Practical Time-Series Forecast and Anomaly Detection in Python

Ahmed Abdulaal
Data Scientist,

Agenda

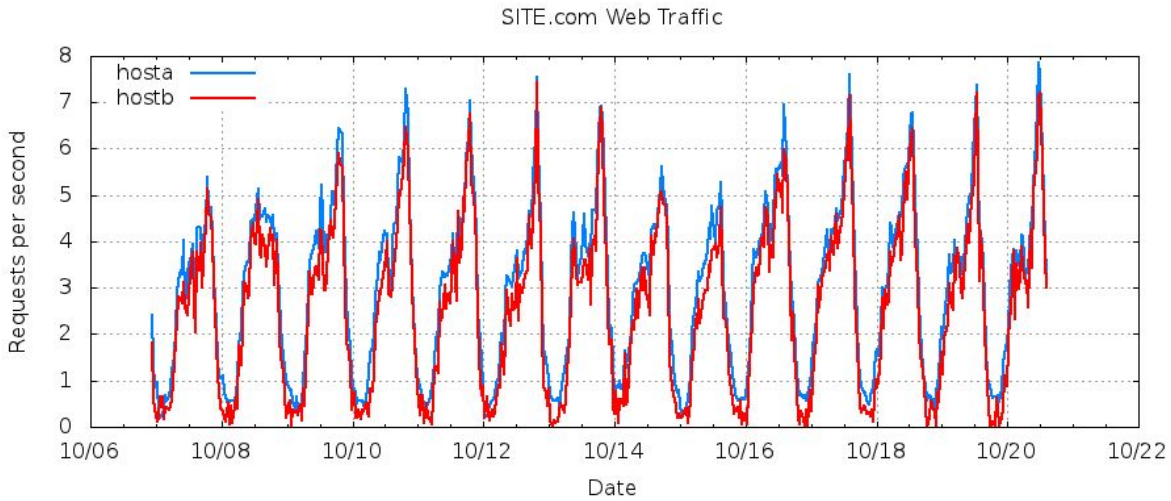
1. Time-Series (TS) Data
2. Intro to ARIMA models
3. Intro to Bayesian methods
4. Intro to Machine Learning for TS
5. Intro to Deep Learning models

6. Anomalies and Challenges

7. Hyperparameter Optimization
8. Ensembling
9. Evaluation
10. Visualization

What is TS?

- Timestamps?
- Sequence?
- Equal intervals?
- Patterns?



Dataset

- Real Ad Exchange Data
- Online advertisement clicking rates
  - Cost-Per-Click (CPC)
- Source: Numenta Anomaly Benchmark
  - https://github.com/numenta/NAB

Data Preparation

- **Import to Python**
- Convert to TS
- Missing Values
- Partitioning

In [15]:
```
import pandas as pd
ts = pd.read_csv('Data\exchange-2_cpc_results.csv')
ts.head()
```

Out[15]:

	timestamp	value
0	2011-07-01 00:00:01	0.081965
1	2011-07-01 01:00:01	0.098972
2	2011-07-01 02:00:01	0.065314
3	2011-07-01 03:00:01	0.070663
4	2011-07-01 04:00:01	0.102490

Data Preparation

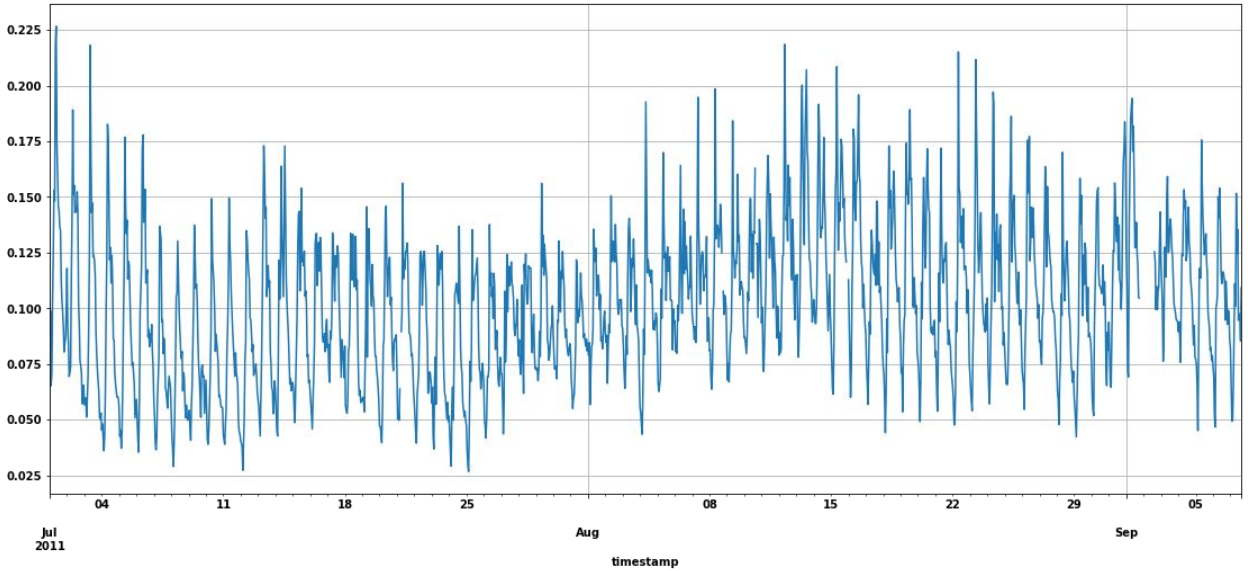
- Import to Python
- **Convert to TS**
- Missing Values
- Partitioning

In [16]:
```
ts['timestamp'] = pd.to_datetime(ts['timestamp'])
ts.set_index('timestamp', drop=True, inplace=True)
ts = ts.resample('h').mean()
ts.columns = ['cpc']
ts = ts['cpc']
ts.head()
```

Out[16]:
```
timestamp
2011-07-01 00:00:00    0.081965
2011-07-01 01:00:00    0.098972
2011-07-01 02:00:00    0.065314
2011-07-01 03:00:00    0.070663
2011-07-01 04:00:00    0.102490
Freq: H, Name: cpc, dtype: float64
```

Data Preparation

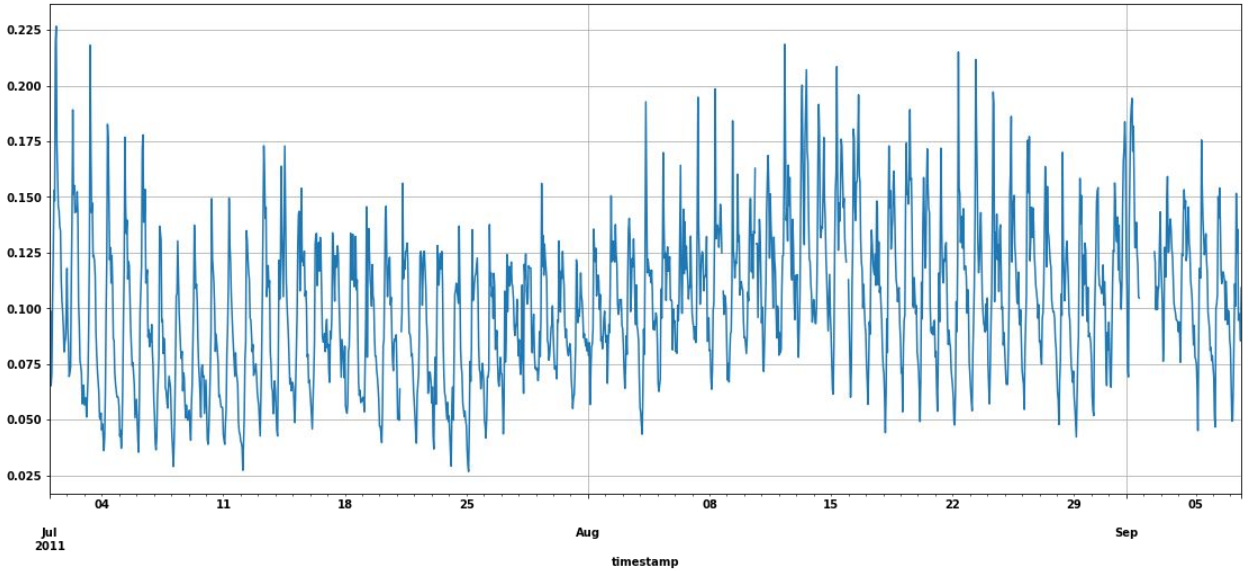
- Import to Python
- Convert to TS
- **Missing Values**
- Partitioning




```
In [22]: ts_clean = ts.loc[ts.index < '2011-09-02']
         ts_clean.fillna(method='pad', inplace=True)
```

Data Preparation

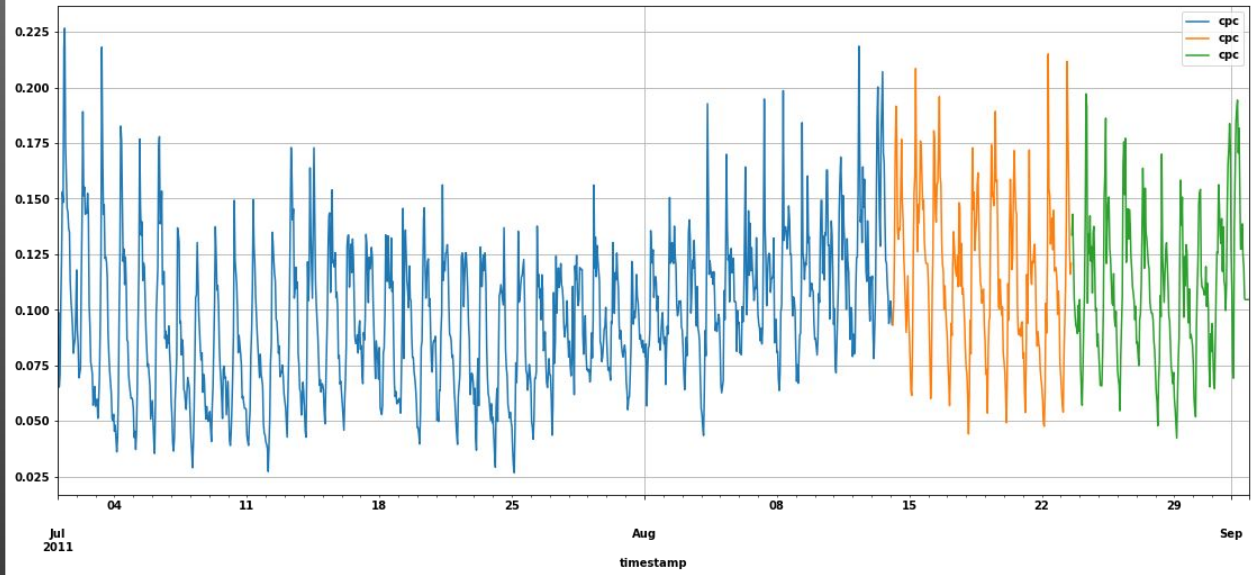
- Import to Python
- Convert to TS
- **Missing Values**
- Partitioning




```
In [22]: ts_clean = ts.loc[ts.index < '2011-09-02']
         ts_clean.fillna(method='pad', inplace=True)
```

Data Preparation

- Import to Python
- Convert to TS
- Missing Values
- **Partitioning**

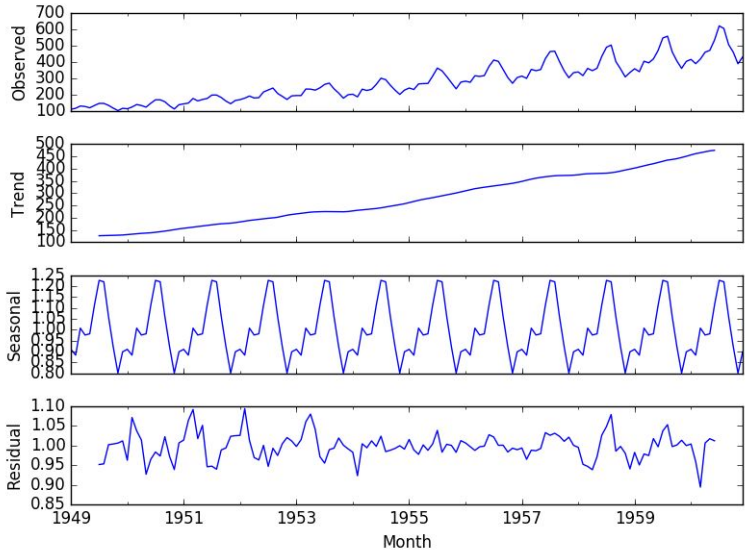



```
In [26]: n_obs = ts_clean.shape[0]
         split1 = ts_clean.index[int(0.7*n_obs)]
         split2 = ts_clean.index[int(0.85*n_obs)]
         train_ts = ts_clean.loc[ts_clean.index <= split1]
         val_ts = ts_clean.loc[(ts_clean.index > split1) & (ts_clean.index <= split2)]
         test_ts = ts_clean.loc[ts_clean.index > split2]
```

Statistical Model: ARIMA

- **A**uto**R**egressive **I**ntegrated **M**oving **A**verage
- Time-Series Decomposition:
  - Trend
  - Seasonality
  - Noise
- Linear Parameters

$$y_t = \Phi_1 y_{t-1} + \Phi_2 y_{t-2} + \ldots + \Phi_p y_{t-p}$$



$$y(t) = Trend + Seasonality + Noise$$

$$y(t) = Trend \times Seasonality \times Noise$$

Statistical Model: SARIMA

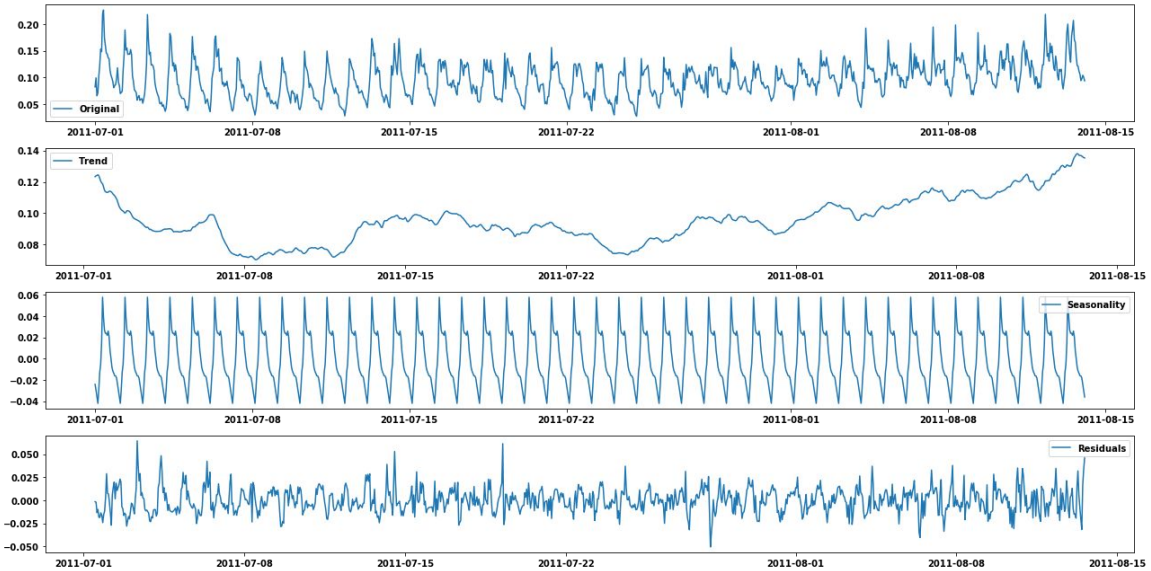
- **S**easonal **A**uto**R**egressive **I**ntegrated **M**oving **A**verage
- Implementation in python StatsModels library
  - http://www.statsmodels.org/stable/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html?highlight=sarimax#statsmodels.tsa.statespace.sarimax.SARIMAX

$$ARIMA(p, d, q) \times (P, D, Q)_S$$

Statistical Model: SARIMA

- **Time-Series Decomposition**
- Parameter Selection
- Training
- Forecasting
- Validation

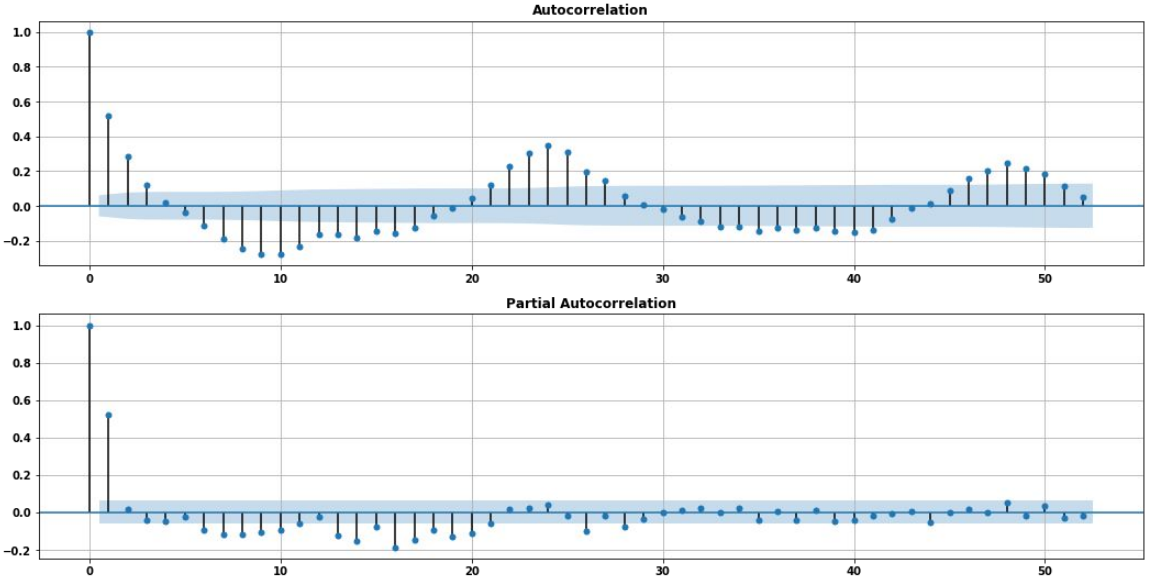
```
In [6]: import matplotlib.pyplot as plt
        from statsmodels.tsa.seasonal import seasonal_decompose

        decomposition = seasonal_decompose(train_ts, model='additive', freq=24)
        trend = decomposition.trend
        seasonal = decomposition.seasonal
        residual = decomposition.resid
```



Statistical Model: SARIMA

- Time-Series Decomposition
- **Parameter Selection**
  - https://people.duke.edu/~rnau/arimrule.htm
- Training
- Forecasting
- Validation

```
In [7]: from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

        resid = residual.dropna()

        plt.figure(figsize=(18,9))
        plt.subplot(211)
        plot_acf(resid, lags=52, ax=plt.gca())
        plt.grid()
        plt.subplot(212)
        plot_pacf(resid, lags=52, ax=plt.gca())
        plt.grid()
        plt.show()
```



Statistical Model: SARIMA

- Time-Series Decomposition
- Parameter Selection
- **Training**
- Forecasting
- Validation

In [9]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX
import time

model = SARIMAX(train_ts, order=(1, 1, 3), seasonal_order=(2, 1, 0, 24))
start = time.time()
model_fit = model.fit(dis=0)
print('fitting complete after {} seconds'.format(time.time()-start))
```

```
fitting complete after 29.97715663909912 seconds
```

In [19]:
```
model_fit.summary()
```

Out[19]:

Statespace Model Results

Dep. Variable:	cpc	**No. Observations:**	1059
Model:	SARIMAX(1, 1, 3)x(2, 1, 0, 24)	**Log Likelihood**	3020.043
Date:	Sat, 05 Oct 2019	**AIC**	-6026.086
Time:	18:25:45	**BIC**	-5991.497
Sample:	07-01-2011	**HQIC**	-6012.961
	- 08-14-2011		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	-0.5366	0.291	-1.845	0.065	-1.107	0.033
ma.L1	0.0006	0.290	0.002	0.998	-0.568	0.569
ma.L2	-0.4958	0.167	-2.972	0.003	-0.823	-0.169
ma.L3	-0.1895	0.063	-2.998	0.003	-0.313	-0.066
ar.S.L24	-0.6015	0.026	-23.467	0.000	-0.652	-0.551
ar.S.L48	-0.2885	0.028	-10.194	0.000	-0.344	-0.233
sigma2	0.0002	5.4e-06	31.105	0.000	0.000	0.000

Ljung-Box (Q):	86.80	**Jarque-Bera (JB):**	153.78
Prob(Q):	0.00	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.27	**Skew:**	-0.08
Prob(H) (two-sided):	0.03	**Kurtosis:**	4.88

Statistical Model: SARIMA

- Time-Series Decomposition
- Parameter Selection
- Training
- **Forecasting**
- Validation

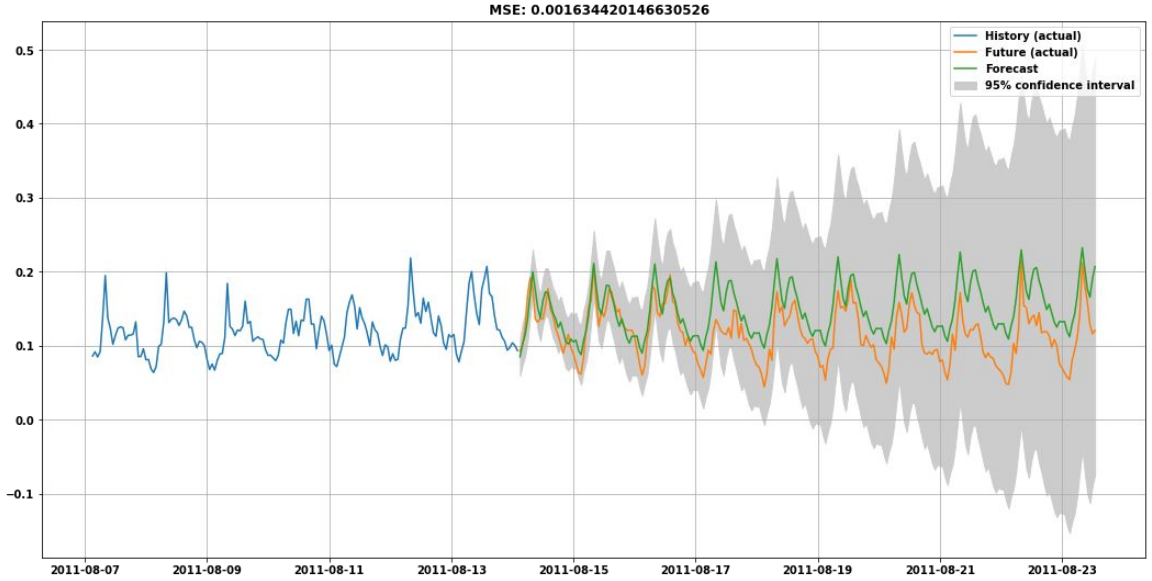
In [15]:
```
f_steps = val_ts.shape[0]
results = model_fit.get_forecast(f_steps)

forecasts = pd.concat([results.predicted_mean, results.conf_int(alpha=0.05)], axis=1)
forecasts.columns = ['Forecasts', 'Lower 95% CI', 'Upper 95% CI']

forecasts.head()
```

Out[15]:

	Forecasts	Lower 95% CI	Upper 95% CI
2011-08-14 03:00:00	0.084992	0.059580	0.110405
2011-08-14 04:00:00	0.102088	0.074073	0.130104
2011-08-14 05:00:00	0.114389	0.085629	0.143149
2011-08-14 06:00:00	0.135657	0.106544	0.164770
2011-08-14 07:00:00	0.169589	0.139945	0.199233

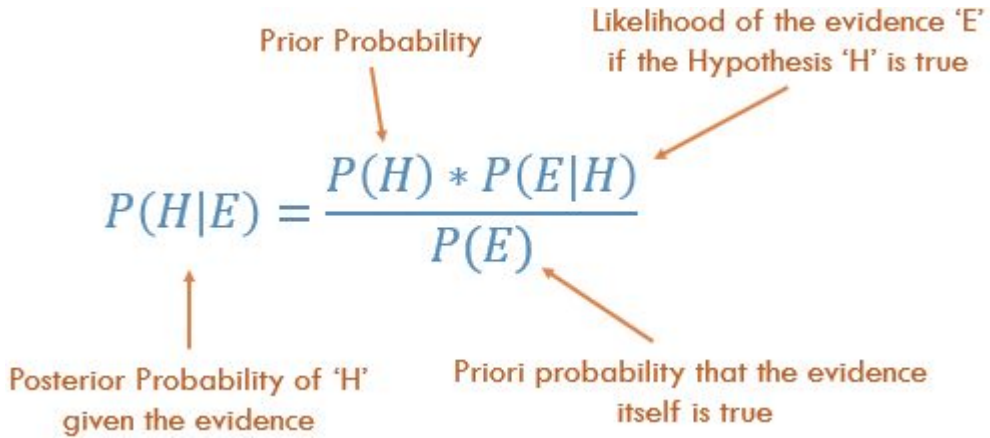
Statistical Model: SARIMA

- Time-Series Decomposition
- Parameter Selection
- Training
- Forecasting
- **Validation**



Bayesian Models

- Distributions vs. point estimates
- Non-Gaussian errors
- Data limitations
- Incorporate prior knowledge
- Bayes vs MLE
- MCMC



Bayesian Model: Probabilistic ARIMA

- Implementation in Python PyFlux library
  - Similar to StatsModels
  - https://pyflux.readthedocs.io/en/latest/arima.html

Bayesian Model: Probabilistic ARIMA

- Time-Series Decomposition
- **Parameter Selection**
- Training
- Bayesian Inference
- Forecasting
- Validation

```
model = pf.ARIMA(data=train_ts.values, ar=24, integ=1, ma=3, family=pf.Normal())
```

Bayesian Model: Probabilistic ARIMA

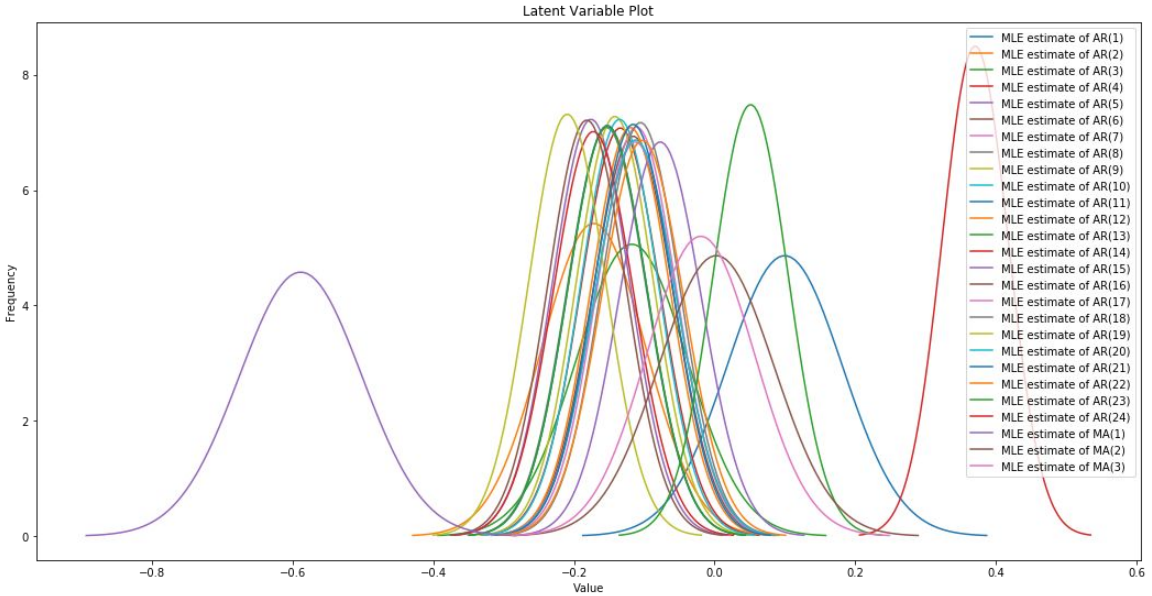
- Time-Series Decomposition
- Parameter Selection
- **Training**
- Bayesian Inference
- Forecasting
- Validation

```
In [79]: import pyflux as pf
         import time

         model = pf.ARIMA(data=train_ts.values, ar=24, integ=1, ma=3, family=pf.Normal())
         start = time.time()
         model_fit = model.fit(dis=0)
         print('fitting complete after {} seconds'.format(time.time()-start))

         fitting complete after 30.95735502243042 seconds
```

```
In [80]: model_fit.summary()

Normal ARIMA(24,1,3)
============================================================ ==============================================
Dependent Variable: Differenced Series                       Method: MLE
Start Date: 24                                               Log Likelihood: 3042.2644
End Date: 1058                                               AIC: -6026.5287
Number of observations: 1034                                 BIC: -5883.2342
=============================================================================================================
Latent Variable                          Estimate   Std Error  z        P>|z|    95% C.I.
========================================= ========== ========== ======== ======== ========================
Constant                                 0.0001     0.0002     0.6578   0.5107   (-0.0002 | 0.0004)
AR(1)                                    0.0999     0.082      1.2176   0.2234   (-0.0609 | 0.2607)
AR(2)                                    -0.1718    0.0736     -2.3355  0.0195   (-0.3159 | -0.0276)
AR(3)                                    -0.1176    0.0788     -1.4924  0.1356   (-0.2721 | 0.0368)
AR(4)                                    -0.134     0.0564     -2.3785  0.0174   (-0.2445 | -0.0236)
AR(5)                                    -0.1755    0.0552     -3.1809  0.0015   (-0.2837 | -0.0674)
AR(6)                                    -0.116     0.0575     -2.0161  0.0438   (-0.2287 | -0.0032)
AR(7)                                    -0.1108    0.0562     -1.9705  0.0488   (-0.221 | -0.0006)
AR(8)                                    -0.1533    0.0563     -2.7225  0.0065   (-0.2637 | -0.0429)
AR(9)                                    -0.1418    0.0548     -2.5868  0.0097   (-0.2492 | -0.0344)
AR(10)                                   -0.1351    0.0552     -2.4473  0.0144   (-0.2432 | -0.0269)
AR(11)                                   -0.1524    0.056      -2.7213  0.0065   (-0.2622 | -0.0426)
AR(12)                                   -0.1197    0.0563     -2.1259  0.0335   (-0.23 | -0.0093)
AR(13)                                   -0.1524    0.0561     -2.7144  0.0066   (-0.2624 | -0.0424)
AR(14)                                   -0.1724    0.0569     -3.0321  0.0024   (-0.2838 | -0.061)
AR(15)                                   -0.0772    0.0584     -1.3227  0.1859   (-0.1916 | 0.0372)
AR(16)                                   -0.1817    0.0553     -3.2853  0.001    (-0.29 | -0.0733)
AR(17)                                   -0.1126    0.0584     -1.9291  0.0537   (-0.2269 | 0.0018)
AR(18)                                   -0.1054    0.0556     -1.8941  0.0582   (-0.2144 | 0.0037)
AR(19)                                   -0.2096    0.0545     -3.8434  0.0001   (-0.3164 | -0.1027)
AR(20)                                   -0.1125    0.0581     -1.9368  0.0528   (-0.2263 | 0.0013)
AR(21)                                   -0.1157    0.0559     -2.0716  0.0383   (-0.2252 | -0.0062)
AR(22)                                   -0.1025    0.0581     -1.7643  0.0777   (-0.2164 | 0.0114)
AR(23)                                   0.0513     0.0533     0.9629   0.3356   (-0.0532 | 0.1558)
AR(24)                                   0.3707     0.0469     7.8981   0.0      (0.2787 | 0.4627)
MA(1)                                    -0.5886    0.0871     -6.7552  0.0      (-0.7594 | -0.4178)
MA(2)                                    0.0025     0.082      0.0309   0.9753   (-0.1582 | 0.1633)
MA(3)                                    -0.0197    0.0767     -0.2567  0.7974   (-0.1701 | 0.1307)
Normal Scale                             0.0128
=============================================================================================================
```

Bayesian Model: Probabilistic ARIMA

- Time-Series Decomposition
- Parameter Selection
- Training
- **Bayesian Inference**
- Forecasting
- Validation




```
In [83]: import numpy as np

         model.plot_z(indices=list(np.arange(1,28)), figsize=(18,9))
```

Bayesian Model: Probabilistic ARIMA

- Time-Series Decomposition
- Parameter Selection
- Training
- Bayesian Inference
- **Forecasting**
- Validation

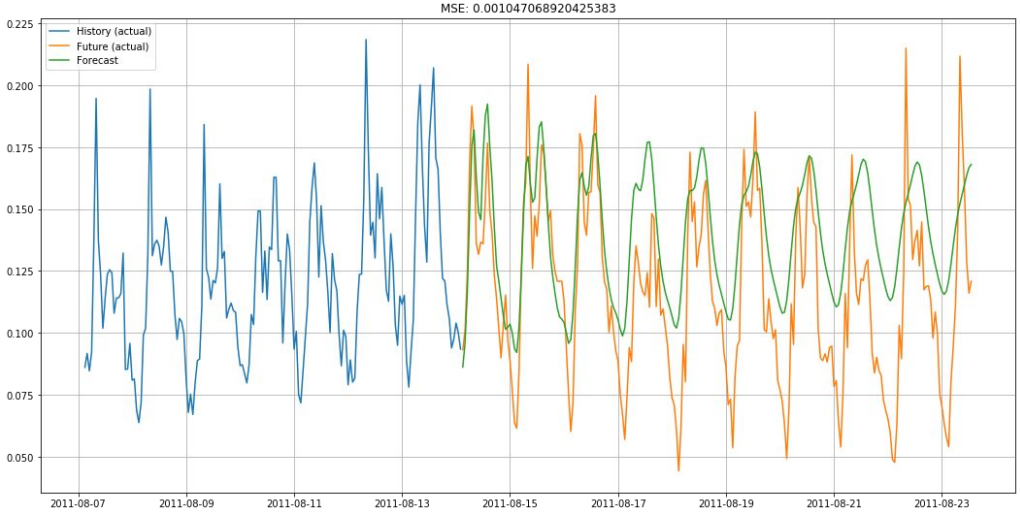
In [70]:
```
f_steps = val_ts.shape[0]
results = model.predict(h=f_steps, intervals=True)
results = results + train_ts[-1]

forecasts = results[['Differenced Series', '5% Prediction Interval', '95% Prediction Interval']]
forecasts.columns = ['Forecasts', 'Lower 95% CI', 'Upper 95% CI']
forecasts.index = val_ts.index

forecasts.head()
```

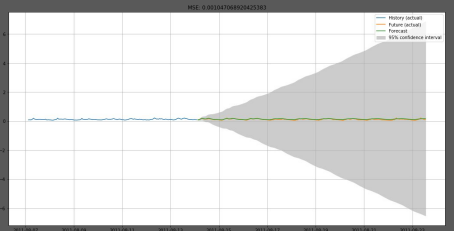
Out[70]:

timestamp	Forecasts	Lower 95% CI	Upper 95% CI
2011-08-14 03:00:00	0.086222	0.065016	0.107027
2011-08-14 04:00:00	0.103188	0.082447	0.124323
2011-08-14 05:00:00	0.114133	0.092792	0.135328
2011-08-14 06:00:00	0.124825	0.102840	0.146578
2011-08-14 07:00:00	0.120846	0.099348	0.142722

Bayesian Model: Probabilistic ARIMA

- Time-Series Decomposition
- Parameter Selection
- Training
- Bayesian Inference
- Forecasting
- **Validation**





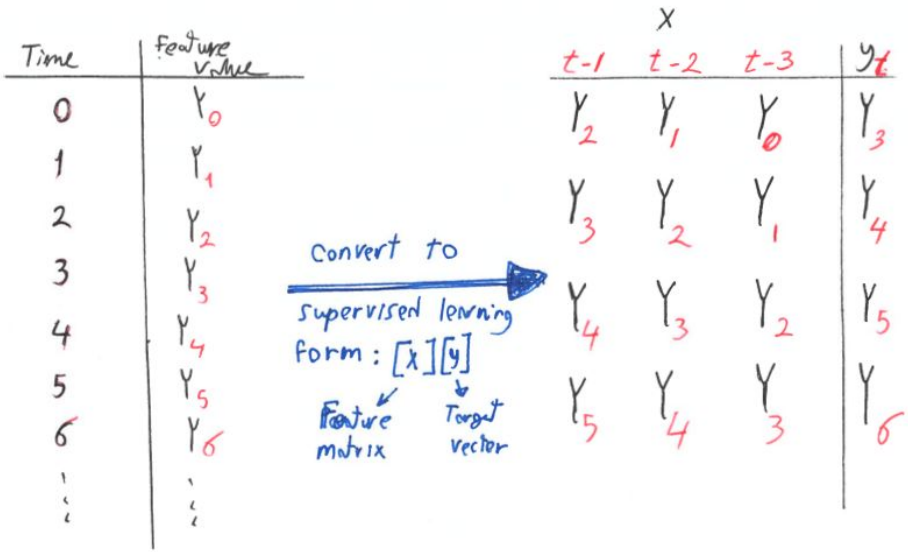
Machine Learning Models: Data Preparation

ML limitations vs Classical TS:

- Extrapolations
- Decomposition
- Distributions

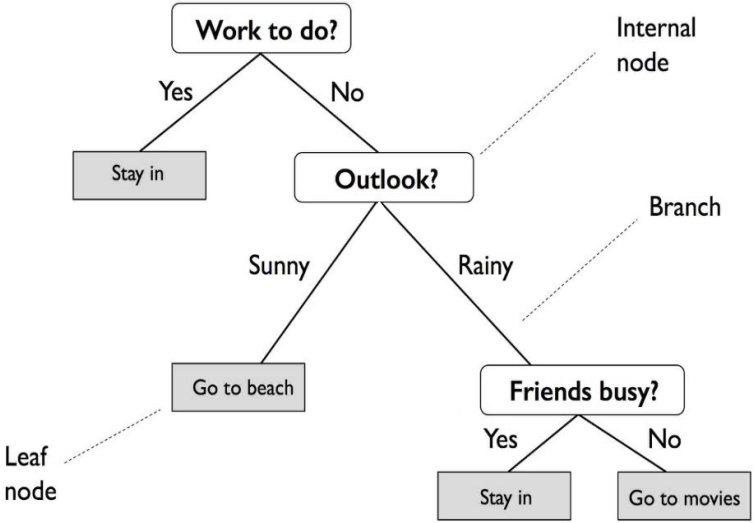
Workarounds:

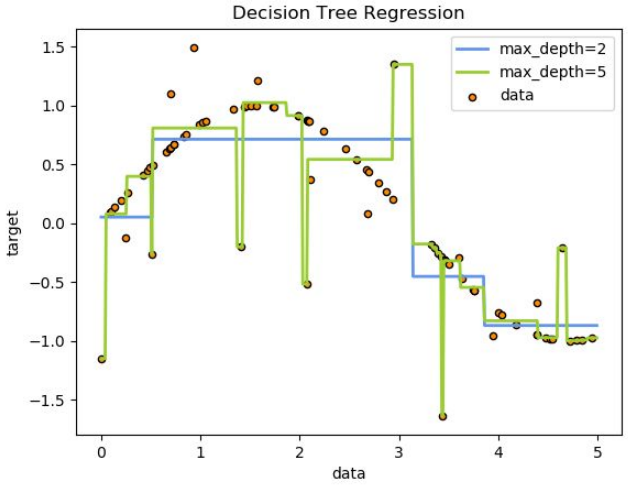
- Differencing
- Feature Extraction
- Post-Processing
- Loss-Function Definition



Machine Learning Models: Decision Trees

Open source library for ML: https://scikit-learn.org/stable/index.html





Machine Learning Model: Regression Trees

- **Pre-Processing**
- Training
- Forecasting
- Validation

In [55]:
```
import numpy as np

train_ts_processed = train_ts.to_frame().diff()
for l in [1,2,3,24,48]:
    train_ts_processed['lag-{}'.format(l)] = train_ts_processed['cpc'].shift(periods=l)
train_ts_processed.dropna(inplace=True, axis=0)
y = train_ts_processed['cpc'].values
X = train_ts_processed.drop('cpc', axis=1).values
print('feature matrix:\n',X)
print('Target vector:\n',y.reshape(-1,1))
```

```
feature matrix:
 [[-0.00882875  0.00263513  0.00571875 -0.02697922  0.0170075 ]
 [ 0.0020447  -0.00882875  0.00263513 -0.0066526  -0.03365829]
 [ 0.00111814  0.0020447  -0.00882875 -0.01484984  0.00534889]
 ...
 [ 0.00356583 -0.01195189 -0.00545777 -0.00321909 -0.0195469 ]
 [ 0.00650718  0.00356583 -0.01195189  0.00358975  0.01004682]
 [-0.00387172  0.00650718  0.00356583 -0.02673232 -0.00897452]]
Target vector:
 [[ 0.0020447 ]
 [ 0.00111814]
 [-0.00859562]
 ...
 [ 0.00650718]
 [-0.00387172]
 [-0.00665327]]
```

Machine Learning Model: Regression Trees

- Pre-Processing
- **Training**
- Forecasting
- Validation

In [60]:
```
from sklearn import tree
import time

clf = tree.DecisionTreeRegressor()
start = time.time()
clf = clf.fit(X, y)
print('fitting complete after {} seconds'.format(time.time()-start))
```

```
fitting complete after 0.008983373641967773 seconds
```

In Reality:

- Ensembling (RFs or GBTs)
- Hyperparameter Tuning via GridSearch CV

Machine Learning Model: Regression Trees

- Pre-Processing
- Training
- **Forecasting**
- Validation

In [61]:
```
from datetime import timedelta

forecasts = pd.DataFrame(columns=train_ts_processed.columns, index=val_ts.index)
attached_df = pd.concat([train_ts_processed[-48:], forecasts], axis=0)
for t in forecast_df.index:
    for l in [1,2,3,24,48]:
        forecasts.loc[t, 'lag-{}'.format(l)] = attached_df.loc[t - timedelta(hours=l), 'cpc']
    step_X = forecasts.loc[t].values[1:].reshape(1,-1)
    step_y = clf.predict(step_X)[0]
    attached_df.loc[t, 'cpc'], forecasts.loc[t, 'cpc'] = step_y, step_y

forecasts.head()
```

Out[61]:

timestamp	cpc	lag-1	lag-2	lag-3	lag-24	lag-48
2011-08-14 03:00:00	0.00191241	-0.00665327	-0.00387172	0.00650718	-0.010359	0.00154876
2011-08-14 04:00:00	0.0195445	0.00191241	-0.00665327	-0.00387172	0.0144919	0.0268069
2011-08-14 05:00:00	0.0291833	0.0195445	0.00191241	-0.00665327	0.0126438	0.0150717
2011-08-14 06:00:00	0.0602823	0.0291833	0.0195445	0.00191241	0.0417596	0.000228741
2011-08-14 07:00:00	-0.0174822	0.0602823	0.0291833	0.0195445	0.0379464	0.0308431

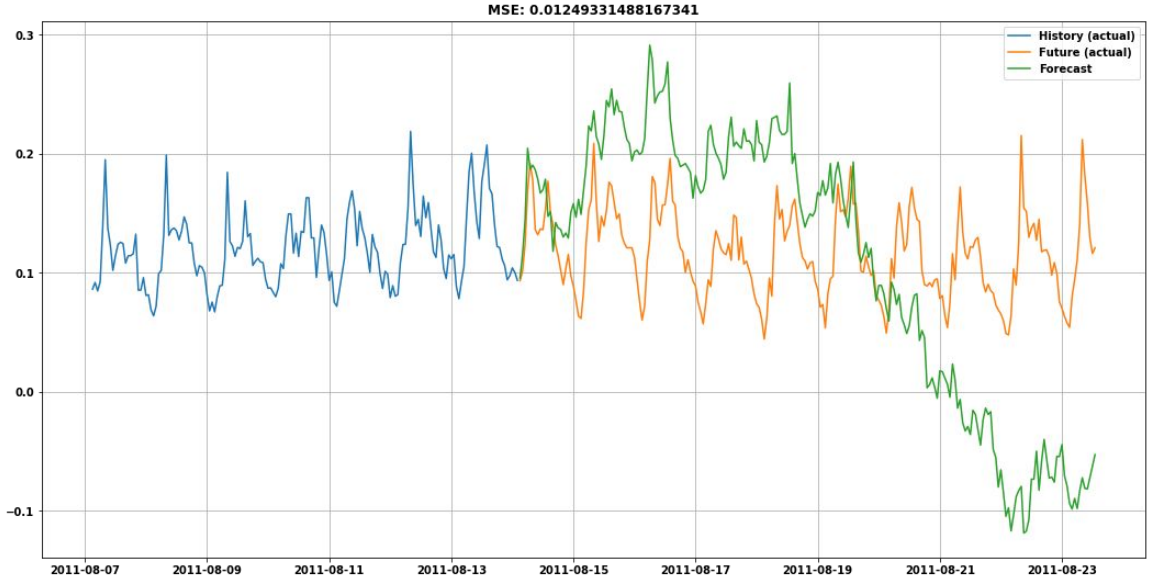
Note that the forecasts need to be re-trended

In [62]:
```
forecasts['cpc'][0] = forecasts['cpc'][0] + train_ts[-1]
forecasts['cpc'] = forecasts['cpc'].cumsum()
forecasts['cpc'].head()
```

Out[62]:
```
timestamp
2011-08-14 03:00:00    0.095362
2011-08-14 04:00:00    0.114906
2011-08-14 05:00:00     0.14409
2011-08-14 06:00:00    0.204372
2011-08-14 07:00:00     0.18689
Freq: H, Name: cpc, dtype: object
```

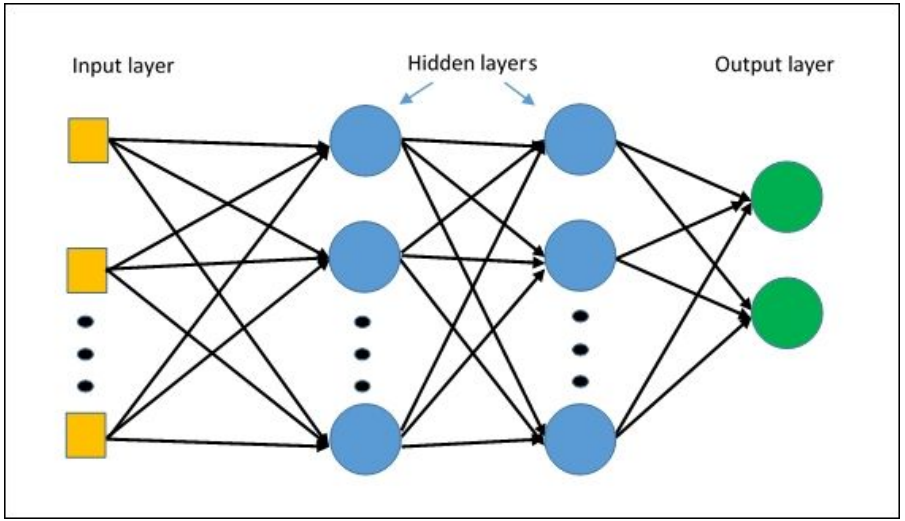
Machine Learning Model: Regression Trees

- Pre-Processing
- Training
- Forecasting
- **Validation**



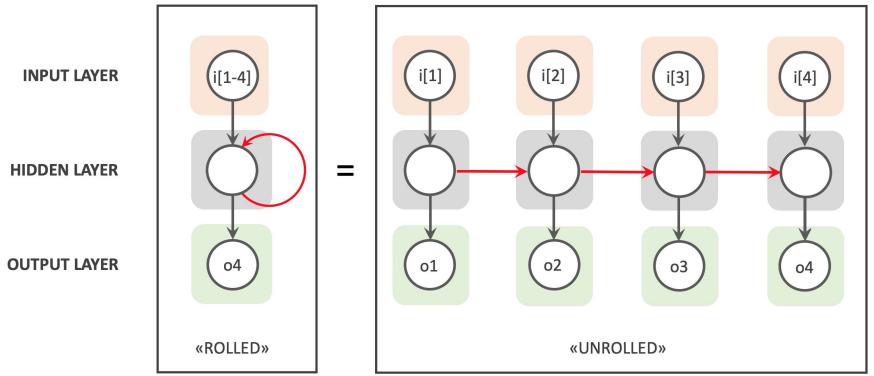
Deep Learning Models: MLP

- **M**ulti-**L**ayer **P**erceptron
- Classical form of ANN
- Feed-forward
- No memory



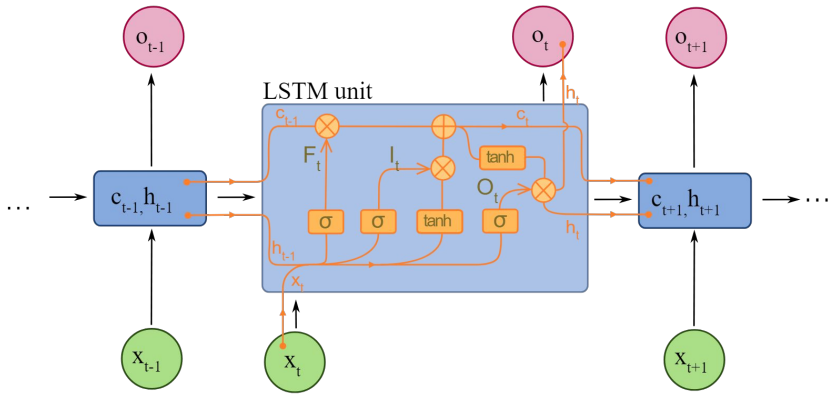
Deep Learning Models: RNN

- Nodes connected along temporal sequence
- Internal state (memory)
- Processes sequences of inputs
- Ideal for NLP
- Problems:
  - Vanishing gradient
  - Exploding gradient



Deep Learning Models: LSTM

- Long memory
- Several gates at the unit (forget gates)
- Remove or add information to the unit's state



Deep Learning Model: LSTM

- **Pre-Processing**
- Architecture
- Learning
- Forecasting
- Validation

In [34]:
```
import numpy as np

train_ts_processed = train_ts.to_frame()
for l in [1,2,3,24,48]:
    train_ts_processed['lag-{}'.format(l)] = train_ts_processed['cpc'].shift(periods=l)
train_ts_processed.dropna(inplace=True, axis=0)
y = train_ts_processed['cpc'].values
X = train_ts_processed.drop('cpc', axis=1).values.reshape(train_ts_processed.shape[0], 5, 1)
print('feature matrix:\n',X)
print('Target vector:\n',y.reshape(-1,1))
```

Deep Learning Model: LSTM

- Pre-Processing
- **Architecture**
- Learning
- Forecasting
- Validation

In [28]:
```
from keras.models import Sequential
from keras.layers.recurrent import LSTM
from keras.layers import Dense

bs = 1
model = Sequential()
model.add(LSTM(10, activation='relu', batch_input_shape=(bs, 5, 1), stateful=True, return_sequences=True))
model.add(LSTM(10, activation='relu', input_shape=(5, 1), stateful=True, return_sequences=False))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')
```

Deep Learning Model: LSTM

- Pre-Processing
- Architecture
- **Learning**
- Forecasting
- Validation

```
In [29]: model.fit(X, y, batch_size=bs, epochs=10, shuffle=False, verbose=1)

         Epoch 1/10
         1011/1011 [==============================] - 36s 36ms/step - loss: 9.3838e-04
         Epoch 2/10
         1011/1011 [==============================] - 34s 34ms/step - loss: 6.0068e-04
         Epoch 3/10
         1011/1011 [==============================] - 33s 33ms/step - loss: 5.0695e-04 0s - loss: 5.03
         Epoch 4/10
         1011/1011 [==============================] - 35s 34ms/step - loss: 4.4176e-04
         Epoch 5/10
         1011/1011 [==============================] - 35s 34ms/step - loss: 3.6697e-04
         Epoch 6/10
         1011/1011 [==============================] - 34s 33ms/step - loss: 3.0328e-04
         Epoch 7/10
         1011/1011 [==============================] - 33s 32ms/step - loss: 2.8482e-04
         Epoch 8/10
         1011/1011 [==============================] - 33s 32ms/step - loss: 2.7504e-04
         Epoch 9/10
         1011/1011 [==============================] - 33s 32ms/step - loss: 2.7244e-04
         Epoch 10/10
         1011/1011 [==============================] - 32s 32ms/step - loss: 2.7314e-04

Out[29]: <keras.callbacks.History at 0x1bbe75a9408>
```

Deep Learning Model: LSTM

- Pre-Processing
- Architecture
- Learning
- **Forecasting**
- Validation

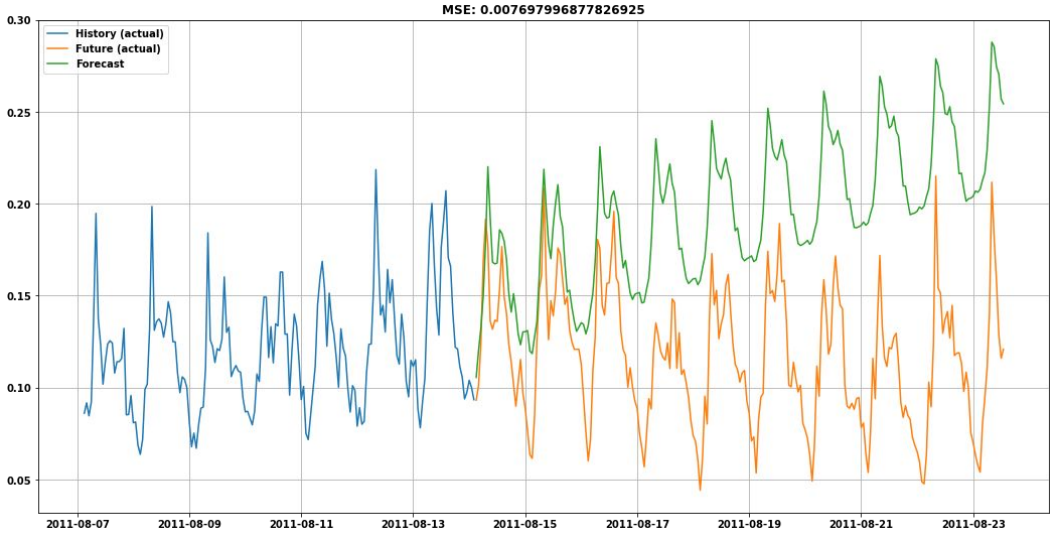
In [30]:
```
from datetime import timedelta

forecasts = pd.DataFrame(columns=train_ts_processed.columns, index=val_ts.index)
attached_df = pd.concat([train_ts_processed[-48:], forecasts], axis=0)
for t in forecasts.index:
    for l in [1,2,3,24,48]:
        forecasts.loc[t, 'lag-{}'.format(l)] = attached_df.loc[t - timedelta(hours=l), 'cpc']
    step_X = forecasts.loc[t].values[1:].reshape(1,5,1)
    step_y = model.predict(step_X, batch_size=bs)[0][0]
    attached_df.loc[t, 'cpc'], forecasts.loc[t, 'cpc'] = step_y, step_y

forecasts.head()
```

Out[30]:

timestamp	cpc	lag-1	lag-2	lag-3	lag-24	lag-48
2011-08-14 03:00:00	0.105557	0.0934496	0.100103	0.103975	0.0781188	0.0816735
2011-08-14 04:00:00	0.1202	0.105557	0.0934496	0.100103	0.0926107	0.10848
2011-08-14 05:00:00	0.132405	0.1202	0.105557	0.0934496	0.105254	0.123552
2011-08-14 06:00:00	0.149551	0.132405	0.1202	0.105557	0.147014	0.123781
2011-08-14 07:00:00	0.178529	0.149551	0.132405	0.1202	0.184961	0.154624

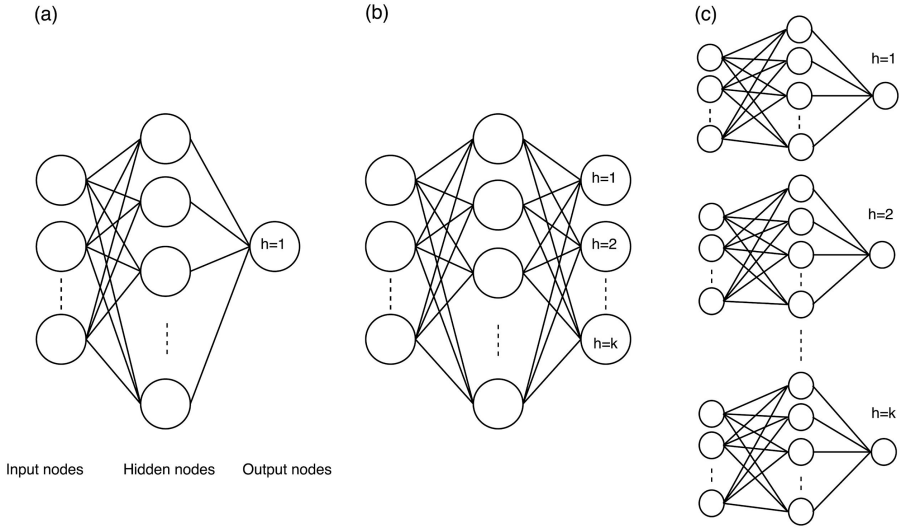
Deep Learning Model: LSTM

- Pre-Processing
- Architecture
- Learning
- Forecasting
- **Validation**



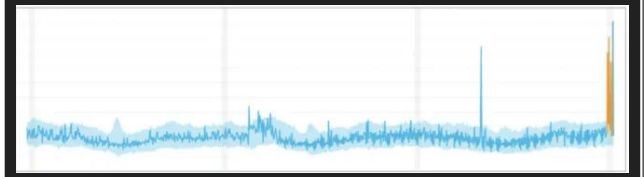
Deep Learning Models: Forecasting approaches

- Iterative Forecasting (a)
- Direct Forecasting (b)
- Multi-Neural Network Forecasting (c)

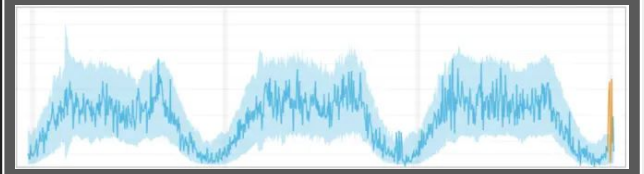


Nature of Anomalies (from anodot docs)

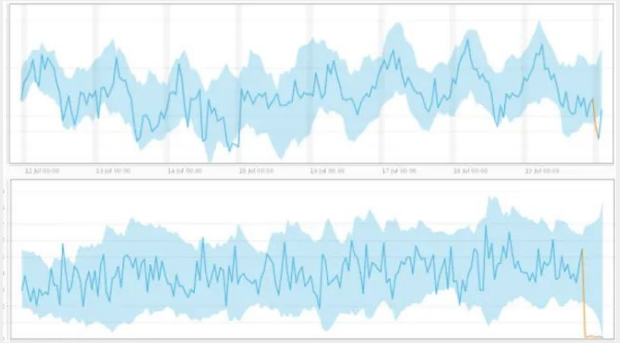
Global



Conditional

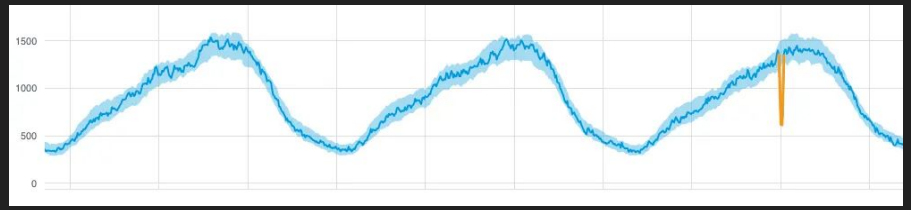


Combined

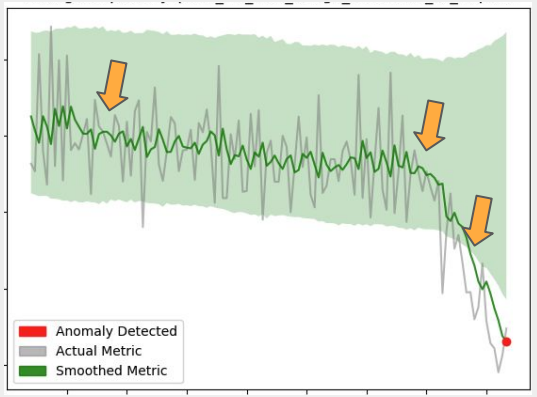


Nature of Anomalies (type 4)

Outliers
(e.g. types 1, 2, & 3)

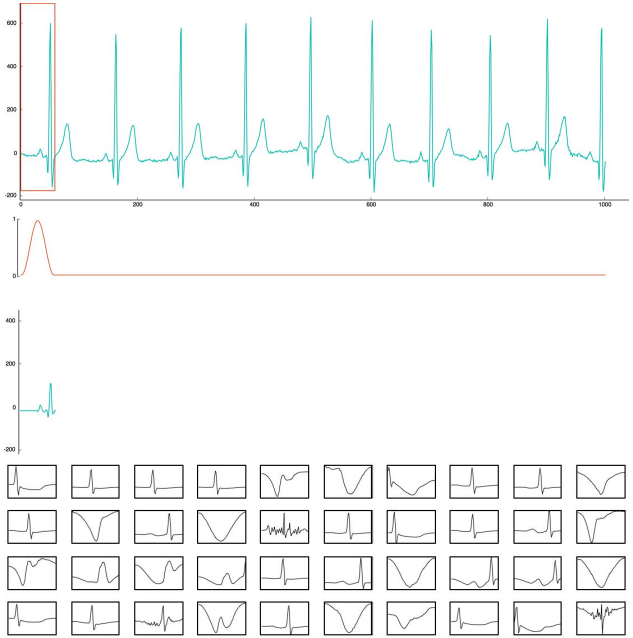


Type 4 - slow bleed
(e.g memory leak)



Nature of Anomalies (challenges)

- Outlier Detection vs Problem Detection
- Confidence Interval approach:
  - Outliers are expected
  - High false positives
  - Non-actionable
  - Contaminated (e.g. continuous learning)
- Parametric modeling approach:
  - Assumptions
  - Speed (TTD)

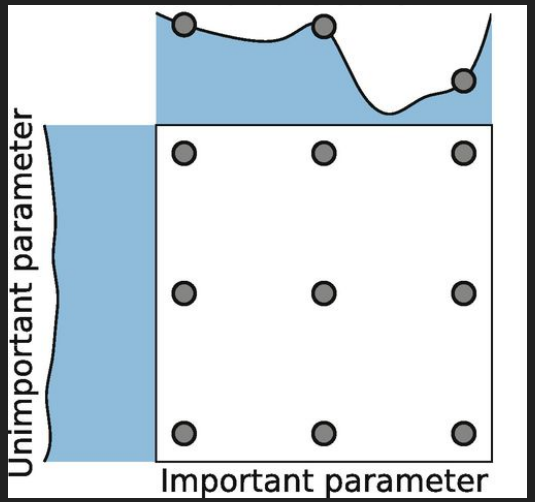


Non-Parametric (e.g. clustering)

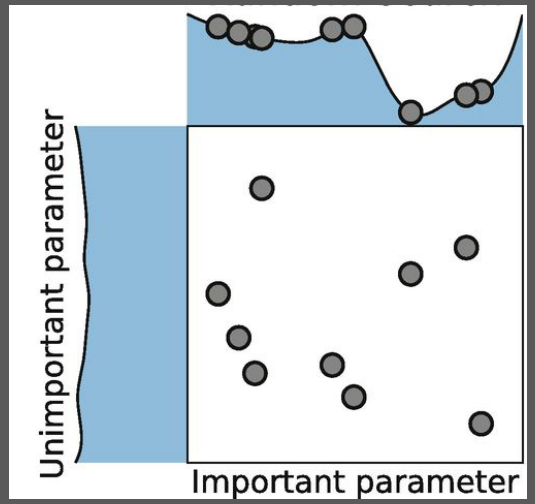
From AnOmaly.io

Hyperparameter Optimization

Grid Search

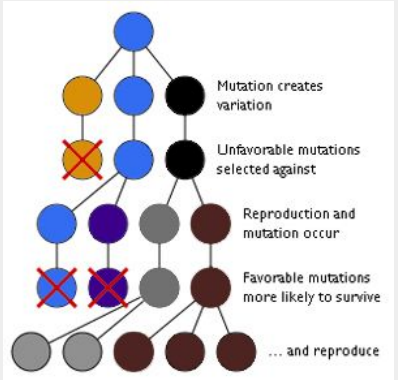


Random Search



Other:

- Evolutionary Algorithms
- Bayesian Methods



Hyperparameter Optimization

- **Option 1:** SciKit-Learn implementation
  - Random Search: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html
  - Grid Search: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
  - Multiprocessing with joblib (https://joblib.readthedocs.io)
  - Requires .fit(), .predict() & .score():
    - Works with SKLearn models
    - Won't work with statsmodels, pyflux, Keras.
- **Option 2:** Wrap models for SciKit-Learn GridSearch
- **Option 3:** Create custom GridSearch

Hyperparameter Optimization: SARIMAX GridSearch

- **Custom GridSearch Module**
  - **Accuracy Metric**
  - Test Function
  - Parameter Feeder/Wrapper
  - GridSearch Object
- Tuning
- Selection & OOS Training
- Forecasting
- Evaluation

$$MAPE = \frac{100\%}{n} \sum \left| \frac{y - \hat{y}}{y} \right|$$

$\hat{y}$ is smaller than the actual value

n = 1 $\hat{y}$ = 10 y = 20

MAPE = 50%

$\hat{y}$ is greater than the actual value

n = 1 $\hat{y}$ = 20 y = 10

MAPE = 100%

In [2]:
```
import numpy as np

def mape(y, yhat):
    return np.mean(abs((yhat - y)/y))
```

Hyperparameter Optimization: SARIMAX GridSearch

- **Custom GridSearch Module**
  - Accuracy Metric
  - **Test Function**
  - Parameter Feeder/Wrapper
  - GridSearch Object
- Tuning
  - Parameter Search
  - Selection & Training
- Forecasting
- Evaluation

In [3]:
```
def train_forecast(train_series, test_series, model, err_fn,
                   kwargs, fkwargs, mkwargs, tkwargs):
    try:
        model_fit = model(train_series, **kwargs, **mkwargs).fit(**fkwargs)
        yhat = model_fit.forecast(len(test_series), **tkwargs)
        return (str(kwargs), {'accuracy': err_fn(test_series.values, yhat),
                'aic': model_fit.aic, 'bic': model_fit.bic})
    except:
        return (str(kwargs), {'accuracy': np.nan,
                'aic': np.nan, 'bic': np.nan})
```

Hyperparameter Optimization: SARIMAX GridSearch

- **Custom GridSearch Module**
  - Accuracy Metric
  - Test Function
  - **Parameter Feeder/Wrapper**
  - GridSearch Object
- Tuning
  - Parameter Search
  - Selection & Training
- Forecasting
- Evaluation

In [4]:
```
def wrap_params(Params, Params_wrapper):
    wrapped = Params_wrapper.copy()
    for key in Params_wrapper.keys():
        typ = type(Params_wrapper[key])
        if typ == str or typ == type(None):
            wrapped[key] = Params[Params_wrapper[key]]
        else:
            wrapped[key] = typ(Params[x] for x in Params_wrapper[key])
    return wrapped
```

Hyperparameter Optimization: SARIMAX GridSearch

- **Custom GridSearch Module**
  - Accuracy Metric
  - Test Function
  - Parameter Feeder/Wrapper
  - **GridSearch Object**
- Tuning
  - Parameter Search
  - Selection & Training
- Forecasting
- Evaluation

In [5]:

```
import pandas as pd
from joblib import Parallel, delayed
import numpy as np
from itertools import product
import warnings

# Gridsearch class object for statsmodels SARIMAX
class GridSearch(object):

    def __init__(self, model, Params, Params_wrapper=None, n_jobs=-1, err_fn=None, fkwargs={}):
        self.model = model
        self.Params_wrapper = Params_wrapper
        self.n_jobs = n_jobs

        # unwrap parameters
        self.Params = list(dict(zip(Params, x)) for x in product(*Params.values()))

        # error function
        self.err_fn = err_fn

        # optional fitting kwargs
        self.fkwargs = fkwargs

    def fit(self, train_series, test_series, mkwargs={}, tkwargs={}):

        #Wrap Parameters if needed
        warnings.filterwarnings("ignore")
        if self.Params_wrapper:
            self.Params = (wrap_params(Params, self.Params_wrapper) for Params in self.Params)
            self.Params = list(self.Params)

        # Parralelize evaluation
        self.output = Parallel(n_jobs=self.n_jobs)(delayed(train_forecast)(
                train_series, test_series, self.model, self.err_fn,
                kwargs, self.fkwargs, mkwargs, tkwargs) for kwargs in self.Params)
        self.output = dict(self.output)
        self.output = pd.DataFrame(self.output).T
        self.output.sort_values('accuracy', inplace=True)

        return self

    def best_param(self, top=1):

        return self.output.index[0:top]


    def best_score(self, top=1):

        return self.output.values[0:top]
```

Hyperparameter Optimization: SARIMAX GridSearch

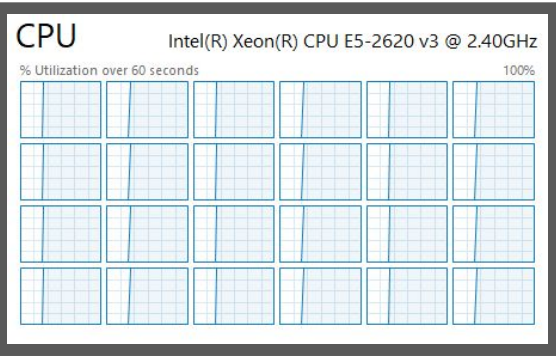
- Custom GridSearch Module
  - Accuracy Metric
  - Test Function
  - Parameter Feeder/Wrapper
  - GridSearch Object
- **Tuning**
  - **Parameter Search**
  - Selection & Training
- Forecasting
- Evaluation

In [6]:
```
from statsmodels.tsa.statespace.sarimax import SARIMAX
import time

model = SARIMAX
Params = {'p': list(np.arange(0, 4)),
          'd': [0, 1],
          'q': list(np.arange(0, 4)),
          'P': list(np.arange(0, 4)),
          'D': [0, 1],
          'Q': list(np.arange(0, 4)),
          'S': [24]}
Params_wrapper = {'order': ('p', 'd', 'q'), 'seasonal_order': ('P', 'D', 'Q', 'S')}
gs = GridSearch(model, Params, Params_wrapper=Params_wrapper, n_jobs=-1, err_fn=mape,
                fkwargs={'method':'nm', 'disp':False})
combinations = len(gs.Params)
print('This will try out {} combinations of model parameters'.format(len(gs.Params)))
```

```
This will try out 1024 combinations of model parameters
```

In [7]:
```
# Apply grid search (best run this on a machine with multiple cores for speed)
start = time.time() # record start time
gs.fit(train_ts, val_ts)
print('{0} models tested in {1} minutes'.format(combinations, (time.time()-start)/60))
```

```
1024 models tested in 17.198094395796456 minutes
```



Hyperparameter Optimization: SARIMAX GridSearch

- Custom GridSearch Module
  - Accuracy Metric
  - Test Function
  - Parameter Feeder/Wrapper
  - GridSearch Object
- **Tuning**
  - Parameter Search
  - **Selection & Training**
- Forecasting
- Evaluation

In [11]:
```
gs.output.head()
```

Out[11]:

	accuracy	aic	bic
{'order': (1, 0, 0), 'seasonal_order': (0, 1, 3, 24)}	0.137611	-6104.469639	-6079.758855
{'order': (3, 0, 0), 'seasonal_order': (0, 1, 3, 24)}	0.138571	-6129.730951	-6095.135854
{'order': (2, 0, 0), 'seasonal_order': (0, 1, 3, 24)}	0.139729	-6123.975556	-6094.322615
{'order': (3, 0, 0), 'seasonal_order': (0, 1, 2, 24)}	0.140001	-6137.131808	-6107.478868
{'order': (2, 0, 1), 'seasonal_order': (0, 1, 3, 24)}	0.140437	-6148.811409	-6114.216312

In [15]:
```
is_ts = train_ts.append(val_ts)

model = SARIMAX(is_ts, order=(1, 0, 0), seasonal_order=(0, 1, 3, 24))
start = time.time()
model_fit = model.fit(dis=0)
print('fitting complete after {} seconds'.format(time.time()-start))
```

```
fitting complete after 94.88529920578003 seconds
```

Hyperparameter Optimization: SARIMAX GridSearch

- Custom GridSearch Module
  - Accuracy Metric
  - Test Function
  - Parameter Feeder/Wrapper
  - GridSearch Object
- Tuning
  - Parameter Search
  - Selection & Training
- **Forecasting**
- Evaluation

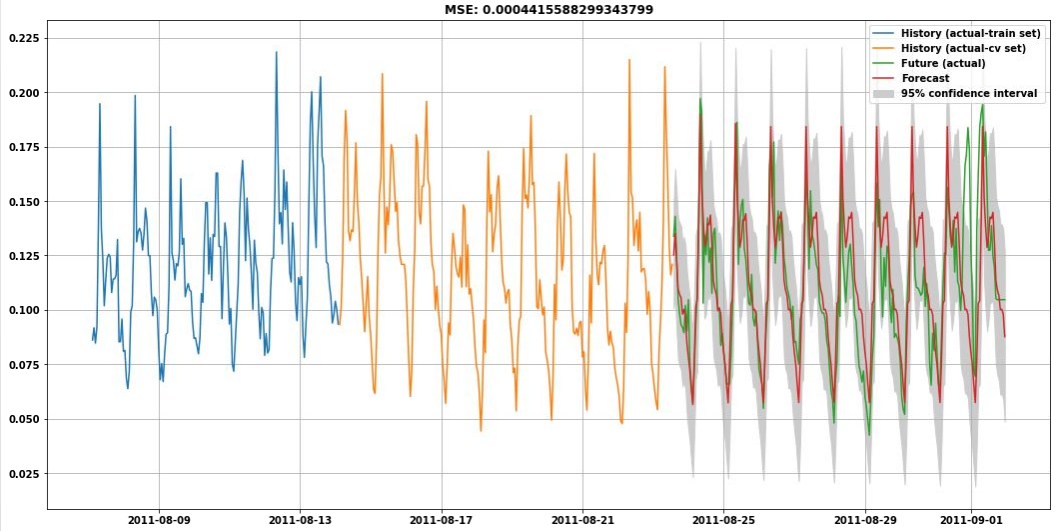
In [16]:
```
f_steps = test_ts.shape[0]
results = model_fit.get_forecast(f_steps)

forecasts = pd.concat([results.predicted_mean, results.conf_int(alpha=0.05)], axis=1)
forecasts.columns = ['Forecasts', 'Lower 95% CI', 'Upper 95% CI']

forecasts.head()
```

Out[16]:

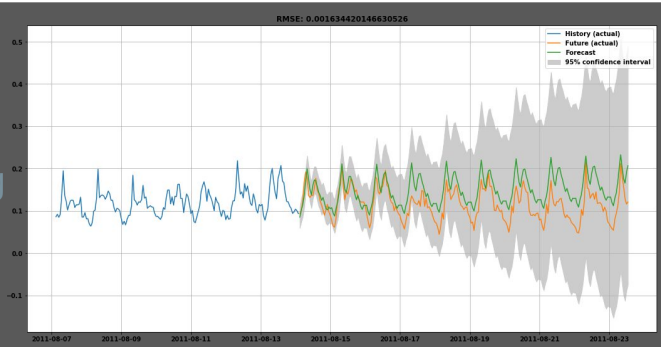
	Forecasts	Lower 95% CI	Upper 95% CI
2011-08-23 14:00:00	0.125252	0.100648	0.149857
2011-08-23 15:00:00	0.135048	0.105400	0.164696
2011-08-23 16:00:00	0.121904	0.090240	0.153568
2011-08-23 17:00:00	0.109568	0.077033	0.142103
2011-08-23 18:00:00	0.107122	0.074202	0.140043

Hyperparameter Optimization: SARIMAX GridSearch

- Custom GridSearch Module
  - Accuracy Metric
  - Test Function
  - Parameter Feeder/Wrapper
  - GridSearch Object
- Tuning
  - Parameter Search
  - Selection & Training
- Forecasting
- **Evaluation**

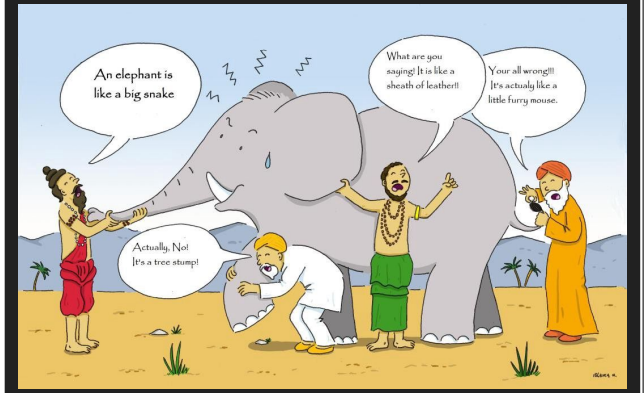


Pre-Tuning

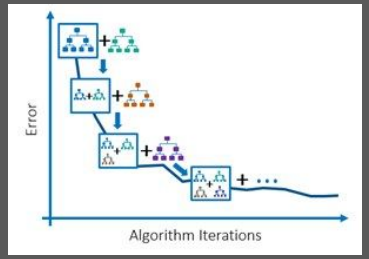


Ensembling

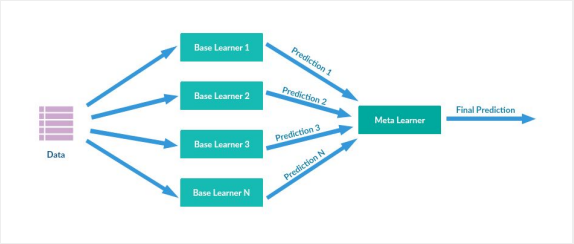
Bagging



Boosting



Stacking



Ensemble Model: Random Forest

- Pre-Processing
- **Training**
- Forecasting
- Validation

In [31]:
```
from sklearn.ensemble import RandomForestRegressor
import time

rfr = RandomForestRegressor(max_depth=3, n_estimators=10)
start = time.time()
rfr = rfr.fit(X, y)
print('fitting complete after {} seconds'.format(time.time()-start))
```

```
fitting complete after 0.01999807357788086 seconds
```

Ensemble Model: Random Forest

- Pre-Processing
- Training
- **Forecasting**
- Validation

In [32]:
```
from datetime import timedelta

forecasts = pd.DataFrame(columns=train_ts_processed.columns, index=val_ts.index)
attached_df = pd.concat([train_ts_processed[-48:], forecasts], axis=0)
for t in forecasts.index:
    for l in [1,2,3,24,48]:
        forecasts.loc[t, 'lag-{}'.format(l)] = attached_df.loc[t - timedelta(hours=l), 'cpc']
    step_X = forecasts.loc[t].values[1:].reshape(1,-1)
    step_y = rfr.predict(step_X)[0]
    attached_df.loc[t, 'cpc'], forecasts.loc[t, 'cpc'] = step_y, step_y

forecasts.head()
```

Out[32]:

timestamp	cpc	lag-1	lag-2	lag-3	lag-24	lag-48
2011-08-14 03:00:00	-0.00380121	-0.00665327	-0.00387172	0.00650718	-0.010359	0.00154876
2011-08-14 04:00:00	0.012166	-0.00380121	-0.00665327	-0.00387172	0.0144919	0.0268069
2011-08-14 05:00:00	0.00931566	0.012166	-0.00380121	-0.00665327	0.0126438	0.0150717
2011-08-14 06:00:00	0.0110425	0.00931566	0.012166	-0.00380121	0.0417596	0.000228741
2011-08-14 07:00:00	0.0313097	0.0110425	0.00931566	0.012166	0.0379464	0.0308431

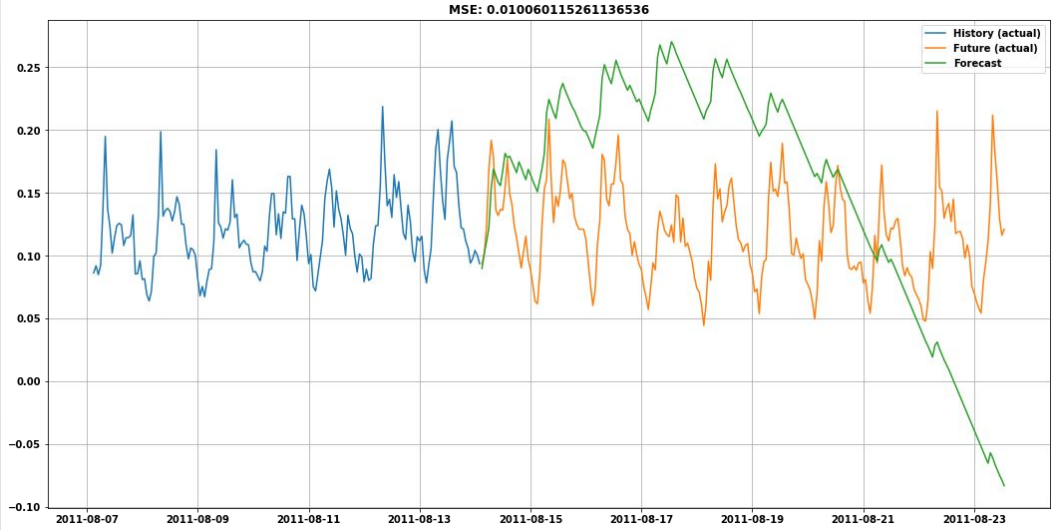
Note that the forecasts need to be re-trended

In [33]:
```
forecasts['cpc'][0] = forecasts['cpc'][0] + train_ts[-1]
forecasts['cpc'] = forecasts['cpc'].cumsum()
forecasts['cpc'].head()
```

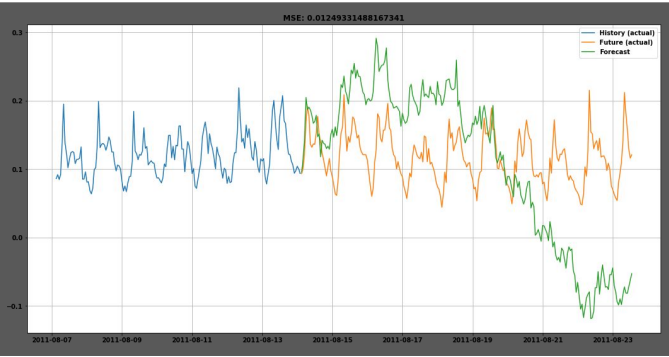
Out[33]:
```
timestamp
2011-08-14 03:00:00    0.0896484
2011-08-14 04:00:00     0.101814
2011-08-14 05:00:00      0.11113
2011-08-14 06:00:00     0.122172
2011-08-14 07:00:00     0.153482
Freq: H, Name: cpc, dtype: object
```

Ensemble Model: Random Forest

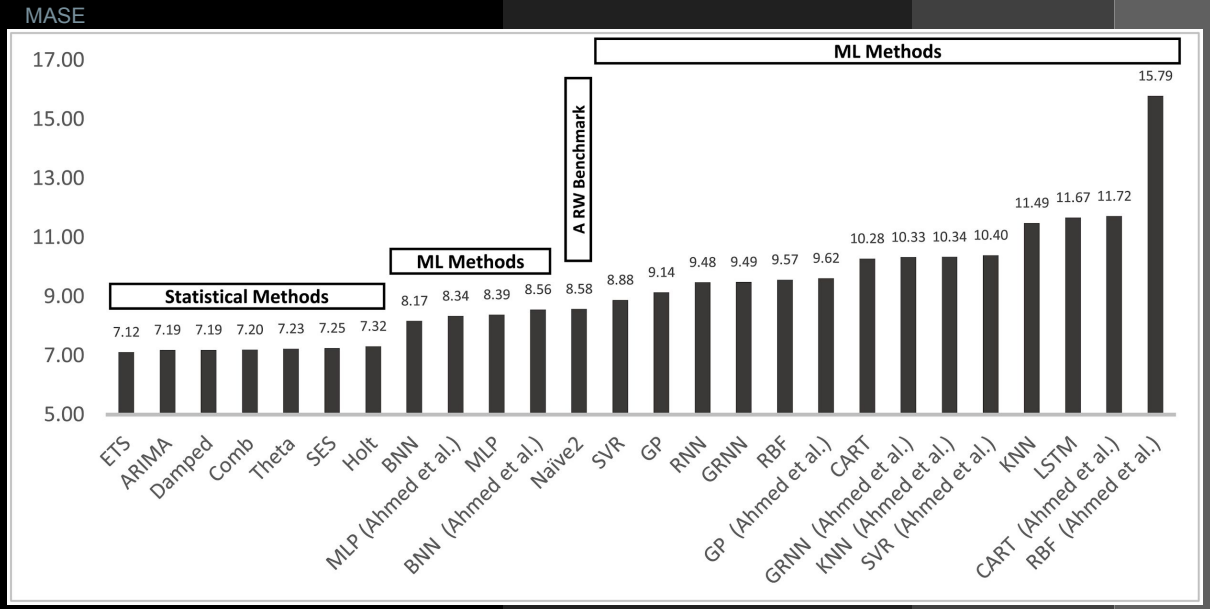
- Pre-Processing
- Training
- Forecasting
- **Validation**



Decision Tree



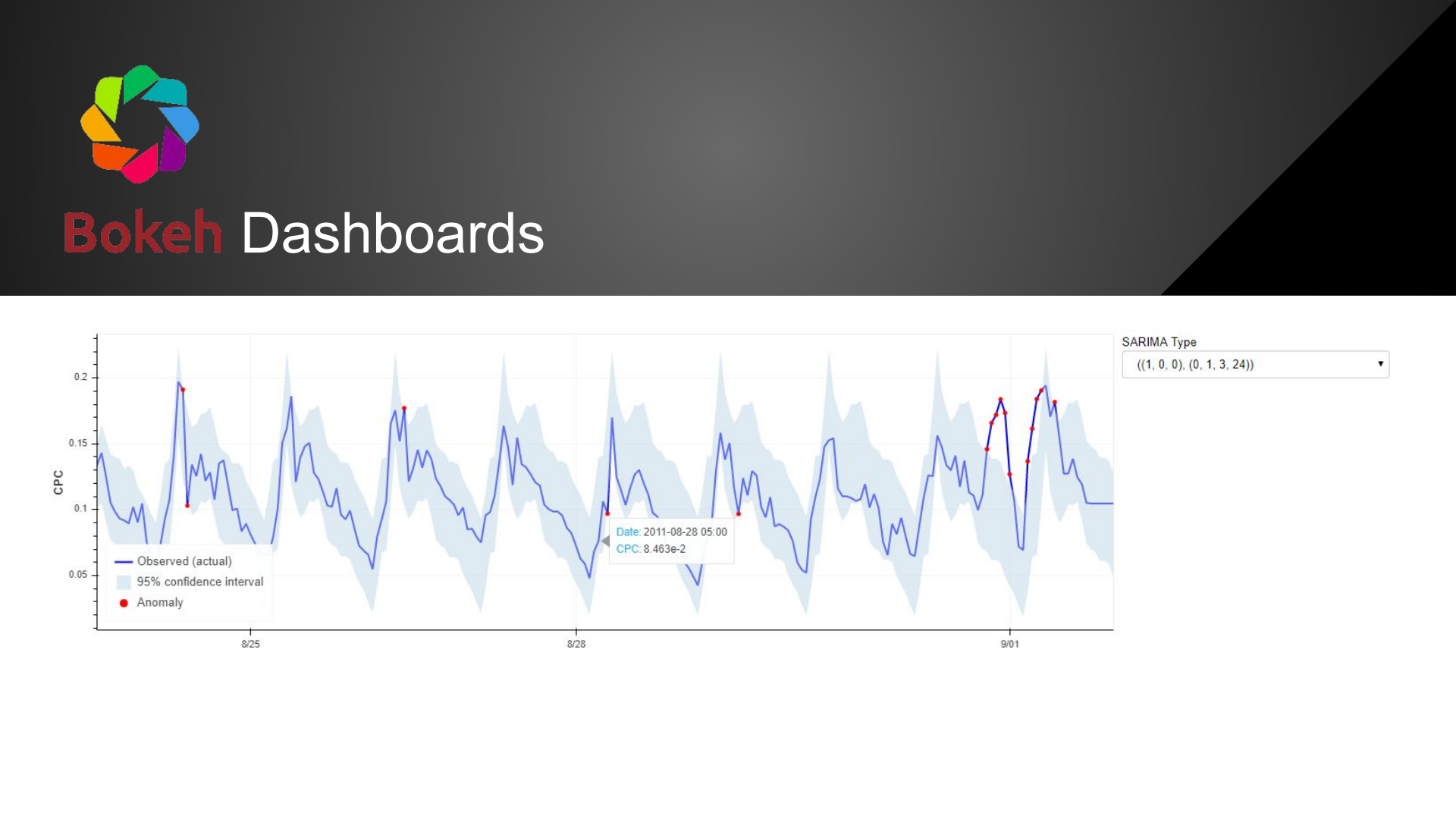
Makridakis Competitions (M3 Results)



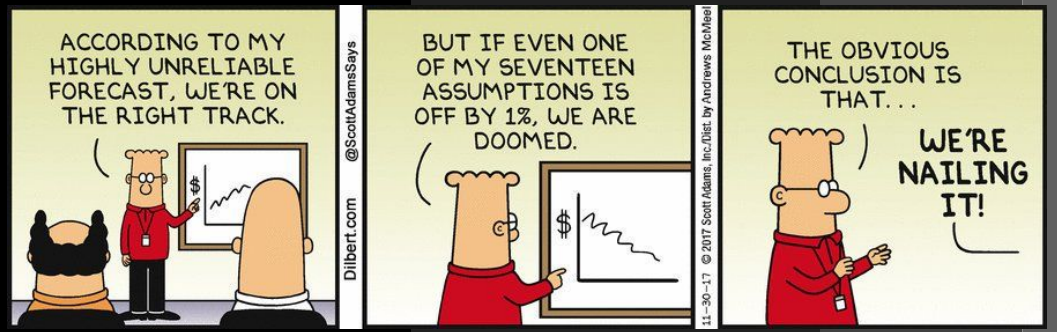
Source: Makridakis, Spyros, Evangelos Spiliotis, and Vassilios Assimakopoulos. "Statistical and Machine Learning forecasting methods: Concerns and ways forward." PloS one 13.3 (2018): e0194889.

Python Visualization Tools

PyViz Bokeh


Bokeh Dashboards
SARIMA Type
((1, 0, 0), (0, 1, 3, 24))
CPC
0.2
0.15
0.1
0.05
Observed (actual)
95% confidence interval
Anomaly
Date: 2011-08-28 05:00
CPC: 8.463e-2
8/25
8/28
9/01

Thank You!



Questions, Feedback, Comments:

AhmedR.Abdulaal@gmail.com

Slides & Notebook:

https://github.com/aabdulaal/Practical_TS_Talk