost - Linear
- XGBoost - Tree
  - Following slides will show metrics of these regression models.

Modeling - XGBoost Tree

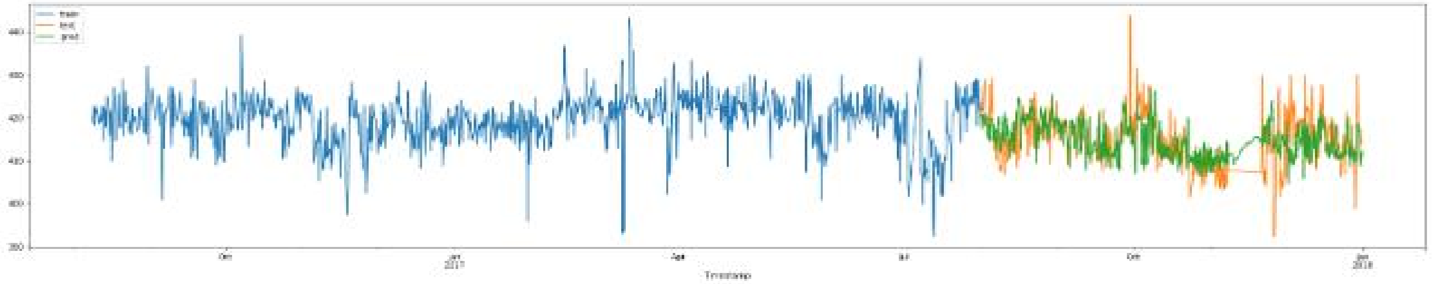
```python
reg = XGBRegressor(random_state=NB_SEED)
p = reg.get_params()
p['n_jobs'] = -1
p['reg_lambda'] = 0.3
```

```python
mdls = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```python
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```python
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 14.541550877559462 test: 27.80543978128069
RMSE train: 3.8133385474619823 test: 5.273086362016148
MAE train: 2.7886077582949684 test: 3.9624533874067103
R2 train: 0.461950233449792 test: 0.22059576703241623
```



Modeling - XGBoost Linear

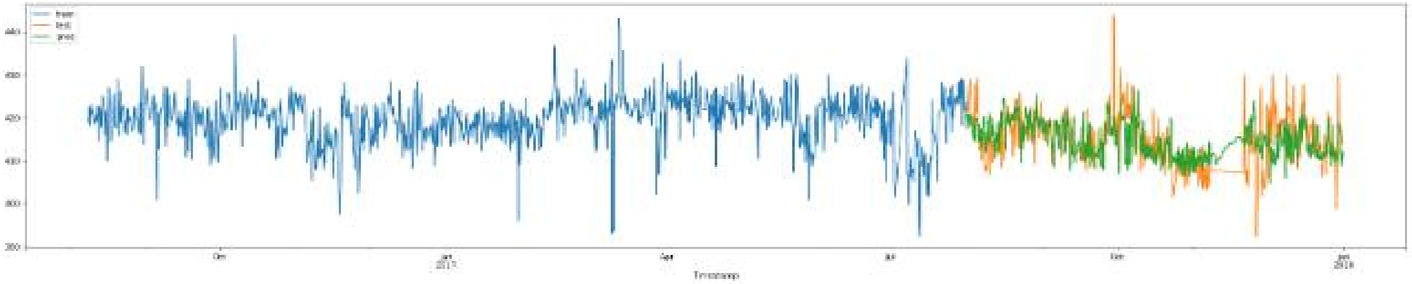
```
reg = XGBRegressor(random_state=NB_SEED, booster='gblinear')
p = reg.get_params()
p['n_jobs'] = -1
p['reg_lambda'] = 0.3
```

```
mdls_lin = fit_model_cv(reg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls_lin)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls_lin)))
for i, m in enumerate(mdls_lin):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 54.968222519221186 test: 99.08411670432872
RMSE train: 7.414055740228906 test: 9.954100496997643
MAE train: 5.821446062320094 test: 7.944689868275418
R2 train: -1.0338710460235712 test: -1.7773910640031927
```



Modeling - Summary

- Before moving on to stacked modeling, the overall performance standing of the models are as follows;

Models	MSE_train	MSE_val	RMSE_train	RMSE_val	MAE_train	MAE_val	R2_train	R_val
Linear Regression	20.761	31.84	4.555	5.643	3.341	4.35	0.232	0.106
Elastic Net	19.164	31.885	4.378	5.645	3.219	4.35	0.291	0.105
Huber Regressor	23.472	39.137	4.854	6.256	3.563	4.762	0.12	-0.097
Ridge Regression	19.66	30.828	4.34	5.551	3.246	4.247	0.271	0.136
Kernel RBF SVR	21.69	62.674	4.656	7.917	3.047	6.402	0.196	-0.756
Extra Trees Regressor	9.125	26.837	3.021	5.18	1.907	3.953	0.661	0.248
AdaBoostRegressor	16.914	26.331	4.113	5.131	3.17	3.792	0.373	0.262
XGB Tree	14.541	27.804	3.812	5.272	2.789	3.9624	0.462	0.22
XGB Linear	54.967	99.083	7.413	9.953	5.82	7.945	-1.03	-1.77

Modeling - Summary R2

- Since R2 can be intuitively interpreted as the percent of prediction eliminated when using a model, it might be a good starting point for choosing base models when building a stacked model.

Models	Average R2
Extra Trees Regressor	0.4545
XGB Tree	0.341
AdaBoostRegressor	0.3175
Ridge Regression	0.2035
Elastic Net	0.198
Linear Regression	0.169
Huber Regressor	0.0115
Kernel RBF SVR	-0.28
XGB Linear	-1.4

Thus we can conclude that we will not be considering Huber Regressor and XGB Linear when building a stacked model. Even though the average R2 score for Kernel RBF SVR is negative I still would like to see its effect so I will continue with the benefit of the doubt.

Modeling - Summary RMSE

- We can also look at the root mean squared error and the percentage of increase from train score to validation score as an indicator of model generalization.

Models	RMSE_train	% increase in RMSE	RMSE_val
AdaBoostRegressor	4.113	0.247507902	5.131
Extra Trees Regressor	3.021	0.714664019	5.18
XGB Tree	3.812	0.383001049	5.272
Ridge Regression	4.34	0.279032258	5.551
Linear Regression	4.555	0.238858397	5.643
Elastic Net	4.378	0.289401553	5.645
Kernel RBF SVR	4.656	0.700386598	7.917

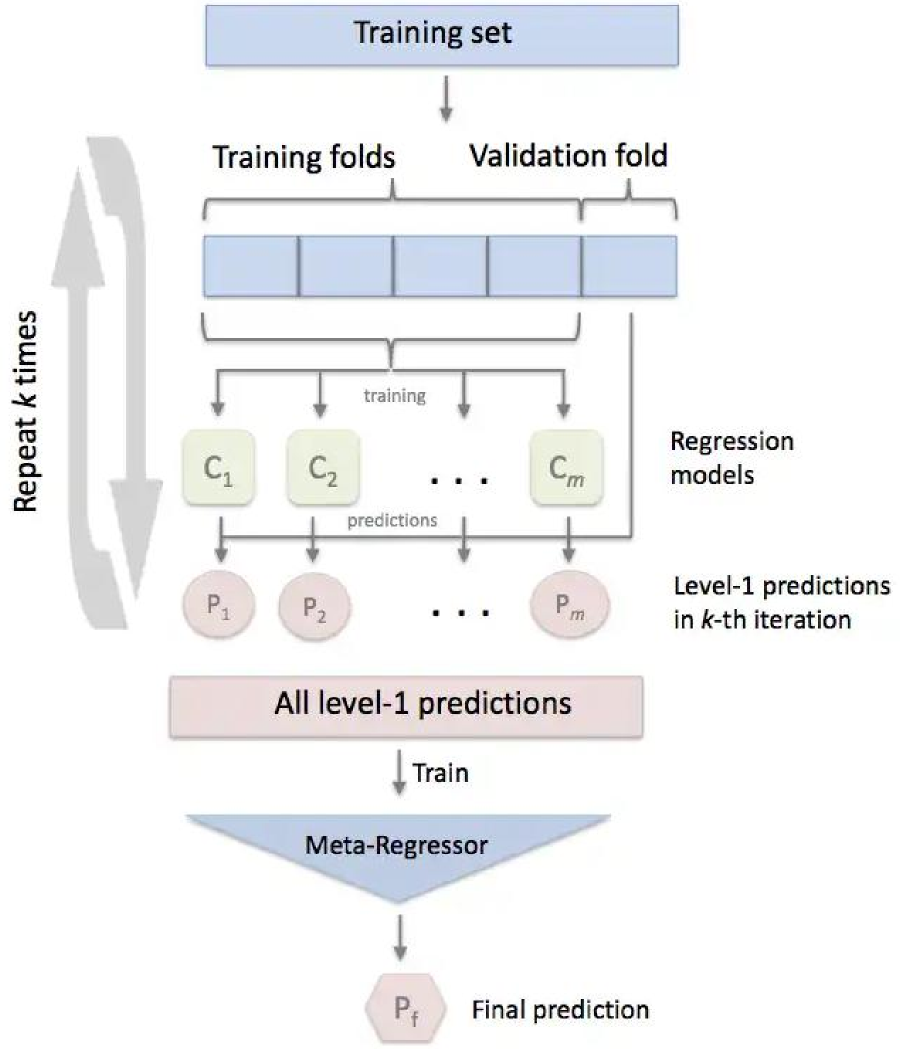
If we sort the performances of models first by lowest RMSE_val and then first by % inc in RMSE, we can get the model goodness order according to our criteria.

Model Blending

- Model blending is a technique that, in a nutshell, enables engineers to combine the best of both worlds.
- The point is to fit models on the original dataset, build predictions, refit a meta-model to the predictions and finally build the resulting prediction.

Model Blending

- For this task, I will be using mlxtend. Mlxtend is a helper package for daily / production ML related tasks.
- To be consistent with the scoring strategy I also choose to fit the stacked model on time-series split cross validation.



Model Blending - Different configurations

- The choice of meta-regressor will be Lasso, for L1 regularization allows the model weights to be zero, negating the effect of unnecessary predictors.
- Model also allows the meta-regressor to be trained with original data along with the predictions of regressors.
- Hence the configurations will be;
  - M1 R: [AdaBoost, ExtreTrees, XGB Tree, Ridge, Linear, ElasticNet, Kernel RBF] M: Lasso, use org data
  - M2 R: [AdaBoost, ExtreTrees, XGB Tree, Ridge, Linear, ElasticNet] M: Lasso, use org data
  - M3 R: [AdaBoost, ExtreTrees, XGB Tree, Ridge, Linear, ElasticNet, Kernel RBF] M: Lasso
  - M4 R: [AdaBoost, ExtreTrees, XGB Tree, Ridge, Linear, ElasticNet] M: Lasso

Model Blending

- The base stacked model will be all regressors with Lasso as meta-regressor and using original data in mid layer.
- Candidate models;

```
rf = RandomForestRegressor(n_estimators=200,n_jobs=-1,random_state=NB_SEED)
ada = AdaBoostRegressor(random_state=NB_SEED)
ext = ExtraTreesRegressor(random_state=NB_SEED)
lr = LinearRegression(n_jobs=-1)
enet = ElasticNet(random_state=NB_SEED,l1_ratio=0.2)
hbr = HuberRegressor()
rdg = Ridge(random_state=NB_SEED)
svr_g = SVR(kernel='rbf')
svr_l = SVR(kernel='linear')
xgb_t = XGBRegressor(n_jobs=-1,random_state=NB_SEED,reg_lambda=0.3)
xgb_l = XGBRegressor(booster='gblinear', n_jobs=-1, random_state=NB_SEED, reg_lambda=0.3)
meta_lasso = Lasso(random_state=NB_SEED)
ml_largeiter = Lasso(max_iter=2000, random_state=NB_SEED)
meta_enet = ElasticNet(random_state=NB_SEED, l1_ratio=0.8)
#New meta regressor, with stochastic gradient descent
meta_sgdr = SGDRegressor(penalty='l1', alpha=0.5, max_iter=1000, random_state=NB_SEED, learning_rate='optimal')
```

Model Blending - Base blending model with all regressors

```
stacked = StackingCVRegressor(regressors=(ada,rf,ext,lr,rdg,enet,hbr,xgb_t,xgb_l,svr_g),
                              meta_regressor=meta_lasso,
                              cv=tscv,
                              use_features_in_secondary=True)
```

```
stacked.fit(x_tr.values,y_tr.values)
```

```
/home/berkkarahan/anaconda3/envs/ml/lib/python3.6/site-packages/sklearn/linear_model/coordinate_descent.py:491: Converg
ng data with very small alpha may cause precision problems.
  ConvergenceWarning)
StackingCVRegressor(cv=TimeSeriesSplit(max_train_size=None, n_splits=5),
          meta_regressor=Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=123123,
   selection='cyclic', tol=0.0001, warm_start=False),
          refit=True,
          regressors=(AdaBoostRegressor(base_estimator=None, learning_rate=1.0, loss='linear',
         n_estimators=50, random_state=123123), RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_i..., epsilon=0.1, gamma='auto',
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)),
          shuffle=True, store_train_meta_features=False,
          use_features_in_secondary=True)
```

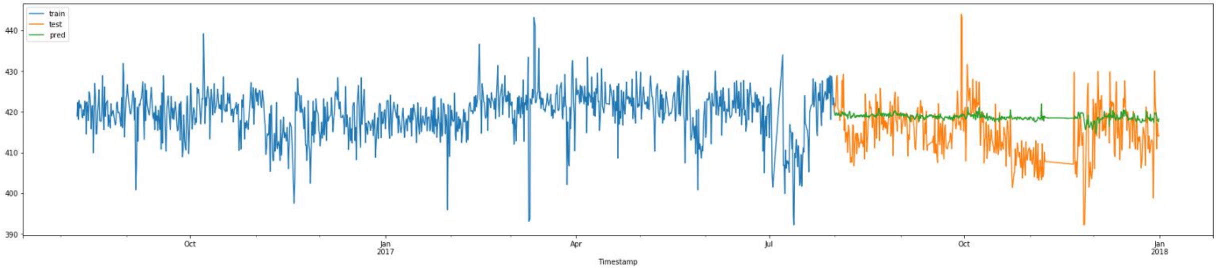
```
y_pred_tr = stacked.predict(x_tr.values)
y_pred_val = stacked.predict(x_ts.values)
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_metric(metrictr, metrics ,mname):
    print(mname +' train: ' + str(metrictr) + ' test: ' + str(metrics))
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 24.1686029076612 test: 50.064898902276845
RMSE train: 4.916157331459318 test: 7.075655369100225
MAE train: 3.5186544734702703 test: 5.654961908228183
R2 train: 0.10574110961029382 test: -0.4033510864948979
```



Model Blending - Base with SGDRegressor as Meta

```
stacked.fit(x_tr.values,y_tr.values)
```

```
StackingCVRegressor(cv=TimeSeriesSplit(max_train_size=None, n_splits=5),
          meta_regressor=SGDRegressor(alpha=0.5, average=False, epsilon=0.1, eta0=0.01,
       fit_intercept=True, l1_ratio=0.15, learning_rate='optimal',
       loss='squared_loss', max_iter=1000, n_iter=None, penalty='l1',
       power_t=0.25, random_state=123123, shuffle=True, tol=None,
       verbose=0, warm_start=False),
          refit=True,
          regressors=(AdaBoostRegressor(base_estimator=None, learning_rate=1.0, loss='linear',
         n_estimators=50, random_state=123123), RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_i..., epsilon=0.1, gamma='auto',
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)),
          shuffle=True, store_train_meta_features=False,
          use_features_in_secondary=True)
```

```
y_pred_tr = stacked.predict(x_tr.values)
y_pred_val = stacked.predict(x_ts.values)
```

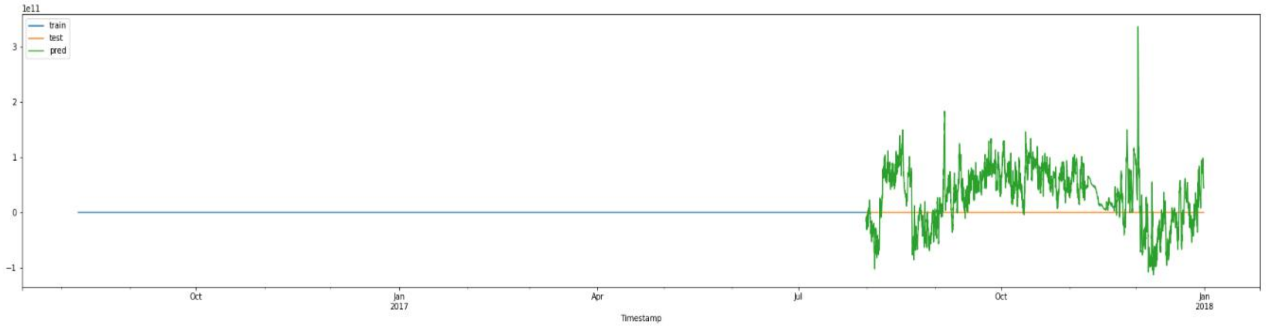
```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_metric(metrictr, metrics ,mname):
    print(mname +' train: ' + str(metrictr) + ' test: ' + str(metrics))
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 3.709654771760358e+21 test: 3.581663911590494e+21
RMSE train: 60906935333.83828 test: 59847004198.96133
MAE train: 48170276495.76109 test: 50288051082.84151
R2 train: -1.3726038582361407e+20 test: -1.0039632660801197e+20
```

Due to SGDRegressor being significantly worse than Lasso as a meta regressor even in base model, I did't put other blending trials in the presentation.



Model Blending - Best blending model on sensor predictors

```
stacked_M6_metalr = StackingCVRegressor(regressors=(ada,ext,xgb_t),
                                        meta_regressor=metalr,
                                        cv=tscv,
                                        use_features_in_secondary=True)
```

```
stacked_M6_metalr.fit(x_tr.values,y_tr.values)
```

```
StackingCVRegressor(cv=TimeSeriesSplit(max_train_size=None, n_splits=5),
          meta_regressor=LinearRegression(copy_X=True, fit_intercept=True, n_jobs=-1, normalize=False),
          refit=True,
          regressors=(AdaBoostRegressor(base_estimator=None, learning_rate=1.0, loss='linear',
         n_estimators=50, random_state=123123), ExtraTreesRegressor(bootstrap=False, criterion='mse', max_depth=None,
          max_features='auto', max_leaf_nodes=None,
          min_impurity_decrease=0.0, min_impu...23123, reg_alpha=0, reg_lambda=0.3,
       scale_pos_weight=1, seed=None, silent=True, subsample=1)),
          shuffle=True, store_train_meta_features=False,
          use_features_in_secondary=True)
```

```
y_pred_tr = stacked_M6_metalr.predict(x_tr.values)
y_pred_val = stacked_M6_metalr.predict(x_ts.values)
```

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_metric(metrictr, metricts ,mname):
    print(mname +' train: ' + str(metrictr) + ' test: ' + str(metricts))
```

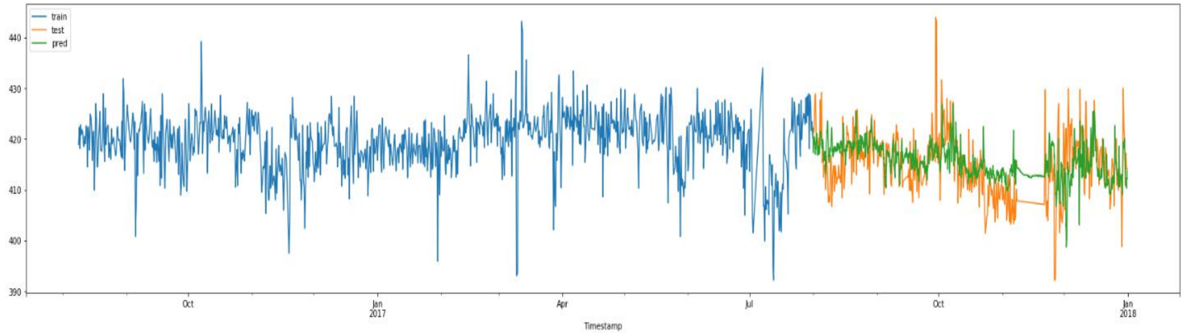
```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 18.43060236901514 test: 27.784144589178755
RMSE train: 4.293087742990486 test: 5.27106674110457
MAE train: 3.0494182538293395 test: 4.0865958673708125
R2 train: 0.31805201621708834 test: 0.22119268486564103
```

Best single stacked model consisted of;
-AdaBoostRegressor
-ExtraTreesRegressor
-XGBoostRegressor
with Linear Regression with original features as meta regressor.

Results are not surprising because all of these models are ensemble models aiming to provide different but imporoved signal generalization capabilities.



Model Blending - Revisiting feature engineering

- While there are other combinations for Lasso and ElasticNet being meta-regressors there are no significant improvements.
- At this step I went back and added some timefeatures including; Hour, Weekday and Month from the datetime index.
- These are all categorical variables and they need to be encoded. They are going to help model the time dependant nature of the problem.
- The final goal would be to make a linear combination of predictions from the time-features model and the predictions direct from the sensor variables.

Feature engineering revisited

```
######## ADDING TIME RELATED FEATURES TO DATA #########
tr['Date'] = tr.index.values
tr['Weekday'] = tr.Date.apply(lambda x: x.weekday())
tr['Hour'] = tr.Date.apply(lambda x: x.hour)
tr['Month'] = tr.Date.apply(lambda x: x.month)


ts['Date'] = ts.index.values
ts['Weekday'] = ts.Date.apply(lambda x: x.weekday())
ts['Hour'] = ts.Date.apply(lambda x: x.hour)
ts['Month'] = ts.Date.apply(lambda x: x.month)

tr = pd.get_dummies(tr, columns=["Month","Hour","Weekday"])
ts = pd.get_dummies(ts, columns=["Month","Hour","Weekday"])
tr.drop('Date',1,inplace=True)
ts.drop('Date',1,inplace=True)
```

Adding time features to the training and prediction data(ts).

Feature engineering revisited

- There was another problem after one-hot-encoding timefeatures. The shapes of timefeatures train and prediction(test) datasets were not the same. This implies that not all of the categorical date features are present in both sets.
- In order to have a quick fix for this problem I have used PCA. PCA uses linear algebra to project the original data to a lower dimensional space.

```
ts_time.shape
```

```
(10656, 35)
```

```
tr_time.shape
```

```
(48961, 44)
```

```
from sklearn.decomposition import PCA
```

```
pca_tr = PCA(n_components=30)
tr_time_vals = pca_tr.fit_transform(tr_time.drop('Target',1))
pca_ts = PCA(n_components=30)
ts_time_vals = pca_ts.fit_transform(ts_time)
tr_time = pd.DataFrame(data=tr_time_vals,index=tr_time.index)
ts_time = pd.DataFrame(data=ts_time_vals,index=ts_time.index)
```

The resulting datasets are now equal in dimensionality, however, we can not speak the effect of the original sensor variables' direct effect from now on.

Model Blending

- In order to make a comparison I trained the base stacked regressor with lasso meta regressor again with the added features.

Base blending with timefeatures

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 26.452969552064705 test: 70.9461364038728
RMSE train: 5.1432450410285435 test: 8.422952950353741
MAE train: 3.6269173578703784 test: 6.993375315983699
R2 train: 0.02121759832284631 test: -0.9886655079305922
```

Base blending

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 24.1686029076612 test: 50.064898902276845
RMSE train: 4.916157331459318 test: 7.075655369100225
MAE train: 3.5186544734702703 test: 5.654961908228183
R2 train: 0.10574110961029382 test: -0.4033510864948979
```

Since the base models for with features are worse I didn't put all of the model combinations with blending with timefeatures.

Modeling - Revisiting XGB

- With the new features, I wanted to see how the linear XGB would perform on the timefeatures only. Removing the rest of the features, I trained both a linear XGBoost and a tree XGBoost model on the timefeatures. And a simple linear regression for score comparison.

```
xg = XGBRegressor(booster="gblinear",
                   eta=0.001,
                  n_jobs=-1,
                  reg_lambda=0.5,
                  reg_alpha=0.5)

xg_t = XGBRegressor(eta=0.001,
                  min_child_weight=8,
                  subsample=0.9,
                  colsample_bytree=0.8,
                  silent=1,
                  max_depth=8,
                  seed=NB_SEED,
                  n_jobs=-1)
lr = LinearRegression(n_jobs=-1)
```

Modeling - XGB Linear

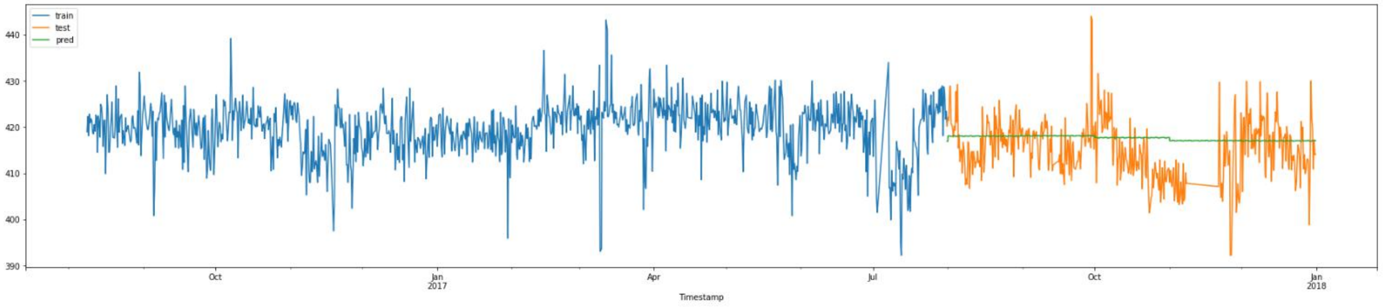
```
mdls = fit_model_cv(xg,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
y_pred = np.zeros((y_ts.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred[:,i]= m.predict(x_ts.values)
y_pred = y_pred.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 32.78714665924634 test: 43.8031897892753
RMSE train: 5.726006170032158 test: 6.618397826458855
MAE train: 4.577395699266501 test: 5.286599076799734
R2 train: -0.21315234904405633 test: -0.22783138147765492
```



Modeling - XGB Tree

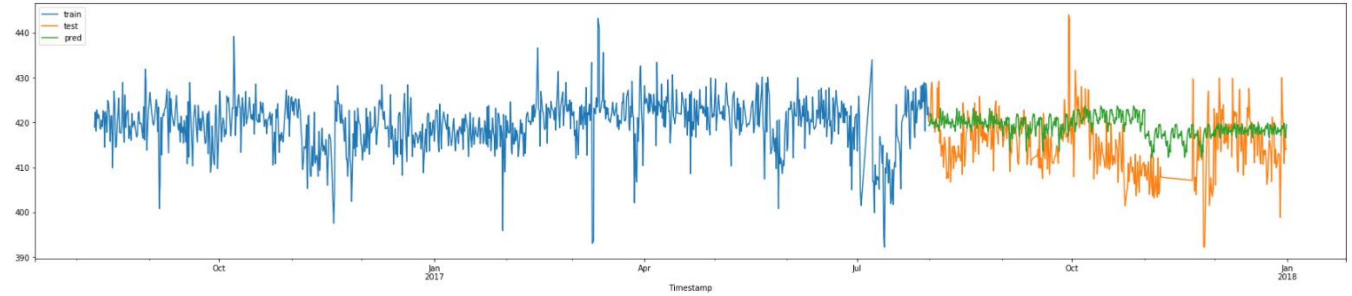
```
mdls = fit_model_cv(xg_t,x_tr.values,y_tr.values)
y_pred_val = np.zeros((y_ts.shape[0],len(mdls)))
y_pred_tr = np.zeros((y_tr.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred_val[:,i]= m.predict(x_ts.values)
    y_pred_tr[:,i] = m.predict(x_tr.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

```
y_pred = np.zeros((y_ts.shape[0],len(mdls)))
for i, m in enumerate(mdls):
    y_pred[:,i]= m.predict(x_ts.values)
y_pred = y_pred.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr.values, y_pred_tr)
r2_tr=r2_score(y_tr, y_pred_tr)
mse_ts = mean_squared_error(y_ts.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts.values, y_pred_val)
r2_ts = r2_score(y_ts.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 22.810663561250255 test: 56.3047992370864
RMSE train: 4.776051042571703 test: 7.503652393140716
MAE train: 3.4269523334329173 test: 6.035558554325833
R2 train: 0.1559860218949437 test: -0.5782594775328478
```



Modeling - Support Vector Regression

```
reg = SVR(kernel='linear')
```

```
svrs = fit_model_cv(reg,x_tr_t.values,y_tr_t.values)
```

```
y_pred_val = np.zeros((y_ts_t.shape[0],len(svrs)))
y_pred_tr = np.zeros((y_tr_t.shape[0],len(svrs)))
for i, m in enumerate(svrs):
    y_pred_val[:,i]= m.predict(x_ts_t.values)
    y_pred_tr[:,i] = m.predict(x_tr_t.values)
y_pred_val = y_pred_val.mean(axis=1)
y_pred_tr = y_pred_tr.mean(axis=1)
```

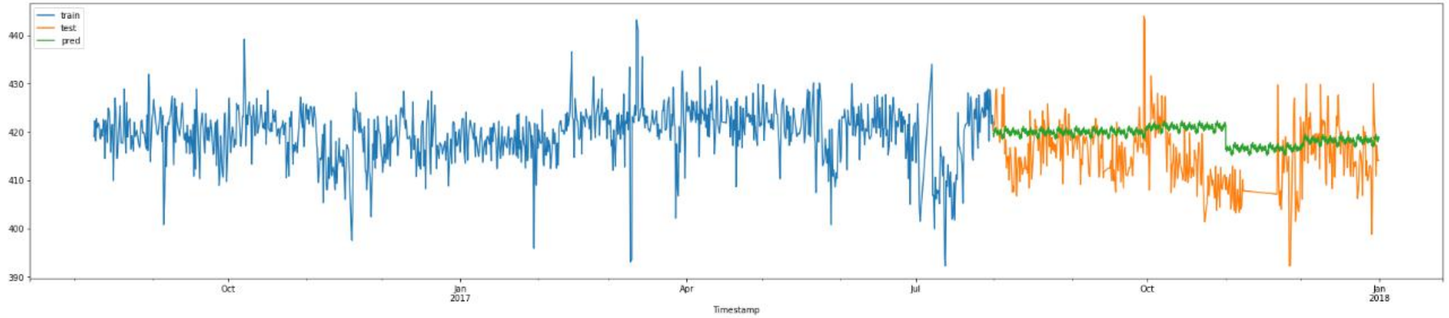
```
y_pred = np.zeros((y_ts_t.shape[0],len(mdls)))
for i, m in enumerate(svrs):
    y_pred[:,i]= m.predict(x_ts_t.values)
y_pred = y_pred.mean(axis=1)
```

```
mse_tr = mean_squared_error(y_tr_t.values, y_pred_tr)
rmse_tr = np.sqrt(mse_tr)
mae_tr = mean_absolute_error(y_tr_t.values, y_pred_tr)
r2_tr=r2_score(y_tr_t, y_pred_tr)
mse_ts = mean_squared_error(y_ts_t.values, y_pred_val)
rmse_ts = np.sqrt(mse_ts)
mae_ts = mean_absolute_error(y_ts_t.values, y_pred_val)
r2_ts = r2_score(y_ts_t.values, y_pred_val)
```

```
print_metric(mse_tr,mse_ts,'MSE')
print_metric(rmse_tr,rmse_ts,'RMSE')
print_metric(mae_tr,mae_ts,'MAE')
print_metric(r2_tr,r2_ts,'R2')
```

```
MSE train: 24.624075815701175 test: 53.66743986048546
RMSE train: 4.962265189981403 test: 7.3258064307273
MAE train: 3.5529632196639245 test: 5.993394929158636
R2 train: 0.08888822411655717 test: -0.5043326100512056
```

The linear SVR was the best performing model for the timefeatures dataset. Although it has negative R2 score, I still want to use it in my final blending because it was able to generalize some of the trend and seasonality.



Final Blending

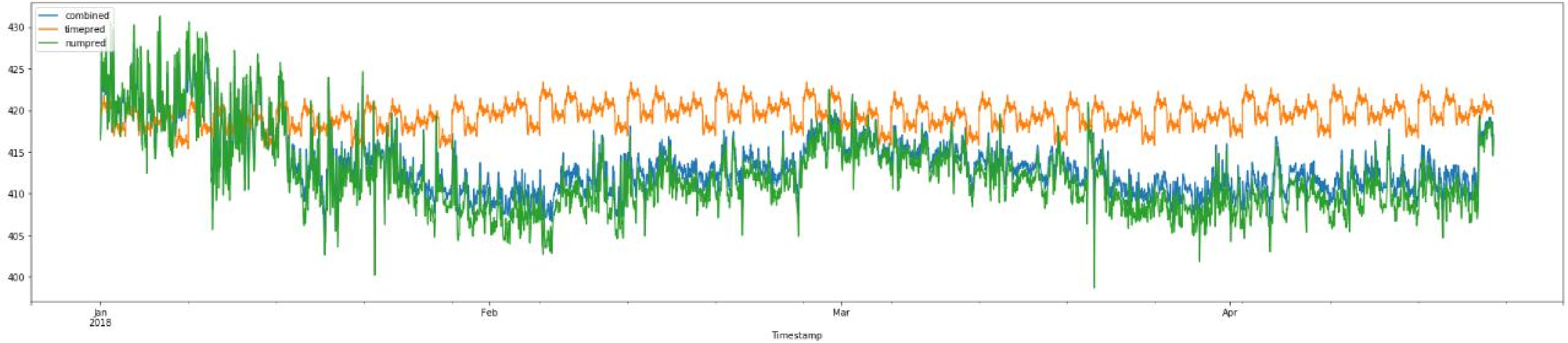
```
#predictions from time features
y_pred_final_time = np.zeros((ts_time.shape[0],len(svrs)))
for i, m in enumerate(svrs):
    y_pred_final_time[:,i]= m.predict(ts_time.values)
y_pred_final_time = y_pred_final_time.mean(axis=1)
```

```
#predictions from sensor variables
y_pred_final_num = beststacked.predict(ts_num.values)
```

```
w1=0.8
w2=1-w1
y_pred_final = y_pred_final_num*w1 + y_pred_final_time*w2
```

```
finalpred = pd.Series(data=y_pred_final,index=ts_time.index)
timepred = pd.Series(data=y_pred_final_time, index=ts_time.index)
numpred = pd.Series(data=y_pred_final_num, index=ts_time.index)
```

I used a linear combination with weights. The time-feature predictions still have an effect but its still considerably low.



Summary

- The final model is a linear combination of;
  - A stacked model trained on sensor variables data with regressors:(AdaBoostRegressor, ExtremeTreesRegressor, XGBoost Tree Regressor) and meta-regressor:Linear Regression
  - A support vecor machines regressor trained on time features data.

Closing Words

- Pre-processing
  - Pre-processing step can be made faster by using dask. Dask scales python natively in local and online clusters.
- Visualisation
  - Findings could easily be present in a more elegant way using a single jupyter notebook and using interative plotting libraries such as plotly.

Closing Words

- Decomposition
    - Decomposition was a necessity in order to have equal dimensionality in the train and prediction tests. PCA was applied as a quick hot-fix to iterate and prototype. My personal favourite as a decomposition method actually is auto-encoders. Auto-encoders are a special type of neural networks that by re-creating original features they learn complex relationships between features. Thus, the information loss while reducing dimension is minimised.

Closing Words

- Modeling
  - Even though stochastic forecasting is eliminated in this case study. It is still worth looking more into. There are very niche and easy to use helping libraries such as pmdarima(bringing R's auto arima to python) and tsfresh(which automatically builds features out of a single pandas.Series object.)
  - Neural networks, LSTMs specifically, are worth looking into given their nature of learning from past sequences.

Closing Words

- Hyperparameter Optimization
  - My main goal was to prototype as rapid as I can while putting out a solid solution. Hyperparameter optimization generally takes significantly longer times compared to fitting single models. However, there is still room for improvement with tuning choosen models' parameters.

Closing Words

- TimeSeriesSplit and fit_model_cv
  - My custom function first fits n models on TimeSeriesSplit cross-validation and then takes their average as the final prediction. However, due to the nature of the split, each fitted model sees also the pre-trained data. There may be a way of better combining the results of the n models rather than taking average. One candidate might be weighted average.