

Automated Machine Learning

Forecasting away from training data

Contents

1. [Introduction](#)
2. [Setup](#)
3. [Data](#)
4. [Prepare remote compute and data.](#)
5. [Create the configuration and train a forecaster](#)
6. [Forecasting from the trained model](#)
7. [Forecasting away from training data](#)

Introduction

This notebook demonstrates the full interface of the `forecast()` function.


The best known and most frequent usage of `forecast` enables forecasting on test sets that immediately follows training data.

However, in many use cases it is necessary to continue using the model for some time before retraining it. This happens especially in **high frequency forecasting** when forecasts need to be made more frequently than the model can be retrained. Examples are in Internet of Things and predictive cloud resource scaling.

Here we show how to use the `forecast()` function when a time gap exists between training data and prediction period.

Terminology:

- forecast origin: the last period when the target value is known
- forecast periods(s): the period(s) for which the value of the target is desired.
- lookback: how many past periods (before forecast origin) the model function depends on. The larger of number of lags and length of rolling window.
- prediction context: `lookback` periods immediately preceding the forecast origin

 Impressions

Setup

Please make sure you have followed the `configuration.ipynb` notebook so that your ML workspace information is saved in the config file.

```
In [ ]: import os
import pandas as pd
import numpy as np
import logging
import warnings

import azureml.core
from azureml.core.dataset import Dataset
from pandas.tseries.frequencies import to_offset
from azureml.core.compute import AmlCompute
from azureml.core.compute import ComputeTarget
from azureml.core.runconfig import RunConfiguration
from azureml.core.conda_dependencies import CondaDependencies

# Squash warning messages for cleaner output in the notebook
warnings.showwarning = lambda *args, **kwargs: None

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

This sample notebook may use features that are not available in previous versions of the Azure ML SDK.

```
In [ ]: print("This notebook was created using version 1.31.0 of the Azure ML SDK")
print("You are currently using version", azureml.core.VERSION, "of the Azure ML SDK")
```

```
In [ ]: from azureml.core.workspace import Workspace
from azureml.core.experiment import Experiment
from azureml.train.automl import AutoMLConfig

ws = Workspace.from_config()

# choose a name for the run history container in the workspace
experiment_name = 'automl-forecast-function-demo'

experiment = Experiment(ws, experiment_name)

output = {}
output['Subscription ID'] = ws.subscription_id
output['Workspace'] = ws.name
output['SKU'] = ws.sku
output['Resource Group'] = ws.resource_group
output['Location'] = ws.location
output['Run History Name'] = experiment_name
pd.set_option('display.max_colwidth', -1)
outputDf = pd.DataFrame(data = output, index = [''])
outputDf.T
```

Data

For the demonstration purposes we will generate the data artificially and use them for the forecasting.

```
In [ ]: TIME_COLUMN_NAME = 'date'
TIME_SERIES_ID_COLUMN_NAME = 'time_series_id'
TARGET_COLUMN_NAME = 'y'

def get_timeseries(train_len: int,
                  test_len: int,
                  time_column_name: str,
                  target_column_name: str,
                  time_series_id_column_name: str,
                  time_series_number: int = 1,
                  freq: str = 'H'):
    """
    Return the time series of designed length.

    :param train_len: The length of training data (one series).
    :type train_len: int
    :param test_len: The length of testing data (one series).
    :type test_len: int
    :param time_column_name: The desired name of a time column.
    :type time_column_name: str
    :param time_series_number: The number of time series in the data set.
    :type time_series_number: int
    :param freq: The frequency string representing pandas offset.
                  see https://pandas.pydata.org/pandas-docs/stable/user\_guide/timese
    :type freq: str
    :returns: the tuple of train and test data sets.
    :rtype: tuple

    """
    data_train = [] # type: List[pd.DataFrame]
    data_test = [] # type: List[pd.DataFrame]
    data_length = train_len + test_len
    for i in range(time_series_number):
        X = pd.DataFrame({
            time_column_name: pd.date_range(start='2000-01-01',
                                           periods=data_length,
                                           freq=freq),
            target_column_name: np.arange(data_length).astype(float) * np.random.ra
            'ext_predictor': np.asarray(range(42, 42 + data_length)),
            time_series_id_column_name: np.repeat('ts{}'.format(i), data_length)
        })
        data_train.append(X[:train_len])
        data_test.append(X[train_len:])
    X_train = pd.concat(data_train)
    y_train = X_train.pop(target_column_name).values
    X_test = pd.concat(data_test)
    y_test = X_test.pop(target_column_name).values
    return X_train, y_train, X_test, y_test

n_test_periods = 6
n_train_periods = 30
X_train, y_train, X_test, y_test = get_timeseries(train_len=n_train_periods,
```

```
test_len=n_test_periods,
time_column_name=TIME_COLUMN_NAME
target_column_name=TARGET_COLUMN_
time_series_id_column_name=TIME_S
time_series_number=2)
```

Let's see what the training data looks like.

```
In [ ]: X_train.tail()
```

```
In [ ]: # plot the example time series
import matplotlib.pyplot as plt
whole_data = X_train.copy()
target_label = 'y'
whole_data[target_label] = y_train
for g in whole_data.groupby('time_series_id'):
    plt.plot(g[1]['date'].values, g[1]['y'].values, label=g[0])
plt.legend()
plt.show()
```

Prepare remote compute and data.

The [Machine Learning service workspace](#), is paired with the storage account, which contains the default data store. We will use it to upload the artificial data and create [tabular dataset](#) for training. A tabular dataset defines a series of lazily-evaluated, immutable operations to load data from the data source into tabular representation.

```
In [ ]: # We need to save thw artificial data and then upload them to default workspace dat
DATA_PATH = "fc_fn_data"
DATA_PATH_X = "{}data_train.csv".format(DATA_PATH)
if not os.path.isdir('data'):
    os.mkdir('data')
pd.DataFrame(whole_data).to_csv("datadata_train.csv", index=False)
# Upload saved data to the default data store.
ds = ws.get_default_datastore()
ds.upload(src_dir='./data', target_path=DATA_PATH, overwrite=True, show_progress=Tr
train_data = Dataset.Tabular.from_delimited_files(path=ds.path(DATA_PATH_X))
```

You will need to create a [compute target](#) for your AutoML run. In this tutorial, you create AmlCompute as your training compute resource.

Note that if you have an AzureML Data Scientist role, you will not have permission to create compute resources. Talk to your workspace or IT admin to create the compute targets described in this section, if they do not already exist.

```
In [ ]: from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# Choose a name for your CPU cluster
```

```
amlcompute_cluster_name = "fcfn-cluster"

# Verify that cluster does not exist already
try:
    compute_target = ComputeTarget(workspace=ws, name=amlcompute_cluster_name)
    print('Found existing cluster, use it.')
except ComputeTargetException:
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_D2_V2',
                                                         max_nodes=6)

    compute_target = ComputeTarget.create(ws, amlcompute_cluster_name, compute_conf

compute_target.wait_for_completion(show_output=True)
```

Create the configuration and train a forecaster

First generate the configuration, in which we:

- Set metadata columns: target, time column and time-series id column names.
- Validate our data using cross validation with rolling window method.
- Set normalized root mean squared error as a metric to select the best model.
- Set early termination to True, so the iterations through the models will stop when no improvements in accuracy score will be made.
- Set limitations on the length of experiment run to 15 minutes.
- Finally, we set the task to be forecasting.
- We apply the lag lead operator to the target value i.e. we use the previous values as a predictor for the future ones.
- [Optional] Forecast frequency parameter (freq) represents the period with which the forecast is desired, for example, daily, weekly, yearly, etc. Use this parameter for the correction of time series containing irregular data points or for padding of short time series. The frequency needs to be a pandas offset alias. Please refer to [pandas documentation](#) for more information.

```
In [ ]: from azureml.core.forecasting_parameters import ForecastingParameters
lags = [1,2,3]
forecast_horizon = n_test_periods
forecasting_parameters = ForecastingParameters(
    time_column_name=TIME_COLUMN_NAME,
    forecast_horizon=forecast_horizon,
    time_series_id_column_names=[ TIME_SERIES_ID_COLUMN_NAME ],
    target_lags=lags,
    freq='H' # Set the forecast frequency to be hourly
)
```

Run the model selection and training process. Validation errors and current status will be shown when setting `show_output=True` and the execution will be synchronous.

```
In [ ]: from azureml.core.workspace import Workspace
from azureml.core.experiment import Experiment
from azureml.train.automl import AutoMLConfig
```

```

automl_config = AutoMLConfig(task='forecasting',
                             debug_log='automl_forecasting_function.log',
                             primary_metric='normalized_root_mean_squared_error',
                             experiment_timeout_hours=0.25,
                             enable_early_stopping=True,
                             training_data=train_data,
                             compute_target=compute_target,
                             n_cross_validations=3,
                             verbosity = logging.INFO,
                             max_concurrent_iterations=4,
                             max_cores_per_iteration=-1,
                             label_column_name=target_label,
                             forecasting_parameters=forecasting_parameters)

remote_run = experiment.submit(automl_config, show_output=False)

```

```
In [ ]: remote_run.wait_for_completion()
```

```
In [ ]: # Retrieve the best model to use it further.
_, fitted_model = remote_run.get_output()
```

Forecasting from the trained model

In this section we will review the `forecast` interface for two main scenarios: forecasting right after the training data, and the more complex interface for forecasting when there is a gap (in the time sense) between training and testing data.

X_train is directly followed by the X_test

Let's first consider the case when the prediction period immediately follows the training data. This is typical in scenarios where we have the time to retrain the model every time we wish to forecast. Forecasts that are made on daily and slower cadence typically fall into this category. Retraining the model every time benefits the accuracy because the most recent data is often the most informative.



Forecasting after training

We use `X_test` as a **forecast request** to generate the predictions.

Typical path: X_test is known, forecast all upcoming periods

```
In [ ]: # The data set contains hourly data, the training set ends at 01/02/2000 at 05:00

# These are predictions we are asking the model to make (does not contain the target)
# for 6 periods beginning with 2000-01-02 06:00, which immediately follows the training set
X_test
```

```
In [ ]: y_pred_no_gap, xy_nogap = fitted_model.forecast(X_test)

# xy_nogap contains the predictions in the _automl_target_col column.
# Those same numbers are output in y_pred_no_gap
xy_nogap
```

Confidence intervals

Forecasting model may be used for the prediction of forecasting intervals by running `forecast_quantiles()`. This method accepts the same parameters as `forecast()`.

```
In [ ]: quantiles = fitted_model.forecast_quantiles(X_test)
quantiles
```

Distribution forecasts

Often the figure of interest is not just the point prediction, but the prediction at some quantile of the distribution. This arises when the forecast is used to control some kind of inventory, for example of grocery items or virtual machines for a cloud service. In such case, the control point is usually something like "we want the item to be in stock and not run out 99% of the time". This is called a "service level". Here is how you get quantile forecasts.

```
In [ ]: # specify which quantiles you would like
fitted_model.quantiles = [0.01, 0.5, 0.95]
# use forecast_quantiles function, not the forecast() one
y_pred_quantiles = fitted_model.forecast_quantiles(X_test)

# quantile forecasts returned in a Dataframe along with the time and time series id
y_pred_quantiles
```

Destination-date forecast: "just do something"

In some scenarios, the `X_test` is not known. The forecast is likely to be weak, because it is missing contemporaneous predictors, which we will need to impute. If you still wish to predict forward under the assumption that the last known values will be carried forward, you can forecast out to "destination date". The destination date still needs to fit within the forecast horizon from training.

```
In [ ]: # We will take the destination date as a last date in the test set.
dest = max(X_test[TIME_COLUMN_NAME])
y_pred_dest, xy_dest = fitted_model.forecast(forecast_destination=dest)

# This form also shows how we imputed the predictors which were not given. (Not so
xy_dest
```

Forecasting away from training data

Suppose we trained a model, some time passed, and now we want to apply the model without re-training. If the model "looks back" -- uses previous values of the target -- then we somehow need to provide those values to the model.

Forecasting after training

The notion of forecast origin comes into play: the forecast origin is **the last period for which we have seen the target value**. This applies per time-series, so each time-series can have a different forecast origin.

The part of data before the forecast origin is the **prediction context**. To provide the context values the model needs when it looks back, we pass definite values in `y_test` (aligned with corresponding times in `X_test`).

```
In [ ]: # generate the same kind of test data we trained on,
# but now make the train set much longer, so that the test set will be in the future
X_context, y_context, X_away, y_away = get_timeseries(train_len=42, # train data was
                                                    test_len=4,
                                                    time_column_name=TIME_COLUMN_NAME,
                                                    target_column_name=TARGET_COLUMN_NAME,
                                                    time_series_id_column_name=TIME_SERIES_ID_COLUMN_NAME,
                                                    time_series_number=2)

# end of the data we trained on
print(X_train.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].max())
# start of the data we want to predict on
print(X_away.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].min())
```

There is a gap of 12 hours between end of training and beginning of `X_away`. (It looks like 13 because all timestamps point to the start of the one hour periods.) Using only `X_away` will fail without adding context data for the model to consume.

```
In [ ]: try:
        y_pred_away, xy_away = fitted_model.forecast(X_away)
        xy_away
    except Exception as e:
        print(e)
```

How should we read that error message? The forecast origin is at the last time the model saw an actual value of `y` (the target). That was at the end of the training data! The model is attempting to forecast from the end of training data. But the requested forecast periods are past the forecast horizon. We need to provide a definite `y` value to establish the forecast origin.

We will use this helper function to take the required amount of context from the data preceding the testing data. Its definition is intentionally simplified to keep the idea in the clear.


```

In [ ]: def make_forecasting_query(fulldata, time_column_name, target_column_name, forecast

    """
    This function will take the full dataset, and create the query
    to predict all values of the time series from the `forecast_origin`
    forward for the next `horizon` horizons. Context from previous
    `lookback` periods will be included.

    fulldata: pandas.DataFrame                a time series dataset. Needs to contain X
    time_column_name: string                  which column (must be in fulldata) is the
    target_column_name: string                which column (must be in fulldata) is to b
    forecast_origin: datetime type            the last time we (pretend to) have target
    horizon: timedelta                       how far forward, in time units (not period
    lookback: timedelta                      how far back does the model look?

    Example:

    ...

    forecast_origin = pd.to_datetime('2012-09-01') + pd.DateOffset(days=5) # foreca
    print(forecast_origin)

    X_query, y_query = make_forecasting_query(data,
                                              forecast_origin = forecast_origin,
                                              horizon = pd.DateOffset(days=7), # 7 days into the future
                                              lookback = pd.DateOffset(days=1), # model has lag 1 period (
                                              )

    ...
    """

    X_past = fulldata[ (fulldata[ time_column_name ] > forecast_origin - lookback)
                      (fulldata[ time_column_name ] <= forecast_origin)
                      ]

    X_future = fulldata[ (fulldata[ time_column_name ] > forecast_origin) &
                        (fulldata[ time_column_name ] <= forecast_origin + horizon
                        ]

    y_past = X_past.pop(target_column_name).values.astype(np.float)
    y_future = X_future.pop(target_column_name).values.astype(np.float)

    # Now take y_future and turn it into question marks
    y_query = y_future.copy().astype(np.float) # because sometimes life hands you
    y_query.fill(np.NaN)

    print("X_past is " + str(X_past.shape) + " - shaped")
    print("X_future is " + str(X_future.shape) + " - shaped")
    print("y_past is " + str(y_past.shape) + " - shaped")
    print("y_query is " + str(y_query.shape) + " - shaped")

```

```

X_pred = pd.concat([X_past, X_future])
y_pred = np.concatenate([y_past, y_query])
return X_pred, y_pred

```

Let's see where the context data ends - it ends, by construction, just before the testing data starts.

```

In [ ]: print(X_context.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].agg(['min', 'max'])
print(X_away.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].agg(['min', 'max'])
X_context.tail(5)

```

```

In [ ]: # Since the length of the lookback is 3,
# we need to add 3 periods from the context to the request
# so that the model has the data it needs

# Put the X and y back together for a while.
# They like each other and it makes them happy.
X_context[TARGET_COLUMN_NAME] = y_context
X_away[TARGET_COLUMN_NAME] = y_away
fulldata = pd.concat([X_context, X_away])

# forecast origin is the last point of data, which is one 1-hr period before test
forecast_origin = X_away[TIME_COLUMN_NAME].min() - pd.DateOffset(hours=1)
# it is indeed the last point of the context
assert forecast_origin == X_context[TIME_COLUMN_NAME].max()
print("Forecast origin: " + str(forecast_origin))

# the model uses lags and rolling windows to look back in time
n_lookback_periods = max(lags)
lookback = pd.DateOffset(hours=n_lookback_periods)

horizon = pd.DateOffset(hours=forecast_horizon)

# now make the forecast query from context (refer to figure)
X_pred, y_pred = make_forecasting_query(fulldata, TIME_COLUMN_NAME, TARGET_COLUMN_NAME,
                                       forecast_origin, horizon, lookback)

# show the forecast request aligned
X_show = X_pred.copy()
X_show[TARGET_COLUMN_NAME] = y_pred
X_show

```

Note that the forecast origin is at 17:00 for both time-series, and periods from 18:00 are to be forecast.

```
In [ ]: # Now everything works
y_pred_away, xy_away = fitted_model.forecast(X_pred, y_pred)

# show the forecast aligned
X_show = xy_away.reset_index()
# without the generated features
X_show[['date', 'time_series_id', 'ext_predictor', '_automl_target_col']]
# prediction is in _automl_target_col
```

Forecasting farther than the forecast horizon

When the forecast destination, or the latest date in the prediction data frame, is farther into the future than the specified forecast horizon, the `forecast()` function will still make point predictions out to the later date using a recursive operation mode. Internally, the method recursively applies the regular forecaster to generate context so that we can forecast further into the future.

To illustrate the use-case and operation of recursive forecasting, we'll consider an example with a single time-series where the forecasting period directly follows the training period and is twice as long as the forecasting horizon given at training time.

 Recursive_forecast_overview

Internally, we apply the forecaster in an iterative manner and finish the forecast task in two iterations. In the first iteration, we apply the forecaster and get the prediction for the first forecast-horizon periods (`y_pred1`). In the second iteration, `y_pred1` is used as the context to produce the prediction for the next forecast-horizon periods (`y_pred2`). The combination of (`y_pred1` and `y_pred2`) gives the results for the total forecast periods.

A caveat: forecast accuracy will likely be worse the farther we predict into the future since errors are compounded with recursive application of the forecaster.

 Recursive_forecast_iter1  Recursive_forecast_iter2

```
In [ ]: # generate the same kind of test data we trained on, but with a single time-series
# as the forecast_horizon.
_, _, X_test_long, y_test_long = get_timeseries(train_len=n_train_periods,
                                                test_len=forecast_horizon*2,
                                                time_column_name=TIME_COLUMN_NAME,
                                                target_column_name=TARGET_COLUMN_NAME,
                                                time_series_id_column_name=TIME_S
                                                time_series_number=1)

print(X_test_long.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].min())
print(X_test_long.groupby(TIME_SERIES_ID_COLUMN_NAME)[TIME_COLUMN_NAME].max())
```

```
In [ ]: # forecast() function will invoke the recursive forecast method internally.  
y_pred_long, X_trans_long = fitted_model.forecast(X_test_long)  
y_pred_long
```

```
In [ ]: # What forecast() function does in this case is equivalent to iterating it twice over  
y_pred1, _ = fitted_model.forecast(X_test_long[:forecast_horizon])  
y_pred_all, _ = fitted_model.forecast(X_test_long, np.concatenate((y_pred1, np.full  
np.array_equal(y_pred_all, y_pred_long)
```

Confidence interval and distributional forecasts

AutoML cannot currently estimate forecast errors beyond the forecast horizon set during training, so the `forecast_quantiles()` function will return missing values for quantiles not equal to 0.5 beyond the forecast horizon.

```
In [ ]: fitted_model.forecast_quantiles(X_test_long)
```

Similarly with the simple scenarios illustrated above, forecasting farther than the forecast horizon in other scenarios like 'multiple time-series', 'Destination-date forecast', and 'forecast away from the training data' are also automatically handled by the `forecast()` function.

