Forecasting time-series data

- Intent of this page
- About the data
    - Plotting raw-data
    - Checking for Stability of Time-series data
        - Results
- Building a forecasting model using ARIMA
    - Results
    - CPU Forecast for 1-day ahead (24-values)
- Forecasting time-series in Wavefront/TO
    - Holt-winters
        - Description
        - Usage
        - FAQs about hw()
        - Reference
    - nnforecast - Forecast using Neural Networks (MLPs)
        - Description
        - Usage
        - FAQs about nnforecast
        - Reference
    - linearforecast - Forecast using linear regression
        - Description
        - Usage
        - Reference
    - Wavefront dashboard/charts
        - Link to dashboard

Intent of this page

To capture forecasting techniques of a time-series using Auto Regressive Integrated Moving Average (ARIMA) and compare it with Holt-Winters (HW) - a model adopted by Wavefront/TO for forecasting. The objective is to eventually show all the work that happens under the hood with HW in WF to solve the following use-cases:

- Predict future trends/values from a time-series to plan say things like resources/capacity (like vertical/horizontal scaling etc.) OR see un-foreseen software defects like say memory leaks etc.
- Reporting anomalies/alerts if current values deviate from predicted values

About the data

For the purpose of this discussion, time-series data of cpu consumption in milli-cores of a default namespace in cluster configured on AWS via TMC will be used. It comprises 144x rows or data from ~6 days at 1hr periodicity. The data is present in k8s dashboard and is retrieved using the wf-cli (an exponent of the WF-API framework):

wf-cli to obtain data from existing WF dashboard

```
$ wf query 'sum(ts("kubernetes.ns.cpu.usage_rate", cluster="tmc-k8s-aws" AND namespace_name="default"), namespace_name, cluster)' -s -144h -e -120h -g h -f csv -F headers > k8s.ns.cpu.usage_hr.csv

$ wf query 'sum(ts("kubernetes.ns.cpu.usage_rate", cluster="tmc-k8s-aws" AND namespace_name="default"), namespace_name, cluster)' -s -120h -e -96h -g h -f csv >> k8s.ns.cpu.usage_hr.csv

.

.
```

Data saved to a .csv file is retrieved and loaded into a *Pandas-DataFrame* as shown below:

```
# Load a sample time-series from csv
import pandas as pd
import datetime
import matplotlib.pyplot as plt

# timestamp in epoch - Transform to datetime format and
# use is it as index
dateparse = lambda epoch_time: datetime.datetime.fromtimestamp(int(epoch_time))

df = pd.read_csv('/Users/ssharat/Downloads/k8s.ns.cpu.usage_hr.csv', sep=',', parse_dates=['timestamp'],
                 index_col='timestamp', date_parser=dateparse)
print(df.shape)
df.head()
```

```
(144, 3)
```

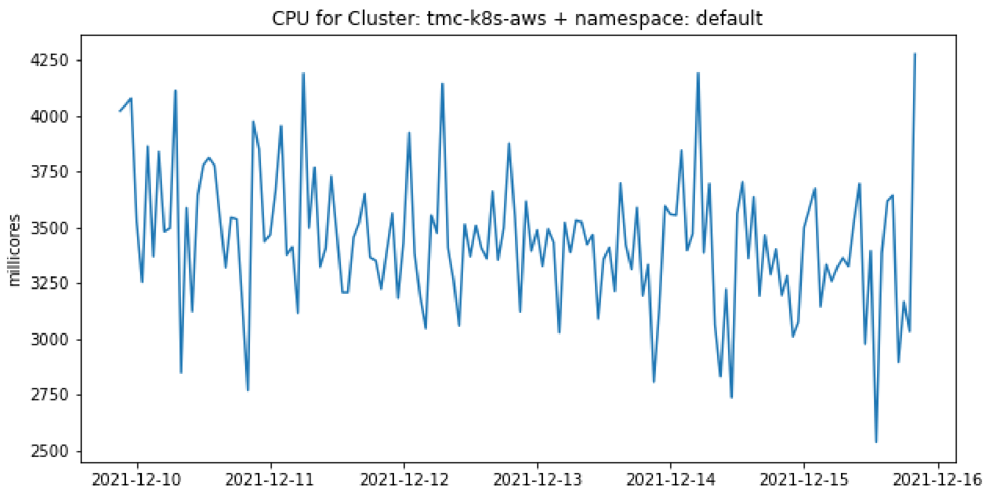
Plotting raw-data

Using matplotlib to plot the raw data obtained from the chart on the WF dashboard

Code:

Code to plot time-series

```
# Change size of plot/grid
f = plt.figure()
f.set_figwidth(10)
f.set_figheight(5)

# Plot the CPU prior any time-series analysis
plt.title('CPU for Cluster: tmc-k8s-aws + namespace: default')
plt.ylabel('millicores')

plt.plot(df.value)
```



Checking for Stability of Time-series data

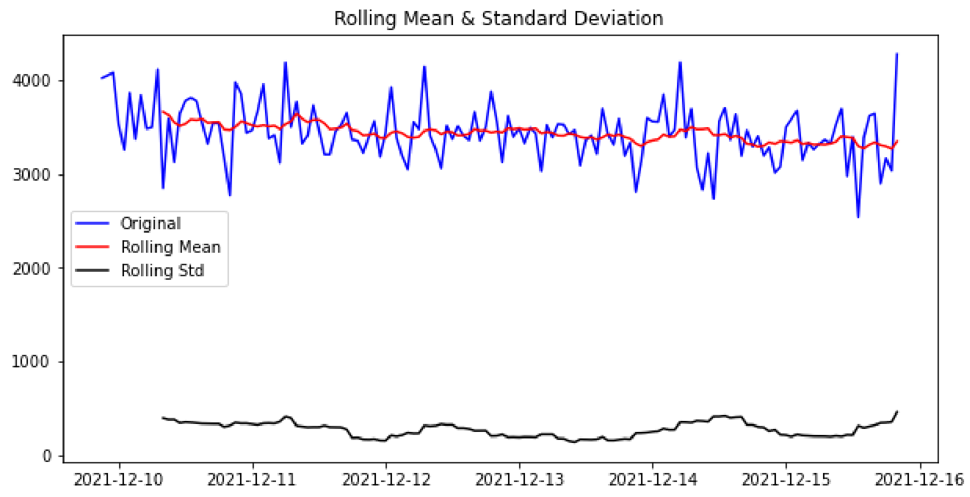
Prior to forecasting time-series data (using ARIMA), it is essential we check for stability and find ways to normalize the data if we cannot dis-prove the NULL Hypothesis (NULL Hypothesis: A given time-series data is not stable). For this we will use the Dickey-Fuller Test.

Code:

Dickey-Fuller stability test

```
# Checking for stability of a TS using Dickey-Fuller test
from statsmodels.tsa.stattools import adfuller
def test_stationarity(timeseries):
    #Determining rolling statistics
    window = 12
    rolmean = timeseries.rolling(window).mean()
    rolstd = timeseries.rolling(window).std()

    # Change size of plot/grid
    f = plt.figure()
    f.set_figwidth(10)
    f.set_figheight(5)

    #Plot rolling statistics:
    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(block=False)

    #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)

test_stationarity(ts)
```

Below is a plot of the rolling Mean and Standard-deviation alongside the time-series (ts):



Results

```
Results of Dickey-Fuller Test:
Test Statistic                 -7.922845e+00
p-value                         3.684683e-12
#Lags Used                      2.000000e+00
Number of Observations Used     1.410000e+02
Critical Value (1%)            -3.477601e+00
Critical Value (5%)            -2.882266e+00
Critical Value (10%)           -2.577822e+00
```

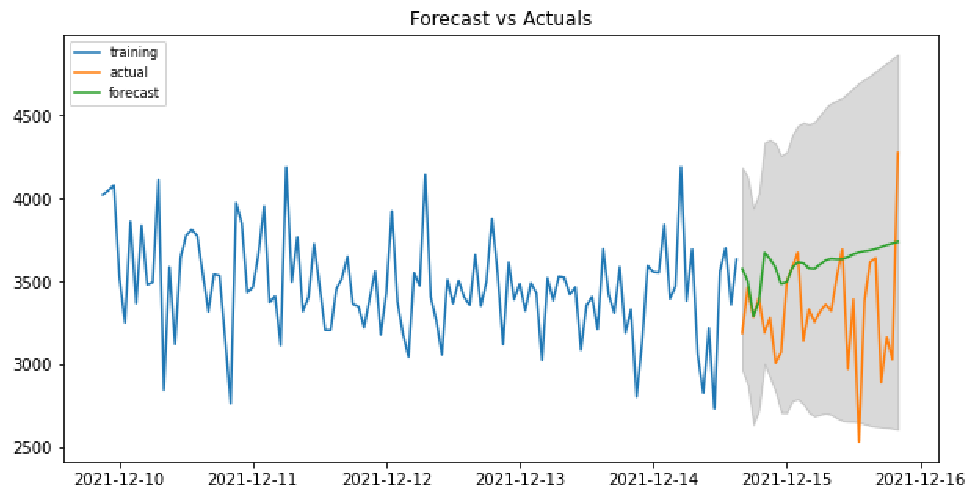
Interpreting the Results:

- Test-statistics -7.9 is << 1/5/10% critical values of -3.47, -2.88 and -2.57

        - - p-value 3.68 x 10^-12 is << 0.05
    - Conclusion:
      Reject the null hypothesis and we can consider the time-series as stable

Building a forecasting model using ARIMA

Using an 80/20 split of the data, following is a plot showing predictions made using the ARIMA model (once we have determined the required p & q from Auto-correlation and Partial Auto-correlation functions)



To better interpret and corroborate predictions with metrics

Accuracy metrics

```
# Accuracy metrics
def forecast_accuracy(forecast, actual):
    mape = np.mean(np.abs(forecast - actual)/np.abs(actual)) # MAPE
    me = np.mean(forecast - actual) # ME
    mae = np.mean(np.abs(forecast - actual)) # MAE
    mpe = np.mean((forecast - actual)/actual) # MPE
    rmse = np.mean((forecast - actual)**2)**.5 # RMSE
    corr = np.corrcoef(forecast, actual)[0,1] # corr
    mins = np.amin(np.hstack([forecast[:,None],
    actual[:,None]]), axis=1)
    maxs = np.amax(np.hstack([forecast[:,None],
    actual[:,None]]), axis=1)
    minmax = 1 - np.mean(mins/maxs) # minmax
    acf1 = acf(fc-test)[1] # ACF1
    return({'mape':mape, 'me':me, 'mae': mae,
            'mpe': mpe, 'rmse':rmse, 'acf1':acf1,
            'corr':corr, 'minmax':minmax})

forecast_accuracy(fc, test.values)
```

Results

```
{'mape': 0.1053728070565144,
 'me': 281.2121152095004,
 'mae': 327.8294346605773,
 'mpe': 0.09408132411118759,
 'rmse': 428.53292548724943,
 'acf1': -0.009959833215436 21,
 'corr': 0.03275715464342374,
 'minmax': 0.08916236145228784}
```

Interpreting the Results:

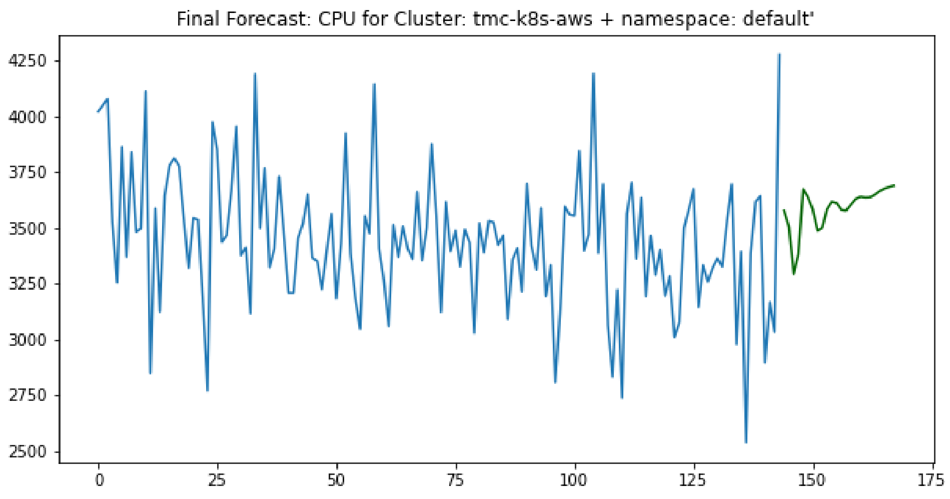
With the Mean Absolute Precision Error (MAPE) at ~10.5% for ARIMA order (5,2,1) the model yields a precision of 89.5%

CPU Forecast for 1-day ahead (24-values)

Using the code excerpt below, the model forecast function is plotted as seen in the chart (line in green is Forecast). Essentially, this is a prediction of CPU milli-cores for NS = default on cluster tmc-k8s-aws in range (3500, 3750).

Forecast using ARIMA

```
# Forecast
n_periods = 24
fc = results_AR.forecast(steps=n_periods)[0]

# Change size of plot/grid
f = plt.figure()
f.set_figwidth(10)
f.set_figheight(5)

# Plot
df_basic = df.reset_index(drop=True)
plt.plot(df_basic.value)

index_of_fc = np.arange(len(df_basic.value), len(df_basic.value) + n_periods)
# make series for plotting purpose
fc_series = pd.Series(fc, index=index_of_fc)

plt.plot(fc_series, color='darkgreen')

plt.title("Final Forecast: CPU for Cluster: tmc-k8s-aws + namespace: default'")
plt.show()
```



Forecasting time-series in Wavefront/TO

In the previous section, we show forecasting using ARIMA (by hand) and see it is a daunting task of having to go through several hoops of:

- Storing historic data and accessing it using api
- Checking for stability of time-series using the DF method (normalize the data, to achieve stability)
- Build the model and make a prediction

In order to keep the model up to date, we ought to repeat this process by setting up a pipeline.

Wavefront/TO does all of this under the hood (the platform stores time-series data for 18M without any roll-ups) and offers prediction/forecast through functions like hw(), nnforecast() and linearforecast(). For methods like hw() and nnforecast() the models are pre-trained

Holt-winters

Description

Forecasting can be largely classified as judgmental, univariate or bivariate. HW is univariate forecasting method. HW uses simple exponential smoothing in order to forecast. The forecast is obtained as a weighted avg. of the past observed values, where the weights decline exponentially so that the values of recent observations contribute to forecast. Further, HW accounts for trend and irregularities in data allowing seasonality to mostly exist.

NOTE:

ARIMA due to its stringent requirements of stability would normalize data to an extent that seasonality is mostly flattened.

Usage

hw(30m, ${etcd_cpu}, 0.5, 0.5)

where the 3rd and 4th args are Smoothing factor and Trend factor respectively.

There is choice between double/triple exponential smoothing based on if there is seasonality in data.

FAQs about hw()

There is no real owner for this function and the usage is not clear. Here are some open questions/comments:

How to determine if data is seasonal?

For Triple Exponential Smoothing the pre-requisite seems to be seasonal data. Seasonality sometimes may not be apparent and so, I was thinking if there was a function to determine if data was seasonal or better even if hw() itself could determine that implicitly and abstract the choice between Double & Triple Exponential smoothing.

Perhaps, some of the documentation can be worded simpler

For instance, under *History Length* in hw(), what is meant by left side of the window?

Reference

https://docs.wavefront.com/ts_hw.html

nnforecast - Forecast using Neural Networks (MLPs)

Description

nnforecast uses a Neural Network based on MLPs that is trained offline in a private GPU-accelerated environment. The model is then ported to the Wavefront cloud environment to be applied on customer time-series data.

Usage

nnforecast(10d, 0.9, <time-series data>, with_bounds) where :

- 10d is the forecast period
- 0.9 is the confidence
- with_bounds would provide lower and upper bound limit

NOTE:

1. To view the prediction ensure the end-date is adjusted to current time + forecast period (drag x-axis/time axis on chart to the right).
    - In the example above, that would be current time + 10d
2. The MLP or neural net is pre-trained with 40 nodes in the input layer to 20 nodes in the output layer.
    - Essentially, that means if we want to get a prediction for 10d the time-series should have at least 20d of prior data
3. nnforecast is also used in Wavefront's *Anomaly Detection* function

FAQs about nnforecast

If the model is built off-line on data/TS not related to customer TS, how can we give customers confidence about forecast (Example: If the NN/model is generic can it cater to erratic nature of say a CPU chart?)

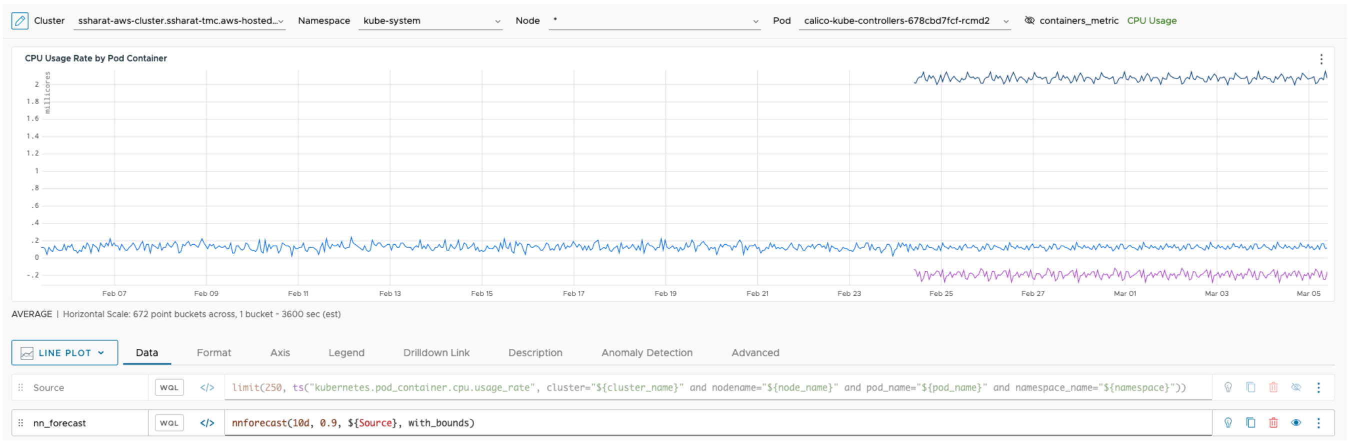
The entire forecasting engine consists of two parts. The first part is applied directly to the customer specific time series. It performs hypothesis testing procedures for trend and periodicity (not implemented) detection. Also it verifies some common classes like piecewise-trendy, piecewise-constant, almost constant, etc. that don't need any NN model. Based on the revealed information, the engine transforms data into a stationary process (without trend, periodicity, etc.). In the second part, the engine applies the NN model trained specifically for a stationary process. In this case, moderate number of time series should be sufficient for training. We see that the first part of engine is responsible for understanding the specific behavior of a customer time series.

What is current input data to train the model? Is it still coming from Mon, 3k time-series, 1-minute monitoring interval?

- Also, how often is the model trained on new data and uploaded to Wavefront customer tenant?

We didn't update the model since its release. Not sure that it is necessary due to the explanations in the previous step. NN model captures only the stationary behavior of time series, actually independently of the monitoring interval.

The with_bounds show a min and max bounds - What does it mean to have negative index for things like CPU? How can we explain the chart below:


Cluster ssharat-aws-cluster.ssharat-tmc.aws-hosted... Namespace kube-system Node * Pod calico-kube-controllers-678cbd7fcf-rcmd2 containers_metric CPU Usage

CPU Usage Rate by Pod Container

AVERAGE | Horizontal Scale: 672 point buckets across, 1 bucket - 3600 sec (est)

Source
```
limit(250, ts("kubernetes.pod_container.cpu.usage_rate", cluster="${cluster_name}" and nodename="${node_name}" and pod_name="${pod_name}" and namespace_name="${namespace}"))
```

nn_forecast
```
nnforecast(10d, 0.9, ${Source}, with_bounds)
```

Ideally, if the corresponding confidence bound is upper or lower than the known historical range of data, we need to shift them accordingly. However, for simplicity, we decided not to carry the corresponding computations. Hence, the conclusion is to ignore such cases as future data points will never violate the corresponding bounds and will never trigger alerts.

I am looking to demo forecast to our customers. Are there any sample/test datasets I could use to show value of forecasting accurately?

I know that the Wavefront-Armenia team had some standard examples for testing. Please check if they are still available. Also, worth mentioning that the Wavefront UI is showing the results after some summarizing, so all beautiful examples from our python notebooks are simply not useful for the model validation in real customer environments. The best approach should be selection of such examples directly from customer environments. Also due to summarizing process, the confidence bounds sometimes seem very strange, as they are not corresponding to the time series that you see on the UI.

Reference

Time Series Forecasting With Deep Learning - Trend Analysis

linearforecast - Forecast using linear regression

Description

Predicts/forecasts values of a time-series using Simple Linear Regression

Usage

linearforecast(9d, 4w, <time-series data>) where:

- 9d is the forecast period
- 4w is the prior data to consider

Reference

https://docs.wavefront.com/ts_linearforecast.html

Wavefront dashboard/charts

Finally, all this comes together via a dashboard. The dashboard has the following sections:

- Section-1: Query stats

Often, it is important to know query times when running a forecasting/math function over complex data (time-series cardinality, length of time etc). Using the *wf-sdk* and an iterator we walk dashboards charts queries: grab the ts queries, run them and extract query run time.

See code excerpt of how to do this (if interested):

Code excerpt about iterating dashboards/charts/ts queries

```
for section in get_dashboard.response.sections:
    for row in section.rows:
        for chart in row.charts:
            # Create a db with chart vars and ts queries
            ts_db = {}
            for src in chart.sources:
                api_response = None
                now = datetime.now()
                epoch_msec = convert_time_to_epoch_msec(now.strftime("%Y-%m-%d %H:%M:%S"))
                # Sequence assumes first query is always ts() - not scoping for
                # derived metrics, in which case first query is a variable
                if re.search('ts', src.query):
                    print('Making query: %s %s %s' % (src.query, epoch_msec - HOUR_MSEC, 'm'))
                    api_response = query_api.query_api(src.query, epoch_msec - HOUR_MSEC, 'm')
                    ts_db[src.name] = src.query
                else:
                    # Replace the var with ts()
                    for var in ts_db:
                        if re.search('\\${%s}' % var, src.query):
                            src_query = src.query.replace('${%s}' % var, ts_db[var])
                            print('Making query: %s %s %s' % (src_query, epoch_msec - HOUR_MSEC, 'm'))
                            api_response = query_api.query_api(src_query, epoch_msec - HOUR_MSEC, 'm')
                            ts_db[src.name] = src_query
                            break
                if api_response is not None:
                    # Send metric
                    print('\t Send metric for chart: %s, query: %s, value: %s to wavefront' % (chart.name,
src.query,
                                                                                                    api_response.
stats.latency))
                    wf_client.send_metric(name='my.query.stats.latency', value=api_response.stats.latency,
                                          timestamp=epoch_msec, tags={'query': src.query, 'chart': chart.
name})
```

- Section-2: Prediction

Captures forecasting functions like linearforecast() (simple linear regression), nnforecast() and hw()

- Section-3: Anomaly Detection (AD) using math

Captures using functions like anomalous, mavg, mmedian and variations to study spikes, anomalies about data

Link to dashboard

https://longboard.wavefront.com/u/fkrYpgRhKK?t=acme-ssharat