Diviner

Release 0.1.1

Databricks

Oct 28, 2022

CONTENTS:

Diviner is an open source library for large-scale (multi-series) time series forecasting. It serves as a wrapper around popular open source forecasting libraries, providing a consolidated framework for simplified modeling of many discrete series components with an efficient high-level API.

To get started with Diviner, see the quickstart guide (*Quickstart Guide*).

Individual forecasting library API guides can be found here:

- *Grouped Prophet API*
- *Grouped Pmdarima API*

Examples of each of the wrapped forecasting libraries can be found *here*.

CHAPTER

ONE

DIVINER: GROUPED TIMESERIES FORECASTING AT SCALE

Diviner is an execution framework wrapper around popular open source time series forecasting libraries. The aim of the project is to simplify the creation, training, orchestration, and MLOps logistics associated with forecasting projects that involve the predictions of many discrete independent events.

1.1 Is this right for my project?

Diviner is meant to help with large-scale forecasting. Instead of describing each individual use case where it may be applicable, here is a non-exhaustive list of projects that it would fit well as a solution for:

- Forecasting regional sales within each country that a company does business in per day
- Predicting inventory demand at regional warehouses for thousands of products
- Forecasting traveler counts at each airport within a country daily
- Predicting electrical demand per neighborhood (or household) in a multi-state region

Each of these examples has a *common theme*:

- The data is temporally homogenous (all of the data is collected daily, hourly, weekly, etc.).
- There is a large number of individual models that need to be built due to the cardinality of the data.
- There is no guarantee of seasonal, trend, or residual homogeneity in each series.
- Varying levels of aggregation may be called for to solve different use cases.

The primary problem that Diviner solves is managing the execution of many discrete time-series modeling tasks. Diviner provides a high-level API and metadata management approach that relieves the operational burden of managing hundreds or thousands of individual models.

1.2 Grouped Modeling Wrappers

Currently, Diviner supports the following open source libraries for forecasting at scale:

- prophet
- pmdarima

1.3 Installing

Install Diviner from PyPI via:

```
pip install diviner
```

1.4 Documentation

Documentation, Examples, and Tutorials for Diviner can be found here.

1.5 Community & Contributing

For assistance with Diviner, see the docs.

Contributions to Diviner are welcome. To file a bug, request a new feature, or to contribute a feature request, please open a GitHub issue. The team will work with you to ensure that your contributions are evaluated and appropriate feedback is provided. See the contributing guidelines for submission guidance.

CHAPTER

TWO

QUICKSTART GUIDE

This guide will walk through the basic elements of Diviner's API, discuss the approach to data structures, and show a simple example of using a `GroupedProphet` model to generate forecasts for a multi-series data set.

Table of Contents

2.1 Installing Diviner

Diviner can be installed by running:

```
pip install diviner
```

Note: Diviner's underlying wrapped libraries have requirements that may require additional environment configuration, depending on your operating system's build environment. Libraries such as Prophet require gcc compilation libraries to build the underlying solvers that are used (PyStan). As such, it is recommended to use a version of Python >= 3.8 to ensure that the build environment resolves correctly and to ensure that the system's pip version is updated.

2.2 Downloading Examples

Examples can be viewed here for each supported forecasting library that is part of Diviner. To see examples of how to use Diviner to perform multi-series forecasting, download the examples by cloning the Diviner repository via:

```
git clone https://github.com/databricks/diviner
```

Once cloned, `cd` into `<root>/diviner/examples` to find both scripts and Jupyter notebook examples for each of the underlying wrapped forecasting libraries available. Running these examples will auto-generate a data set, generate the names of columns that define the groupings within the data set, and generate grouped forecasts for each of the defined groups in the data set.

2.3 Input Dataset Structure

The Diviner APIs use a DataFrame-driven approach to define and process distinct univariate time series data. The principle concept in use to accomplish the definition of discrete series information is *stacking*, wherein there is a single column that defines the endogenous regression term (a 'y value' that represents the data that is to be forecast), a column that contains the datetime (or date) values, and identifier column(s) that describe the distinct series data within the datetime and endogenous columns.

As an example, let's look at a denormalized data set that is typical in other types of machine learning. The data shown below is typically the result of transformation logic from the underlying stored data in a Data Warehouse, rather than how data such as sales figures would be stored in fact tables.

Table 1: Denormalized Data

Date	London Sales	Paris Sales	Munich Sales	Rome Sales
2021-01-01	154.9	83.9	113.2	17.5
2021-01-02	172.4	91.2	101.7	13.4
2021-01-03	191.2	87.8	99.9	18.1
2021-01-04	155.5	82.9	127.8	19.2
2021-01-05	168.4	92.8	104.4	9.6

This storage paradigm, while useful for creating a feature vector for supervised machine learning, is not efficient for large-scale forecasting of univariate time series data. Instead of requiring data in this format, Diviner requires a 'normalized' (stacked) data structure similar to how data would be stored in a database. Below is the representation of the same data, restructured how Diviner requires it to be structured.

Table 2: Stacked Data (Format for Diviner)

Date	Country	City	Sales
2021-01-01	United Kingdom	London	154.9
2021-01-02	United Kingdom	London	172.4
2021-01-03	United Kingdom	London	191.2
2021-01-04	United Kingdom	London	155.5
2021-01-05	United Kingdom	London	168.4
2021-01-01	France	Paris	83.9
2021-01-02	France	Paris	91.2
2021-01-03	France	Paris	87.8
2021-01-04	France	Paris	82.9
2021-01-05	France	Paris	92.8
2021-01-01	Germany	Munich	113.2
2021-01-02	Germany	Munich	101.7
2021-01-03	Germany	Munich	99.9
2021-01-04	Germany	Munich	127.8
2021-01-05	Germany	Munich	104.4
2021-01-01	Italy	Rome	17.5
2021-01-02	Italy	Rome	13.4
2021-01-03	Italy	Rome	18.1
2021-01-04	Italy	Rome	19.2
2021-01-05	Italy	Rome	9.6

This data structure paradigm enables several utilization paths that a 'pivoted' data structure, as shown above in the 'Denormalized Data' example, does not, such as:

- **The ability to dynamically group data based on hierarchical relationships.**

  - Group on {"Country", "City"}
  - Group on {"Country"}
  - Group on {"City"}
- Less data manipulation transformation code required when pulling data from source systems.
- Increased legibility of visual representations of the data.

2.4 Basic Example

To illustrate how to build forecasts for our country sales data above, here is an example of building a grouped forecast for each of the cities using the *GroupedProphet API*.

```
import pandas as pd
from diviner import GroupedProphet

series_data = pd.read_csv("/data/countries")

grouping_columns = ["Country", "City"]

grouped_prophet_model = GroupedProphet().fit(
    df=series_data,
    group_key_columns=grouping_columns
)

forecast_data = grouped_prophet_model.forecast(horizon=30, frequency="D")
```

This example parses the columns "Country" and "City", generates grouping keys, and builds Prophet models for each of the combinations present in the data set:

`{("United Kingdom", "London"), ("France", "Paris"), ("Germany", "Munich"), ("Italy", "Rome")}`

Alternatively, if we had multiple city values for each country and wished to forecast sales by country, we could have submitted `grouping_columns = ["Country"]`; this would have aggregated data and built models at the country level.

Following the model building, a 30 day forecast is generated (returned as a stacked consolidated Pandas DataFrame).

Note: For more in-depth examples (including per-group parameter extraction, cross validation metrics results, and serialization), see the *examples*.

CHAPTER

THREE

TUTORIALS AND EXAMPLES

The links below will direct you to tutorials and examples of how to use Diviner with various underlying forecasting libraries.

3.1 GroupedProphet example notebook

This notebook provides an example of the GroupedProphet API, complete with a self-contained data generator.

[1]:
```
import os
import sys
import logging
import itertools
import pandas as pd
import numpy as np
import string
import random
from datetime import timedelta, datetime
from collections import namedtuple
from diviner import GroupedProphet
```

```
Importing plotly failed. Interactive plots will not work.
```

3.1.1 Create a synthetic grouped data generator for time series

This will create seasonal data for multiple series worth of data, arranging the generated DataFrame such that identifying columns that define a distinct series are created (and returned when called).

[ ]:
```
def _generate_time_series(series_size: int):
    residuals = np.random.lognormal(
        mean=np.random.uniform(low=0.5, high=3.0),
        sigma=np.random.uniform(low=0.6, high=0.98),
        size=series_size,
    )
    trend = [
        np.polyval([23.0, 1.0, 5], x)
        for x in np.linspace(
```

(continues on next page)

(continued from previous page)

```
            start=0, stop=np.random.randint(low=0, high=4), num=series_size
        )
    ]
    seasonality = [
        90 * np.sin(2 * np.pi * 1000 * (i / (series_size * 200))) + 40
        for i in np.arange(0, series_size)
    ]

    return residuals + trend + seasonality + np.random.uniform(low=20.0, high=1000.0)


def _generate_grouping_columns(column_count: int, series_count: int):
    candidate_list = list(string.ascii_uppercase)
    candidates = random.sample(
        list(itertools.permutations(candidate_list, column_count)), series_count
    )
    column_names = sorted([f"key{x}" for x in range(column_count)], reverse=True)
    return [dict(zip(column_names, entries)) for entries in candidates]


def _generate_raw_df(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int,
):
    candidates = _generate_grouping_columns(column_count, series_count)
    start_date = datetime.strptime(start_dt, "%Y-%M-%d")
    dates = np.arange(
        start_date,
        start_date + timedelta(days=series_size * days_period),
        timedelta(days=days_period),
    )
    df_collection = []
    for entry in candidates:
        generated_series = _generate_time_series(series_size)
        series_dict = {"ds": dates, "y": generated_series}
        series_df = pd.DataFrame.from_dict(series_dict)
        for column, value in entry.items():
            series_df[column] = value
        df_collection.append(series_df)
    return pd.concat(df_collection)


def generate_example_data(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int = 1,
):
```

(continues on next page)

(continued from previous page)

```
    Structure = namedtuple("Structure", "df key_columns")
    data = _generate_raw_df(
        column_count, series_count, series_size, start_dt, days_period
    )
    key_columns = list(data.columns)

    for key in ["ds", "y"]:
        key_columns.remove(key)

    return Structure(data, key_columns)
```

Suppress optimizer stdout messaging

[2]:
```
class suppress_stdout_stderr():
    """
    Context manager to prevent the PyStan solver from filling stdout with a wall of text
    """

    def __init__(self):
        self.devnull_stdout = os.open(os.devnull, os.O_RDWR)
        self.devnull_stderr = os.open(os.devnull, os.O_RDWR)
        self.stdout = os.dup(1)
        self.stderr = os.dup(2)

    def __enter__(self):

        os.dup2(self.devnull_stdout, 1)
        os.dup2(self.devnull_stderr, 2)

    def __exit__(self, *_):

        os.dup2(self.stdout, 1)
        os.dup2(self.stderr, 2)

        os.close(self.devnull_stdout)
        os.close(self.devnull_stderr)
```

[3]:
```
logging.getLogger().setLevel(logging.CRITICAL)
```

3.1.2 Generate the stacked example data

[4]:
```
generated = generate_example_data(
    column_count=3,
    series_count=40,
    series_size=365*4,
    start_dt="2018-02-02",
    days_period=1
)
train = generated.df
key_columns = generated.key_columns
```

View the data structure

As can be seen, the elements that are required to define the stacked series are here: * column 'y' - the series data that we're going to use for training of the models * column 'ds' - the datetime values that correspond to each 'y' entry * columns ['key2', 'key1', 'key0'], the combination of which define a unique series.

[5]:
```
train
```

[5]:
```
                     ds           y key2 key1 key0
0    2018-01-02 00:02:00  139.540404    H    D    L
1    2018-01-03 00:02:00  145.638506    H    D    L
2    2018-01-04 00:02:00  144.510473    H    D    L
3    2018-01-05 00:02:00  145.096257    H    D    L
4    2018-01-06 00:02:00  145.901135    H    D    L
...                  ...         ...  ...  ...  ...
1455 2021-12-27 00:02:00   81.341661    Q    Z    O
1456 2021-12-28 00:02:00   82.385532    Q    Z    O
1457 2021-12-29 00:02:00   85.898131    Q    Z    O
1458 2021-12-30 00:02:00   88.452676    Q    Z    O
1459 2021-12-31 00:02:00   88.664839    Q    Z    O

[58400 rows x 5 columns]
```

3.1.3 Fit the GroupedProphet models on each of the distinct groups defined by the 'key_columns' argument.

[6]:
```
with suppress_stdout_stderr(): # Suppress stdout from PyStan
    logging.getLogger("prophet").setLevel(logging.CRITICAL) # Suppress INFO warnings
    grouped_prophet_model = GroupedProphet(n_changepoints=30, uncertainty_samples=0).fit(
    train, key_columns
)
```

3.1.4 Extract the parameters from the trained models

[7]:
```
params = grouped_prophet_model.extract_model_params()
params
```

[7]:

	grouping_key_columns	key2	key1	key0	changepoint_prior_scale	\
0	(key2, key1, key0)	A	I	S	0.05	
1	(key2, key1, key0)	A	Q	J	0.05	
2	(key2, key1, key0)	B	O	Q	0.05	
3	(key2, key1, key0)	C	O	Y	0.05	
4	(key2, key1, key0)	C	T	D	0.05	
5	(key2, key1, key0)	D	A	C	0.05	
6	(key2, key1, key0)	D	M	U	0.05	
7	(key2, key1, key0)	F	O	H	0.05	
8	(key2, key1, key0)	G	U	L	0.05	
9	(key2, key1, key0)	H	D	L	0.05	
10	(key2, key1, key0)	H	R	X	0.05	
11	(key2, key1, key0)	I	F	C	0.05	
12	(key2, key1, key0)	I	U	H	0.05	
13	(key2, key1, key0)	I	W	H	0.05	
14	(key2, key1, key0)	I	Y	P	0.05	
15	(key2, key1, key0)	J	A	S	0.05	
16	(key2, key1, key0)	J	T	G	0.05	
17	(key2, key1, key0)	K	H	Y	0.05	
18	(key2, key1, key0)	L	D	E	0.05	
19	(key2, key1, key0)	L	S	E	0.05	
20	(key2, key1, key0)	L	Z	S	0.05	
21	(key2, key1, key0)	M	D	T	0.05	
22	(key2, key1, key0)	M	O	X	0.05	
23	(key2, key1, key0)	M	W	L	0.05	
24	(key2, key1, key0)	N	Q	R	0.05	
25	(key2, key1, key0)	N	W	R	0.05	
26	(key2, key1, key0)	P	F	H	0.05	
27	(key2, key1, key0)	Q	X	Y	0.05	
28	(key2, key1, key0)	Q	Z	O	0.05	
29	(key2, key1, key0)	R	I	J	0.05	
30	(key2, key1, key0)	S	B	T	0.05	
31	(key2, key1, key0)	S	C	Z	0.05	
32	(key2, key1, key0)	T	A	Y	0.05	
33	(key2, key1, key0)	T	B	P	0.05	
34	(key2, key1, key0)	T	Z	F	0.05	
35	(key2, key1, key0)	U	T	S	0.05	
36	(key2, key1, key0)	W	G	R	0.05	
37	(key2, key1, key0)	X	I	A	0.05	
38	(key2, key1, key0)	X	Z	N	0.05	
39	(key2, key1, key0)	Y	J	F	0.05	

	changepoint_range	component_modes	\
0	0.8	{'additive': ['yearly', 'weekly', 'additive_te...	
1	0.8	{'additive': ['yearly', 'weekly', 'additive_te...	
2	0.8	{'additive': ['yearly', 'weekly', 'additive_te...	
3	0.8	{'additive': ['yearly', 'weekly', 'additive_te...	
4	0.8	{'additive': ['yearly', 'weekly', 'additive_te...	

(continues on next page)

(continued from previous page)

```
5                0.8  {'additive': ['yearly', 'weekly', 'additive_te...
6                0.8  {'additive': ['yearly', 'weekly', 'additive_te...
7                0.8  {'additive': ['yearly', 'weekly', 'additive_te...
8                0.8  {'additive': ['yearly', 'weekly', 'additive_te...
9                0.8  {'additive': ['yearly', 'weekly', 'additive_te...
10               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
11               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
12               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
13               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
14               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
15               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
16               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
17               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
18               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
19               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
20               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
21               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
22               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
23               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
24               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
25               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
26               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
27               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
28               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
29               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
30               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
31               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
32               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
33               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
34               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
35               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
36               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
37               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
38               0.8  {'additive': ['yearly', 'weekly', 'additive_te...
39               0.8  {'additive': ['yearly', 'weekly', 'additive_te...

   country_holidays daily_seasonality extra_regressors  ...  \
0              None              auto               {}  ...
1              None              auto               {}  ...
2              None              auto               {}  ...
3              None              auto               {}  ...
4              None              auto               {}  ...
5              None              auto               {}  ...
6              None              auto               {}  ...
7              None              auto               {}  ...
8              None              auto               {}  ...
9              None              auto               {}  ...
10             None              auto               {}  ...
11             None              auto               {}  ...
12             None              auto               {}  ...
13             None              auto               {}  ...
14             None              auto               {}  ...
```

(continues on next page)

(continued from previous page)

```
15              None              auto               {}  ...
16              None              auto               {}  ...
17              None              auto               {}  ...
18              None              auto               {}  ...
19              None              auto               {}  ...
20              None              auto               {}  ...
21              None              auto               {}  ...
22              None              auto               {}  ...
23              None              auto               {}  ...
24              None              auto               {}  ...
25              None              auto               {}  ...
26              None              auto               {}  ...
27              None              auto               {}  ...
28              None              auto               {}  ...
29              None              auto               {}  ...
30              None              auto               {}  ...
31              None              auto               {}  ...
32              None              auto               {}  ...
33              None              auto               {}  ...
34              None              auto               {}  ...
35              None              auto               {}  ...
36              None              auto               {}  ...
37              None              auto               {}  ...
38              None              auto               {}  ...
39              None              auto               {}  ...

   seasonality_prior_scale specified_changepoints  \
0                     10.0                   False
1                     10.0                   False
2                     10.0                   False
3                     10.0                   False
4                     10.0                   False
5                     10.0                   False
6                     10.0                   False
7                     10.0                   False
8                     10.0                   False
9                     10.0                   False
10                    10.0                   False
11                    10.0                   False
12                    10.0                   False
13                    10.0                   False
14                    10.0                   False
15                    10.0                   False
16                    10.0                   False
17                    10.0                   False
18                    10.0                   False
19                    10.0                   False
20                    10.0                   False
21                    10.0                   False
22                    10.0                   False
23                    10.0                   False
24                    10.0                   False
```

(continues on next page)

(continued from previous page)

```
25                     10.0                  False
26                     10.0                  False
27                     10.0                  False
28                     10.0                  False
29                     10.0                  False
30                     10.0                  False
31                     10.0                  False
32                     10.0                  False
33                     10.0                  False
34                     10.0                  False
35                     10.0                  False
36                     10.0                  False
37                     10.0                  False
38                     10.0                  False
39                     10.0                  False

                                      stan_backend               start  \
0   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
1   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
2   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
3   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
4   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
5   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
6   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
7   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
8   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
9   <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
10  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
11  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
12  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
13  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
14  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
15  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
16  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
17  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
18  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
19  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
20  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
21  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
22  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
23  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
24  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
25  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
26  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
27  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
28  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
29  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
30  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
31  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
32  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
33  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
34  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
```

(continues on next page)

(continued from previous page)

```
35  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
36  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
37  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
38  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00
39  <prophet.models.PyStanBackend object at 0x7fd8... 2018-01-02 00:02:00

     t_scale train_holiday_names uncertainty_samples weekly_seasonality  \
0  1459 days                None                    0                auto
1  1459 days                None                    0                auto
2  1459 days                None                    0                auto
3  1459 days                None                    0                auto
4  1459 days                None                    0                auto
5  1459 days                None                    0                auto
6  1459 days                None                    0                auto
7  1459 days                None                    0                auto
8  1459 days                None                    0                auto
9  1459 days                None                    0                auto
10 1459 days                None                    0                auto
11 1459 days                None                    0                auto
12 1459 days                None                    0                auto
13 1459 days                None                    0                auto
14 1459 days                None                    0                auto
15 1459 days                None                    0                auto
16 1459 days                None                    0                auto
17 1459 days                None                    0                auto
18 1459 days                None                    0                auto
19 1459 days                None                    0                auto
20 1459 days                None                    0                auto
21 1459 days                None                    0                auto
22 1459 days                None                    0                auto
23 1459 days                None                    0                auto
24 1459 days                None                    0                auto
25 1459 days                None                    0                auto
26 1459 days                None                    0                auto
27 1459 days                None                    0                auto
28 1459 days                None                    0                auto
29 1459 days                None                    0                auto
30 1459 days                None                    0                auto
31 1459 days                None                    0                auto
32 1459 days                None                    0                auto
33 1459 days                None                    0                auto
34 1459 days                None                    0                auto
35 1459 days                None                    0                auto
36 1459 days                None                    0                auto
37 1459 days                None                    0                auto
38 1459 days                None                    0                auto
39 1459 days                None                    0                auto

        y_scale yearly_seasonality
0    981.029852               auto
1    616.079172               auto
2    903.068555               auto
```

(continues on next page)

(continued from previous page)

```
3   1031.305845               auto
4   1043.772833               auto
5   1195.890246               auto
6    628.177374               auto
7   1247.246604               auto
8    279.778234               auto
9     313.89009               auto
10   310.707764               auto
11  1126.160902               auto
12  1287.238874               auto
13   606.188878               auto
14   301.869476               auto
15  1364.239088               auto
16   980.384209               auto
17   917.979213               auto
18  1079.828476               auto
19   915.850019               auto
20   828.615341               auto
21   336.260444               auto
22   842.892671               auto
23   242.552077               auto
24   475.901684               auto
25  1078.139666               auto
26   818.370678               auto
27  1065.844067               auto
28   182.534667               auto
29   592.192065               auto
30    630.32091               auto
31  1013.812286               auto
32   785.534176               auto
33    804.37043               auto
34   937.109632               auto
35   744.092648               auto
36  1011.435103               auto
37   772.332066               auto
38  1300.590936               auto
39  1074.841913               auto

[40 rows x 29 columns]
```

3.1.5 Cross validate each model to return the scoring metrics for each group

[8]:
```
with suppress_stdout_stderr():
    metrics = grouped_prophet_model.cross_validate_and_score(
        horizon="30 days",
        period="365 days",
        initial="730 days",
        parallel="threads",
        rolling_window=0.05,
        monthly=False
    )
```

[9]:
```
metrics
```

[9]:

	grouping_key_columns	key2	key1	key0	mse	rmse	mae	mape	mdape	smape
0	(key2, key1, key0)	A	I	S	262.540068	15.062691	13.915674	0.016464	0.015734	0.016646
1	(key2, key1, key0)	A	Q	J	469.382836	20.242075	19.657388	0.041805	0.042604	0.042899
2	(key2, key1, key0)	B	O	Q	2817.155728	50.511383	42.431716	0.055081	0.048438	0.057438
3	(key2, key1, key0)	C	O	Y	585.259877	22.515334	21.410191	0.024674	0.023593	0.025063
4	(key2, key1, key0)	C	T	D	757.678167	24.916772	23.975279	0.026917	0.025814	0.027394
5	(key2, key1, key0)	D	A	C	1798.440133	39.779557	37.261880	0.037357	0.034757	0.038267
6	(key2, key1, key0)	D	M	U	2947.507801	51.200966	44.124248	0.163164	0.143661	0.148713
7	(key2, key1, key0)	F	O	H	2275.917905	45.236682	38.896113	0.035109	0.032875	0.036025
8	(key2, key1, key0)	G	U	L	11708.036260	107.953013	86.074602			
9	(key2, key1, key0)	H	D	L	10671.352270	103.017873	81.313511			
10	(key2, key1, key0)	H	R	X	12140.010983	109.888034	88.612042			
11	(key2, key1, key0)	I	F	C	882.160772	27.834954	26.533773			
12	(key2, key1, key0)	I	U	H	342.033309	17.099936	16.129785			
13	(key2, key1, key0)	I	W	H	664.345357	21.451403	20.532673			
14	(key2, key1, key0)	I	Y	P	10648.205364	102.705291	85.009991			
15	(key2, key1, key0)	J	A	S	3753.890398	59.210502	50.237830			
16	(key2, key1, key0)	J	T	G	712.125767	25.003669	22.405640			
17	(key2, key1, key0)	K	H	Y	443.912007	19.960495	17.500978			
18	(key2, key1, key0)	L	D	E	1467.502779	36.169521	34.984144			
19	(key2, key1, key0)	L	S	E	260.607376	15.042063	14.491292			
20	(key2, key1, key0)	L	Z	S	540.464149	21.388109	20.170990			
21	(key2, key1, key0)	M	D	T	11475.321478	106.907205	85.239846			
22	(key2, key1, key0)	M	O	X	980.723118	29.443091	27.917302			
23	(key2, key1, key0)	M	W	L	11471.704236	106.876550	85.262401			
24	(key2, key1, key0)	N	Q	R	12126.909866	109.861268	87.720958			
25	(key2, key1, key0)	N	W	R	831.001845	26.453888	25.005874			
26	(key2, key1, key0)	P	F	H	3385.137185	55.836240	47.652862			
27	(key2, key1, key0)	Q	X	Y	696.874649	24.708330	22.547868			
28	(key2, key1, key0)	Q	Z	O	10837.921043	103.902697	83.108939			
29	(key2, key1, key0)	R	I	J	1338.534546	34.192363	33.305858			
30	(key2, key1, key0)	S	B	T	1228.447613	33.062395	32.205870			
31	(key2, key1, key0)	S	C	Z	2721.175850	49.565001	43.222989			
32	(key2, key1, key0)	T	A	Y	580.708758	22.265963	21.240699			
33	(key2, key1, key0)	T	B	P	1559.715623	36.489070	33.756588			
34	(key2, key1, key0)	T	Z	F	257.901818	14.821516	14.286622			
35	(key2, key1, key0)	U	T	S	1707.310374	38.692001	37.233078			
36	(key2, key1, key0)	W	G	R	443.086490	19.538874	18.915095			
37	(key2, key1, key0)	X	I	A	2638.423929	48.920116	46.225857			
38	(key2, key1, key0)	X	Z	N	1164.674674	31.577529	30.914779			
39	(key2, key1, key0)	Y	J	F	238.461604	13.881420	13.325958			

(continues on next page)

(continued from previous page)

```
8    1.332295   1.592902  0.637987
9    0.704081   0.857213  0.439851
10   3.650613   4.432163  0.900864
11   0.027547   0.027449  0.028020
12   0.013927   0.013389  0.014053
13   0.047146   0.050368  0.048850
14   1.322913   1.517187  0.629965
15   0.049434   0.044973  0.051284
16   0.026714   0.023557  0.027216
17   0.021829   0.020405  0.022165
18   0.043103   0.042812  0.044233
19   0.018544   0.018222  0.018755
20   0.029981   0.029070  0.030580
21   0.719624   0.872762  0.455814
22   0.040580   0.037341  0.041620
23   6.551329   6.945873  0.957528
24   0.428517   0.512430  0.317728
25   0.027805   0.028192  0.028326
26   0.094587   0.090453  0.101384
27   0.024816   0.021517  0.025236
28  22.734702  17.173468  1.136795
29   0.074509   0.076687  0.078025
30   0.068078   0.068130  0.070888
31   0.051009   0.045872  0.052886
32   0.032808   0.031591  0.033493
33   0.053180   0.050840  0.055136
34   0.017740   0.017585  0.017936
35   0.063790   0.064129  0.066334
36   0.022123   0.021703  0.022428
37   0.084699   0.086252  0.089271
38   0.029353   0.029181  0.029889
39   0.014331   0.013884  0.014468
```

3.1.6 Save the model

[10]:
```
save_path = "/tmp/model/grouped_prophet"
grouped_prophet_model.save(save_path)
```

3.1.7 Load the model

[11]:
```
loaded_model = GroupedProphet.load(save_path)
```

3.1.8 Generate forecasts for each group

Forecasting is not limited to the frequency of the originating series for each group. The training data, with a periodicity of daily, can have weekly predictions generated by specifying the frequency of `W`, as shown below.

[12]:
```
forecast = loaded_model.forecast(horizon=16, frequency="W")
```

[13]:
```
forecast
```

[13]:
```
    grouping_key_columns key2 key1 key0                  ds        trend  \
0     (key2, key1, key0)    A    I    S 2022-01-02 00:02:00   946.265699
1     (key2, key1, key0)    A    I    S 2022-01-09 00:02:00   949.014237
2     (key2, key1, key0)    A    I    S 2022-01-16 00:02:00   951.762775
3     (key2, key1, key0)    A    I    S 2022-01-23 00:02:00   954.511313
4     (key2, key1, key0)    A    I    S 2022-01-30 00:02:00   957.259851
..                   ...  ...  ...  ...                 ...          ...
635   (key2, key1, key0)    Y    J    F 2022-03-20 00:02:00  1063.621168
636   (key2, key1, key0)    Y    J    F 2022-03-27 00:02:00  1066.317513
637   (key2, key1, key0)    Y    J    F 2022-04-03 00:02:00  1069.013858
638   (key2, key1, key0)    Y    J    F 2022-04-10 00:02:00  1071.710203
639   (key2, key1, key0)    Y    J    F 2022-04-17 00:02:00  1074.406547

     additive_terms    weekly     yearly  multiplicative_terms         yhat
0        -54.783190 -0.041724 -54.741466                   0.0   891.482510
1        -45.960688 -0.041724 -45.918964                   0.0   903.053549
2        -36.963503 -0.041724 -36.921779                   0.0   914.799272
3        -26.911199 -0.041724 -26.869476                   0.0   927.600114
4        -15.607711 -0.041724 -15.565987                   0.0   941.652140
..              ...       ...        ...                   ...          ...
635       60.727302  0.220967  60.506335                   0.0  1124.348470
636       71.955841  0.220967  71.734874                   0.0  1138.273354
637       82.569743  0.220967  82.348776                   0.0  1151.583600
638       91.797486  0.220967  91.576519                   0.0  1163.507689
639       99.418800  0.220967  99.197833                   0.0  1173.825347

[640 rows x 11 columns]
```

[14]:
```
os.remove(save_path)
```

3.2 GroupedProphet Example Scripts

The scripts included below show the various options available for utilizing the `GroupedProphet` API. For an alternative view of these examples with data visualizations, see the *notebooks here*

Scripts

- *GroupedProphet Example*
- *GroupedProphet Subset Group Prediction Example*
- *Supplementary*

3.2.1 GroupedProphet Example

This script shows a simple example of training a series of Prophet models, saving a `GroupedProphet` instance, loading that instance, cross validating through backtesting, and generating a forecast for each group.

Listing 1: GroupedProphet Script

```
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedProphet


def execute_grouped_prophet():
    """
    This function call will generate synthetic group time series data in a normalized␣
↪format.
    The structure will be of:

    ============ ====== =========== =========== ===========
    ds           y      group_key_1 group_key_2 group_key_3
    ============ ====== =========== =========== ===========
    "2016-02-01" 1234.5 A           B           C
    ============ ====== =========== =========== ===========

    With the grouping key values that are generated per ``ds`` and ``y`` values assigned in␣
↪a
    non-deterministic fashion.

    For utililzation of this API, the normalized representation of the data is required,␣
↪such that
    a particular target variables' data 'y' and the associated indexed datetime values in ``
↪'ds'``
    are 'stacked' (unioned) from a more traditional denormalized data storage paradigm.

    For guidance on this data transposition from denormalized representations, see:
    https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html
    """

    generated_data = generate_example_data(
        column_count=3,
```

(continues on next page)

(continued from previous page)

```
        series_count=10,
        series_size=365 * 5,
        start_dt="2016-02-01",
        days_period=1,
    )

    # Extract the normalized grouped datetime series data
    training_data = generated_data.df

    # Extract the names of the grouping columns that define the unique series data
    grouping_key_columns = generated_data.key_columns

    # Create a GroupedProphet model instance
    grouped_model = GroupedProphet(n_changepoints=20, uncertainty_samples=0).fit(
        training_data, grouping_key_columns
    )

    # Save the model to the local file system
    save_path = "/tmp/grouped_prophet.gpm"
    grouped_model.save(path="/tmp/grouped_prophet.gpm")

    # Load the model from the local storage location
    retrieved_model = GroupedProphet.load(save_path)

    # Score the model and print the results
    model_scores = retrieved_model.cross_validate_and_score(
        horizon="30 days",
        period="180 days",
        initial="365 days",
        parallel="threads",
        rolling_window=0.05,
        monthly=False,
    )

    print(f"Model scores:\n{model_scores.to_string()}")

    # Run a forecast for each group
    forecasts = retrieved_model.forecast(horizon=20, frequency="D")

    print(f"Forecasted data:\n{forecasts[:50].to_string()}")

    # Extract the parameters from each model for logging
    params = retrieved_model.extract_model_params()

    print(f"Model parameters:\n{params.to_string()}")


if __name__ == "__main__":
    execute_grouped_prophet()
```

3.2.2 GroupedProphet Subset Group Prediction Example

This script shows a simple example of training a series of Prophet models and generating a group subset prediction.

Listing 2: GroupedProphet Subset Groups Script

```python
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedProphet

if __name__ == "__main__":

    generated_data = generate_example_data(
        column_count=3,
        series_count=10,
        series_size=365 * 5,
        start_dt="2016-02-01",
        days_period=1,
    )

    # Extract the normalized grouped datetime series data
    training_data = generated_data.df

    # Extract the names of the grouping columns that define the unique series data
    group_key_columns = generated_data.key_columns

    # Create a GroupedProphet model instance
    grouped_model = GroupedProphet(n_changepoints=20, uncertainty_samples=0).fit(
        training_data, group_key_columns
    )

    # Get a subset of group keys to generate forecasts for
    group_df = training_data.copy()
    group_df["groups"] = list(zip(*[group_df[c] for c in group_key_columns]))
    distinct_groups = group_df["groups"].unique()
    groups_to_predict = list(distinct_groups[:3])

    print("-" * 65)
    print(f"\nUnique groups that have been modeled: \n{distinct_groups}\n")
    print(f"Subset of groups to generate predictions for: \n{groups_to_predict}\n")
    print("-" * 65)

    forecasts = grouped_model.predict_groups(
        groups=groups_to_predict,
        horizon=60,
        frequency="D",
        predict_col="forecast_values",
        on_error="warn",
    )

    print(f"\nForecast values:\n{forecasts.to_string()}")
```

3.2.3 Supplementary

Note: To run these examples for yourself with the data generator example, utilize the following code:

Listing 3: Synthetic Data Generator

```
import itertools
import pandas as pd
import numpy as np
import string
import random
from datetime import timedelta, datetime
from collections import namedtuple


def _generate_time_series(series_size: int):
    residuals = np.random.lognormal(
        mean=np.random.uniform(low=0.5, high=3.0),
        sigma=np.random.uniform(low=0.6, high=0.98),
        size=series_size,
    )
    trend = [
        np.polyval([23.0, 1.0, 5], x)
        for x in np.linspace(start=0, stop=np.random.randint(low=0, high=4), num=series_
↪size)
    ]
    seasonality = [
        90 * np.sin(2 * np.pi * 1000 * (i / (series_size * 200))) + 40
        for i in np.arange(0, series_size)
    ]

    return residuals + trend + seasonality + np.random.uniform(low=20.0, high=1000.0)


def _generate_grouping_columns(column_count: int, series_count: int):
    candidate_list = list(string.ascii_uppercase)
    candidates = random.sample(
        list(itertools.permutations(candidate_list, column_count)), series_count
    )
    column_names = sorted([f"key{x}" for x in range(column_count)], reverse=True)
    return [dict(zip(column_names, entries)) for entries in candidates]


def _generate_raw_df(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int,
):
    candidates = _generate_grouping_columns(column_count, series_count)
```

(continues on next page)

(continued from previous page)

```
    start_date = datetime.strptime(start_dt, "%Y-%M-%d")
    dates = np.arange(
        start_date,
        start_date + timedelta(days=series_size * days_period),
        timedelta(days=days_period),
    )
    df_collection = []
    for entry in candidates:
        generated_series = _generate_time_series(series_size)
        series_dict = {"ds": dates, "y": generated_series}
        series_df = pd.DataFrame.from_dict(series_dict)
        for column, value in entry.items():
            series_df[column] = value
        df_collection.append(series_df)
    return pd.concat(df_collection)


def generate_example_data(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int = 1,
):

    Structure = namedtuple("Structure", "df key_columns")
    data = _generate_raw_df(column_count, series_count, series_size, start_dt, days_
↪period)
    key_columns = list(data.columns)

    for key in ["ds", "y"]:
        key_columns.remove(key)

    return Structure(data, key_columns)
```

3.3 Notebook example of Diviner's GroupedPmdarima API

This notebook shows a comparison of 4 primary means of utilizing the GroupedPmdarima API in Diviner.

3.3.1 Examples shown

- Standard ARIMA
- A user-supplied manual configuration of ARIMA models to be built based on the configured order terms. This approach is useful for hierarchical multi-series optimization techniques (where ordering terms are determined by evaluating the optimal parameters from a higher level aggregation of disparate series that share a similar seasonality with one another).
- Note that this is the fastest execution available and is recommended to be used if there is homogeny amongst a large collection of individual series (i.e., forecasting SKU scales at 500 different stores would use a global

SKU forecasting model through AutoARIMA to determine the optimal ordering terms, then apply those to each individual store's ARIMA models through this mode).

- AutoARIMA
- An automate approach that will perform a best-effort optimization of ordering terms.
- Seasonal AutoARIMA
- A much slower, but, depending on the nature of the series data, potentially much more accurate model for each series.
- Pipeline preprocessing + AutoARIMA
- This mode applies a data transformer (exogeneous transformers are currently not supported) such as a LogEndogTransformer or BoxCoxEndogTransformer. Depending on the nature of the data, this may dramatically improve the forecasting quality.

[2]:
```
import itertools
import pandas as pd
import numpy as np
import string
import random
from datetime import timedelta, datetime
from collections import namedtuple
import matplotlib.pyplot as plt
import matplotlib.cm as cmx

from pmdarima.arima.arima import ARIMA
from pmdarima.arima.auto import AutoARIMA
from pmdarima.pipeline import Pipeline
from pmdarima.preprocessing import LogEndogTransformer
from pmdarima.model_selection import SlidingWindowForecastCV
from diviner import GroupedPmdarima, PmdarimaAnalyzer
```

[3]:
```
def _build_trend(size):
    raw_trend = (
        np.arange(size) * np.random.uniform(-0.05, 0.2)
    ) + np.random.randint(200, 500)
    return raw_trend


def _build_seasonality(size, period):
    repeated_x = np.arange(period) + 3

    raw_values = np.where(repeated_x < 5, repeated_x**4,
                          np.where(repeated_x < 7, repeated_x**3,
                                   repeated_x**2)
                         )

    seasonality = raw_values
    for i in range(int(size / period) - 1):
        seasonality = np.append(seasonality, raw_values)
    return seasonality * np.random.randint(1, 4)
```

(continues on next page)

(continued from previous page)

```
def _build_residuals(size):
    return np.random.randn(size) * np.random.randint(4, 10)


def _generate_time_series(size, seasonal_period):
    return (
        _build_trend(size)
        + _build_seasonality(size, seasonal_period)
        + _build_residuals(size)
    )


def _generate_grouping_columns(column_count: int, series_count: int):
    candidate_list = list(string.ascii_uppercase)
    candidates = random.sample(
        list(itertools.permutations(candidate_list, column_count)), series_count
    )
    column_names = sorted([f"key{x}" for x in range(column_count)], reverse=True)
    return [dict(zip(column_names, entries)) for entries in candidates]


def _generate_raw_df(
    column_count: int,
    series_count: int,
    series_size: int,
    series_seasonal_period: int,
    start_dt: str,
    days_period: int,
):
    candidates = _generate_grouping_columns(column_count, series_count)
    start_date = datetime.strptime(start_dt, "%Y-%M-%d")
    dates = np.arange(
        start_date,
        start_date + timedelta(days=series_size * days_period),
        timedelta(days=days_period),
    )
    df_collection = []
    for entry in candidates:
        generated_series = _generate_time_series(series_size, series_seasonal_period)
        series_dict = {"ds": dates, "y": generated_series}
        series_df = pd.DataFrame.from_dict(series_dict)
        for column, value in entry.items():
            series_df[column] = value
        df_collection.append(series_df)
    return pd.concat(df_collection)


def generate_example_data(
    column_count: int,
    series_count: int,
    series_size: int,
    series_seasonal_period: int,
```

(continues on next page)

(continued from previous page)

```
    start_dt: str,
    days_period: int = 1,
):

    Structure = namedtuple("Structure", "df key_columns")
    data = _generate_raw_df(
        column_count, series_count, series_size, series_seasonal_period, start_dt, days_
↪period
    )
    key_columns = list(data.columns)

    for key in ["ds", "y"]:
        key_columns.remove(key)

    return Structure(data, key_columns)

def plot_grouped_series(df, key_columns, time_col, y_col):
    grouped = df.groupby(key_columns)
    ncols = 1
    nrows = int(np.ceil(grouped.ngroups/ncols))
    fig, axes = plt.subplots(nrows=nrows,
                             ncols=ncols,
                             figsize=(16, 6*grouped.ngroups),
                             sharey=False,
                             sharex=False
                            )
    cmap = [cmx.Dark2(x) for x in np.linspace(0.0, 1.0, grouped.ngroups)]
    i=0
    for (key, ax) in zip(grouped.groups.keys(), axes.flatten()):
        ser = grouped.get_group(key)
        rgb = cmap[i]
        ax.plot(ser[time_col], ser[y_col], label="y value", c=rgb)
        ax.legend()
        ax.title.set_text(f"Group: {key}")
        i+=1
    plt.show()

def plot_grouped_series_forecast(df, key_columns, time_col, y_col, yhat_lower_col, yhat_
↪upper_col):

    grouped = df.groupby(key_columns)
    ncols = 1
    nrows = int(np.ceil(grouped.ngroups/ncols))
    fig, axes = plt.subplots(nrows=nrows,
                             ncols=ncols,
                             figsize=(16, 8*grouped.ngroups),
                             sharey=False,
                             sharex=False
                            )
    cmap = [cmx.Dark2(x) for x in np.linspace(0.0, 1.0, grouped.ngroups)]
    i=0
    for (key, ax) in zip(grouped.groups.keys(), axes.flatten()):
```

(continues on next page)

(continued from previous page)

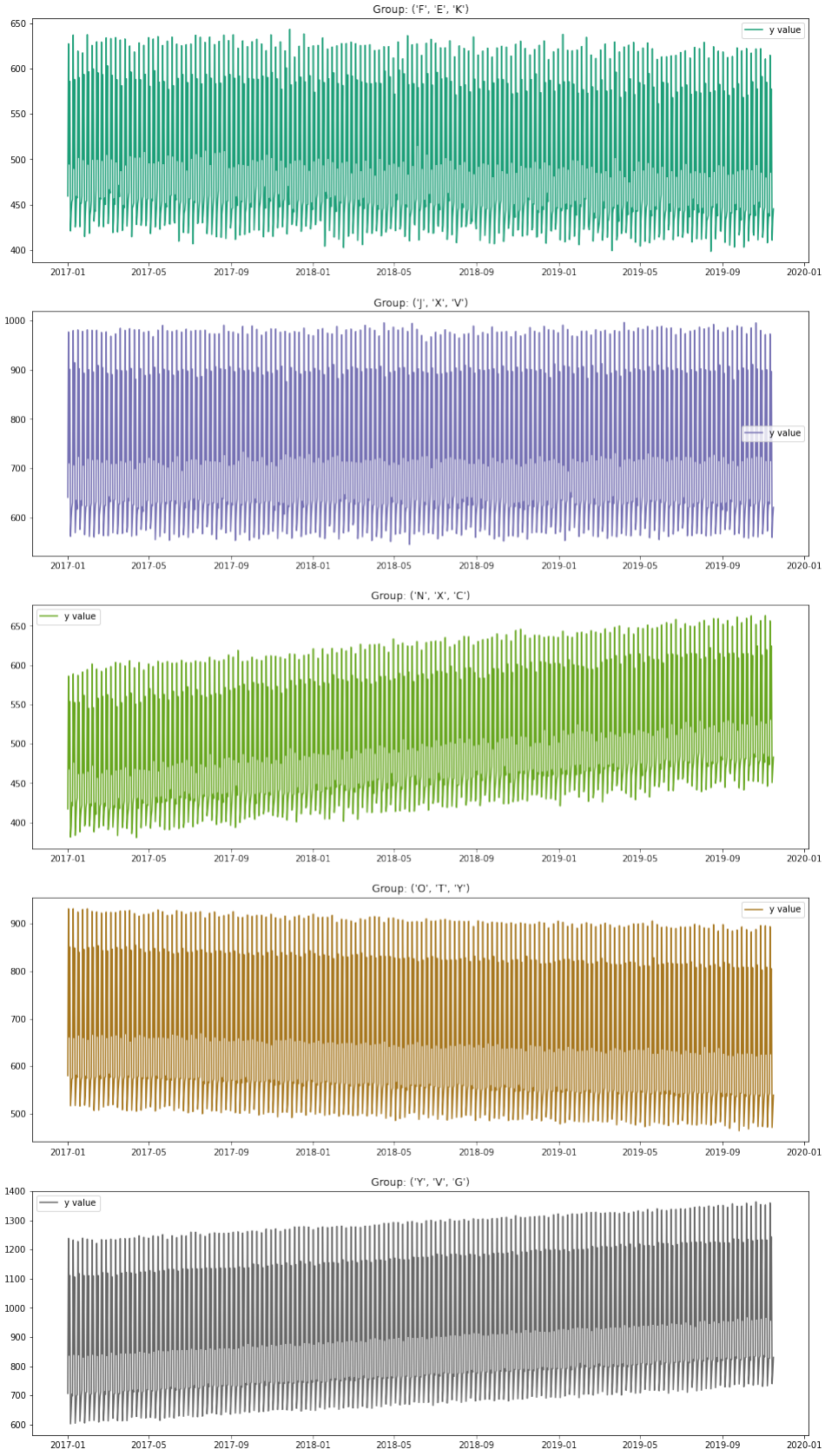
```
        ser = grouped.get_group(key)
        rgb = cmap[i]
        ax.plot(ser[time_col], ser[y_col], label="y value", c=rgb)
        ax.fill_between(ser[time_col],
                        ser[yhat_lower_col],
                        ser[yhat_upper_col],
                        color=rgb,
                        alpha=0.3,
                        label="error"
                       )
        ax.legend(loc="upper left")
        ax.title.set_text(f"Group: {key}")
        ax.grid(color=rgb, linewidth=0.5, alpha=0.5)
        i+=1
    plt.show()
```

[4]:
```
data = generate_example_data(3, 5, 1050, 7, "2017-01-01", 1)
```

3.3.2 View the synthetic generated data

Note that this is a randomly generated data set.

[5]:
```
plot_grouped_series(data.df, data.key_columns, "ds", "y")
```


Group: ('F', 'E', 'K')
y value
Group: ('J', 'X', 'V')
y value
Group: ('N', 'X', 'C')
y value
Group: ('O', 'T', 'Y')
y value
Group: ('Y', 'V', 'G')
y value
2017-01
2017-05
2017-09
2018-01
2018-05
2018-09
2019-01
2019-05
2019-09
2020-01

3.3.3 ARIMA manually defined ordering terms

[6]:
```
data_utils = PmdarimaAnalyzer(data.df, data.key_columns, "y", "ds")

ndiffs = data_utils.calculate_ndiffs(alpha=0.05, test="kpss", max_d=7)
ndiffs
```

[6]:
```
{('F', 'E', 'K'): 1,
 ('J', 'X', 'V'): 0,
 ('N', 'X', 'C'): 1,
 ('O', 'T', 'Y'): 1,
 ('Y', 'V', 'G'): 1}
```

The results above for the ndiffs method shows that each group's differencing value should be set to '1'. This isn't surprising based on how the generated data was created. Real world data sets may have different optimal 'd' values, though. Performing a manual validation can dramatically reduce optimization time for the AutoARIMA methods (which will be covered further down in this example notebook).

Let's check the seasonal differencing as well.

[7]:
```
nsdiffs = data_utils.calculate_nsdiffs(m=20, test="ocsb", max_D=30)
nsdiffs
```

[7]:
```
{('F', 'E', 'K'): 0,
 ('J', 'X', 'V'): 0,
 ('N', 'X', 'C'): 0,
 ('O', 'T', 'Y'): 0,
 ('Y', 'V', 'G'): 0}
```

Calculate the acf values for each group to aid in setting the ARIMA 'p' parameter.

[8]:
```
acf = data_utils.calculate_acf(alpha=0.05, qstat=True)
acf
```

[8]:
```
{('F',
  'E',
  'K'): {'acf': array([ 1.        , -0.08070348,  0.17113533, -0.57432053, -0.57269233,
          0.16892029, -0.08007331,  0.9866866 , -0.08029439,  0.16979978,
         -0.57130432, -0.56881352,  0.16684712, -0.07890206,  0.98009748,
         -0.07907823,  0.16909447, -0.56757814, -0.56480278,  0.16576373,
         -0.07812178,  0.97376666, -0.07840144,  0.16723731, -0.56373238,
         -0.56227819,  0.1649875 , -0.07771043,  0.96709519, -0.07748238,
          0.16583409]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
         [-1.41189283e-01, -2.02176793e-02],
         [ 1.10256857e-01,  2.32013805e-01],
         [-6.36934311e-01, -5.11706747e-01],
         [-6.52278609e-01, -4.93106048e-01],
         [ 7.54654815e-02,  2.62375098e-01],
         [-1.74638559e-01,  1.44919437e-02],
         [ 8.91873617e-01,  1.08149958e+00],
         [-2.07231513e-01,  4.66427328e-02],
```

(continues on next page)

(continued from previous page)

```
        [ 4.26769784e-02,  2.96922590e-01],
        [-6.99254204e-01, -4.43354434e-01],
        [-7.05778407e-01, -4.31848638e-01],
        [ 2.14964890e-02,  3.12197760e-01],
        [-2.24951705e-01,  6.71475893e-02],
        [ 8.33891965e-01,  1.12630299e+00],
        [-2.47615343e-01,  8.94588851e-02],
        [ 4.21665597e-04,  3.37767275e-01],
        [-7.36869989e-01, -3.98286283e-01],
        [-7.40918901e-01, -3.88686666e-01],
        [-1.68589604e-02,  3.48386416e-01],
        [-2.61294110e-01,  1.05050546e-01],
        [ 7.90472475e-01,  1.15706084e+00],
        [-2.79734390e-01,  1.22931504e-01],
        [-3.42072984e-02,  3.68681928e-01],
        [-7.65684304e-01, -3.61780464e-01],
        [-7.69907420e-01, -3.54648952e-01],
        [-4.81397856e-02,  3.78114778e-01],
        [-2.91304472e-01,  1.35883613e-01],
        [ 7.53397735e-01,  1.18079264e+00],
        [-3.06633108e-01,  1.51668346e-01],
        [-6.34124686e-02,  3.95080645e-01]]), 'qstat': array([6.85826223e+00, 3.
↪77273016e+01, 3.85717521e+02, 7.32068234e+02,
        7.62229696e+02, 7.69013606e+02, 1.80006234e+03, 1.80689685e+03,
        1.83749031e+03, 2.18415269e+03, 2.52812962e+03, 2.55775372e+03,
        2.56438508e+03, 3.58858286e+03, 3.59525674e+03, 3.62580196e+03,
        3.97027563e+03, 4.31171925e+03, 4.34115841e+03, 4.34770344e+03,
        5.36559020e+03, 5.37219501e+03, 5.40227661e+03, 5.74441645e+03,
        6.08512548e+03, 6.11448893e+03, 6.12100954e+03, 7.13187321e+03,
        7.13836830e+03, 7.16815021e+03]), 'pvalues': array([8.82323175e-003, 6.
↪42126449e-009, 2.74600191e-083, 3.96377730e-157,
        1.71230216e-162, 7.61823371e-163, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000])},
('J',
 'X',
 'V'): {'acf': array([ 1.        , -0.08971388,  0.17178181, -0.57976408, -0.57751633,
         0.16928219, -0.08856344,  0.99113133, -0.08908178,  0.17049929,
        -0.57609728, -0.57368404,  0.16817819, -0.08774527,  0.9848123 ,
        -0.08839067,  0.16942107, -0.57210998, -0.56969456,  0.16699015,
        -0.08711197,  0.97796277, -0.08810664,  0.16820989, -0.5684689 ,
        -0.56560049,  0.16574469, -0.08629873,  0.97111987, -0.0872756 ,
         0.16688735]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
        [-1.50199679e-01, -2.92280757e-02],
        [ 1.10811128e-01,  2.32752494e-01],
        [-6.42480452e-01, -5.17047707e-01],
        [-6.57471324e-01, -4.97561327e-01],
```

(continues on next page)

(continued from previous page)

```
        [ 7.52969431e-02,  2.63267435e-01],
        [-1.83657644e-01,  6.53076441e-03],
        [ 8.95735843e-01,  1.08652682e+00],
        [-2.16706827e-01,  3.85432747e-02],
        [ 4.26469632e-02,  2.98351626e-01],
        [-7.04778770e-01, -4.47415791e-01],
        [-7.11478716e-01, -4.35889368e-01],
        [ 2.19061278e-02,  3.14450251e-01],
        [-2.34723063e-01,  5.92325199e-02],
        [ 8.37642982e-01,  1.13198161e+00],
        [-2.57964550e-01,  8.11832075e-02],
        [-3.21283670e-04,  3.39163431e-01],
        [-7.42469873e-01, -4.01750086e-01],
        [-7.46944218e-01, -3.92444907e-01],
        [-1.68364179e-02,  3.50816720e-01],
        [-2.71492690e-01,  9.72687454e-02],
        [ 7.93431540e-01,  1.16249400e+00],
        [-2.90714385e-01,  1.14501097e-01],
        [-3.45379763e-02,  3.70957757e-01],
        [-7.71726697e-01, -3.65211110e-01],
        [-7.74594003e-01, -3.56606968e-01],
        [-4.87758122e-02,  3.80265202e-01],
        [-3.01287231e-01,  1.28689779e-01],
        [ 7.56004669e-01,  1.18623508e+00],
        [-3.17872831e-01,  1.43321630e-01],
        [-6.38306951e-02,  3.97605398e-01]]), 'qstat': array([   8.47517754,   39.
↪5778789 ,  394.19603408,  746.40619894,
         776.69703653,  784.99580201, 1825.3545896 , 1833.76689546,
        1864.61294006, 2217.11637211, 2567.00919151, 2597.10784245,
        2605.30897302, 3639.38437728, 3647.72267797, 3678.386011  ,
        4028.38256167, 4375.76630468, 4405.64269257, 4413.78080166,
        5440.45892422, 5448.80013774, 5479.23263921, 5827.14599696,
        6171.8931755 , 6201.52677151, 6209.56829208, 7228.86313254,
        7237.10384927, 7267.26526426]), 'pvalues': array([3.60025244e-003, 2.54549820e-
↪009, 4.00232684e-085, 3.11215093e-160,
        1.27131779e-165, 2.68648911e-166, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000])},
 ('N',
  'X',
  'C'): {'acf': array([ 1.        , -0.01968557,  0.22007368, -0.48359781, -0.48169266,
         0.217541  , -0.01947545,  0.98802832, -0.02016042,  0.21825466,
        -0.48114571, -0.47876755,  0.21525037, -0.02022865,  0.98066016,
        -0.02062928,  0.21593687, -0.47865674, -0.47685675,  0.21294954,
        -0.02105458,  0.97360499, -0.0210813 ,  0.21410622, -0.47613525,
        -0.47476485,  0.21066211, -0.02169228,  0.96607903, -0.02189333,
         0.21159558]), 'confidence_intervals': array([[ 1.        ,  1.        ],
        [-0.08017137,  0.04080023],
```

(continues on next page)

(continued from previous page)

```
        [ 0.15956444,  0.28058291],
        [-0.54696776, -0.42022786],
        [-0.5573694 , -0.40601591],
        [ 0.13137411,  0.30370788],
        [-0.10762875,  0.06867786],
        [ 0.89985927,  1.07619737],
        [-0.14229436,  0.10197352],
        [ 0.09610855,  0.34040077],
        [-0.60471035, -0.35758106],
        [-0.6090063 , -0.34852881],
        [ 0.07872441,  0.35177632],
        [-0.1579906 ,  0.11753331],
        [ 0.84288734,  1.11843299],
        [-0.18193064,  0.14067209],
        [ 0.05462586,  0.37724789],
        [-0.64102185, -0.31629163],
        [-0.64430483, -0.30940867],
        [ 0.04060482,  0.38529426],
        [-0.19435926,  0.1522501 ],
        [ 0.80029094,  1.14691903],
        [-0.21336663,  0.17120404],
        [ 0.02181242,  0.40640001],
        [-0.66929924, -0.28297126],
        [-0.67217595, -0.27735376],
        [ 0.00911703,  0.4122072 ],
        [-0.22404133,  0.18065678],
        [ 0.76372146,  1.16843659],
        [-0.24047436,  0.19668769],
        [-0.00699346,  0.43018463]]), 'qstat': array([4.08061309e-01, 5.14562056e+01, 2.
↪98189072e+02, 5.43215771e+02,
        5.93238914e+02, 5.93640224e+02, 1.62749495e+03, 1.62792581e+03,
        1.67847117e+03, 1.92435215e+03, 2.16804283e+03, 2.21734834e+03,
        2.21778421e+03, 3.24315833e+03, 3.24361252e+03, 3.29342499e+03,
        3.53841766e+03, 3.78180680e+03, 3.83039153e+03, 3.83086693e+03,
        4.84841572e+03, 4.84889326e+03, 4.89819851e+03, 5.14227073e+03,
        5.38517676e+03, 5.43304836e+03, 5.43355645e+03, 6.44229694e+03,
        6.44281550e+03, 6.49130169e+03]), 'pvalues': array([5.22955152e-001, 6.
↪70543443e-012, 2.45301824e-064, 3.00422546e-116,
        5.84337991e-126, 5.48966949e-125, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000]),
('O',
 'T',
 'Y'): {'acf': array([ 1.        , -0.08259995,  0.17734596, -0.57067234, -0.5684687 ,
         0.17509455, -0.08130592,  0.99248242, -0.08216322,  0.17599856,
        -0.56701055, -0.56461355,  0.17391106, -0.08060416,  0.98566707,
        -0.08166764,  0.17457252, -0.56337294, -0.56083338,  0.17276644,
        -0.08014475,  0.97890769, -0.08129729,  0.17329419, -0.55960215,
```

(continues on next page)

(continued from previous page)

```
        -0.55701671,  0.17141666, -0.07962553,  0.97213105, -0.08084958,
         0.17208343]), 'confidence_intervals': array([[ 1.        ,  1.        ],
        [-0.14308575, -0.02211415],
        [ 0.11644888,  0.23824304],
        [-0.63343051, -0.50791417],
        [-0.64797665, -0.48896075],
        [ 0.08189545,  0.26829364],
        [-0.17570083,  0.01308899],
        [ 0.89783164,  1.0871332 ],
        [-0.20930973,  0.04498329],
        [ 0.04865795,  0.30333917],
        [-0.695238  , -0.43878309],
        [-0.70170739, -0.4275197 ],
        [ 0.02855867,  0.31926346],
        [-0.22671584,  0.06550753],
        [ 0.8393928 ,  1.13194135],
        [-0.25050182,  0.08716655],
        [ 0.00559388,  0.34355117],
        [-0.73301013, -0.39373576],
        [-0.73718284, -0.38448393],
        [-0.00999187,  0.35552474],
        [-0.2634996 ,  0.10321009],
        [ 0.79542472,  1.16239065],
        [-0.2829843 ,  0.12038972],
        [-0.02851267,  0.37510105],
        [-0.76195271, -0.3572516 ],
        [-0.76495209, -0.34908134],
        [-0.04190791,  0.38474122],
        [-0.29345343,  0.13420237],
        [ 0.7581947 ,  1.18606741],
        [-0.31037882,  0.14867966],
        [-0.05754997,  0.40171684]]), 'qstat': array([7.18437719e+00, 4.03345925e+01, 3.
↪83917858e+02, 7.25178711e+02,
        7.57585350e+02, 7.64579724e+02, 1.80777684e+03, 1.81493320e+03,
        1.84780115e+03, 2.18927226e+03, 2.52818827e+03, 2.56037391e+03,
        2.56729446e+03, 3.60316572e+03, 3.61028383e+03, 3.64284022e+03,
        3.98222838e+03, 4.31888957e+03, 4.35086859e+03, 4.35775699e+03,
        5.38642004e+03, 5.39352177e+03, 5.42582177e+03, 5.76296654e+03,
        6.09732909e+03, 6.12902558e+03, 6.13587154e+03, 7.15729017e+03,
        7.16436205e+03, 7.19643087e+03]), 'pvalues': array([7.35410753e-003, 1.
↪74363074e-009, 6.73724508e-083, 1.23042405e-155,
        1.73027963e-161, 6.91273826e-162, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000])},
 ('Y',
  'V',
  'G'): {'acf': array([ 1.        , -0.06089902,  0.19406527, -0.53915391, -0.53707731,
         0.19151146, -0.06008225,  0.99253338, -0.0607289 ,  0.19239331,
```

(continues on next page)

(continued from previous page)

```
        -0.53581884, -0.53377634,  0.18991977, -0.06011947,  0.98545395,
        -0.0607944 ,  0.19067928, -0.53253409, -0.53052952,  0.18844171,
        -0.06005286,  0.97843607, -0.06083826,  0.18909121, -0.52922907,
        -0.52732228,  0.18674487, -0.06008383,  0.9713386 , -0.06075151,
         0.18737217]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
        [-1.21384817e-01, -4.13213869e-04],
        [ 1.33355563e-01,  2.54774984e-01],
        [-6.02092283e-01, -4.76215529e-01],
        [-6.15104319e-01, -4.59050299e-01],
        [ 1.00964032e-01,  2.82058885e-01],
        [-1.52099649e-01,  3.19351498e-02],
        [ 9.00372563e-01,  1.08469419e+00],
        [-1.86035803e-01,  6.45780064e-02],
        [ 6.69787750e-02,  3.17807845e-01],
        [-6.62308552e-01, -4.09329121e-01],
        [-6.68314035e-01, -3.99238650e-01],
        [ 4.78453231e-02,  3.31994214e-01],
        [-2.03119714e-01,  8.28807826e-02],
        [ 8.42361262e-01,  1.12854664e+00],
        [-2.26870469e-01,  1.05281673e-01],
        [ 2.45218148e-02,  3.56836755e-01],
        [-6.99490201e-01, -3.65577979e-01],
        [-7.03588498e-01, -3.57470552e-01],
        [ 9.53145029e-03,  3.67351967e-01],
        [-2.39687804e-01,  1.19582075e-01],
        [ 7.98727701e-01,  1.15814445e+00],
        [-2.59080530e-01,  1.37404007e-01],
        [-9.21935613e-03,  3.87401771e-01],
        [-7.28198173e-01, -3.30259963e-01],
        [-7.31376425e-01, -3.23268139e-01],
        [-2.22353699e-02,  3.95725109e-01],
        [-2.69673698e-01,  1.49506040e-01],
        [ 7.61685720e-01,  1.18099147e+00],
        [-2.86268629e-01,  1.64765599e-01],
        [-3.82048147e-02,  4.12949146e-01]]), 'qstat': array([3.90526128e+00, 4.
↪36005909e+01, 3.50279471e+02, 6.54891439e+02,
        6.93659875e+02, 6.97479290e+02, 1.74078352e+03, 1.74469309e+03,
        1.78396972e+03, 2.08890498e+03, 2.39181116e+03, 2.43019497e+03,
        2.43404493e+03, 3.46946828e+03, 3.47341278e+03, 3.51225388e+03,
        3.81550300e+03, 4.11676507e+03, 4.15481032e+03, 4.15867786e+03,
        5.18634999e+03, 5.19032708e+03, 5.22878424e+03, 5.53032433e+03,
        5.82998754e+03, 5.86760612e+03, 5.87150414e+03, 6.89125818e+03,
        6.89525113e+03, 6.93327138e+03]), 'pvalues': array([4.81351422e-002, 3.
↪40605757e-010, 1.29766125e-075, 2.03514101e-140,
        1.15368607e-147, 2.14170577e-147, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000, 0.00000000e+000,
        0.00000000e+000, 0.00000000e+000])}}
```

And the partial autocorrelation function values...

[9]:
```
pacf = data_utils.calculate_pacf(nlags=40, alpha=0.05)
pacf
```

[9]:
```
{('F',
  'E',
  'K'): {'pacf': array([ 1.        , -0.08078041,  0.16601981, -0.57034304, -0.9620566 ,
          0.61323735,  0.04746202,  0.77862671,  0.00637416, -0.08100356,
         -0.05871874, -0.13700759,  0.1174423 ,  0.0605576 ,  0.42934315,
          0.09243144, -0.07328271,  0.00398806, -0.100308  ,  0.1303113 ,
          0.05355023,  0.31839881,  0.08861446, -0.22613491,  0.01178529,
         -0.22631973,  0.18613883,  0.15154231,  0.24202115,  0.14823025,
         -0.36025751, -0.00747563, -0.09967852,  0.27377749,  0.2899224 ,
          0.03164718, -0.01778687, -0.72500717,  0.08315487,  0.44580926,
          1.87065584]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
         [-1.41266216e-01, -2.02946130e-02],
         [ 1.05534006e-01,  2.26505610e-01],
         [-6.30828837e-01, -5.09857234e-01],
         [-1.02254240e+00, -9.01570796e-01],
         [ 5.52751546e-01,  6.73723149e-01],
         [-1.30237844e-02,  1.07947819e-01],
         [ 7.18140908e-01,  8.39112511e-01],
         [-5.41116405e-02,  6.68599629e-02],
         [-1.41489364e-01, -2.05177602e-02],
         [-1.19204540e-01,  1.76706309e-03],
         [-1.97493394e-01, -7.65217907e-02],
         [ 5.69564994e-02,  1.77928103e-01],
         [ 7.18002697e-05,  1.21043404e-01],
         [ 3.68857350e-01,  4.89828953e-01],
         [ 3.19456401e-02,  1.52917244e-01],
         [-1.33768515e-01, -1.27969120e-02],
         [-5.64977374e-02,  6.44738661e-02],
         [-1.60793800e-01, -3.98221961e-02],
         [ 6.98254990e-02,  1.90797102e-01],
         [-6.93556812e-03,  1.14036035e-01],
         [ 2.57913009e-01,  3.78884612e-01],
         [ 2.81286603e-02,  1.49100264e-01],
         [-2.86620716e-01, -1.65649112e-01],
         [-4.87005161e-02,  7.22710873e-02],
         [-2.86805530e-01, -1.65833926e-01],
         [ 1.25653024e-01,  2.46624627e-01],
         [ 9.10565039e-02,  2.12028107e-01],
         [ 1.81535344e-01,  3.02506947e-01],
         [ 8.77444507e-02,  2.08716054e-01],
         [-4.20743308e-01, -2.99771705e-01],
         [-6.79614321e-02,  5.30101714e-02],
         [-1.60164320e-01, -3.91927162e-02],
         [ 2.13291689e-01,  3.34263293e-01],
         [ 2.29436600e-01,  3.50408204e-01],
         [-2.88386224e-02,  9.21329811e-02],
         [-7.82726713e-02,  4.26989322e-02],
```

(continues on next page)

(continued from previous page)

```
        [-7.85492971e-01, -6.64521368e-01],
        [ 2.26690715e-02,  1.43640675e-01],
        [ 3.85323463e-01,  5.06295066e-01],
        [ 1.81017004e+00,  1.93114164e+00]])},
 ('J',
  'X',
  'V'): {'pacf': array([ 1.00000000e+00, -8.97994006e-02,  1.65379314e-01, -5.73871649e-
↪01,
        -9.90181691e-01,  2.28389649e-02, -4.93185201e-01,  7.69240477e-01,
         1.03879793e-01, -5.28893124e-01,  1.34794508e-01, -3.51589756e-01,
         1.85686790e-01,  1.55883714e-01,  1.98571228e-01,  3.29974040e-01,
        -7.80295225e-01,  1.35211200e+00,  2.41677843e+00, -9.85720454e-01,
         3.04881070e+01,  1.02424994e+00, -2.40557539e-01, -5.62704983e-01,
         2.76483779e-01, -6.86759801e-02,  6.54149325e-01, -8.11543320e-02,
        -6.08749521e-01, -7.23722874e-01, -2.39649064e+00,  4.45281360e-01,
        -2.59062504e+00,  5.16719035e-01,  7.25642747e-01,  2.43975398e-01,
         3.81543627e-01, -1.01150937e+00, -2.65585463e+01,  9.87381089e-01,
        -2.23305532e+00]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
        [-1.50285202e-01, -2.93135989e-02],
        [ 1.04893512e-01,  2.25865116e-01],
        [-6.34357451e-01, -5.13385847e-01],
        [-1.05066749e+00, -9.29695889e-01],
        [-3.76468369e-02,  8.33247666e-02],
        [-5.53671003e-01, -4.32699399e-01],
        [ 7.08754675e-01,  8.29726278e-01],
        [ 4.33939913e-02,  1.64365595e-01],
        [-5.89378926e-01, -4.68407322e-01],
        [ 7.43087068e-02,  1.95280310e-01],
        [-4.12075557e-01, -2.91103954e-01],
        [ 1.25200988e-01,  2.46172591e-01],
        [ 9.53979124e-02,  2.16369516e-01],
        [ 1.38085427e-01,  2.59057030e-01],
        [ 2.69488238e-01,  3.90459842e-01],
        [-8.40781027e-01, -7.19809423e-01],
        [ 1.29162620e+00,  1.41259780e+00],
        [ 2.35629263e+00,  2.47726423e+00],
        [-1.04620626e+00, -9.25234652e-01],
        [ 3.04276212e+01,  3.05485928e+01],
        [ 9.63764140e-01,  1.08473574e+00],
        [-3.01043341e-01, -1.80071738e-01],
        [-6.23190784e-01, -5.02219181e-01],
        [ 2.15997977e-01,  3.36969581e-01],
        [-1.29161782e-01, -8.19017841e-03],
        [ 5.93663523e-01,  7.14635127e-01],
        [-1.41640134e-01, -2.06685303e-02],
        [-6.69235322e-01, -5.48263719e-01],
        [-7.84208676e-01, -6.63237072e-01],
        [-2.45697644e+00, -2.33600484e+00],
        [ 3.84795558e-01,  5.05767162e-01],
        [-2.65111084e+00, -2.53013923e+00],
        [ 4.56233234e-01,  5.77204837e-01],
```

(continues on next page)

(continued from previous page)

```
        [ 6.65156945e-01,  7.86128549e-01],
        [ 1.83489596e-01,  3.04461199e-01],
        [ 3.21057825e-01,  4.42029428e-01],
        [-1.07199518e+00, -9.51023573e-01],
        [-2.66190321e+01, -2.64980605e+01],
        [ 9.26895288e-01,  1.04786689e+00],
        [-2.29354112e+00, -2.17256952e+00]])},
('N',
 'X',
 'C'): {'pacf': array([ 1.        , -0.01970433,  0.2201909 , -0.50176994, -0.69433673,
         0.95803865,  0.20401309,  0.8161953 , -0.01727898, -0.14639157,
         0.02840954, -0.17105713,  0.16707132, -0.03883989,  0.35872681,
         0.0038246 , -0.25951443,  0.10018443, -0.25434774,  0.14468707,
         0.00653858,  0.20690802,  0.10056146, -0.30360206,  0.22331149,
        -0.39152792,  0.26718222,  0.05444163, -0.14915054,  0.51674794,
        -1.46615051, -2.28765331,  0.89992292, -2.75190377, -1.41158622,
         0.72795062, -0.36145038,  0.05923125, -0.03008689, -0.12517299,
         0.28304207]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
        [-8.01901350e-02,  4.07814684e-02],
        [ 1.59705095e-01,  2.80676698e-01],
        [-5.62255742e-01, -4.41284138e-01],
        [-7.54822534e-01, -6.33850930e-01],
        [ 8.97552847e-01,  1.01852445e+00],
        [ 1.43527288e-01,  2.64498891e-01],
        [ 7.55709496e-01,  8.76681100e-01],
        [-7.77647795e-02,  4.32068239e-02],
        [-2.06877375e-01, -8.59057717e-02],
        [-3.20762615e-02,  8.88953419e-02],
        [-2.31542936e-01, -1.10571333e-01],
        [ 1.06585516e-01,  2.27557120e-01],
        [-9.93256891e-02,  2.16459144e-02],
        [ 2.98241006e-01,  4.19212609e-01],
        [-5.66612031e-02,  6.43104003e-02],
        [-3.20000229e-01, -1.99028626e-01],
        [ 3.96986243e-02,  1.60670228e-01],
        [-3.14833537e-01, -1.93861933e-01],
        [ 8.42012654e-02,  2.05172869e-01],
        [-5.39472258e-02,  6.70243777e-02],
        [ 1.46422222e-01,  2.67393825e-01],
        [ 4.00756561e-02,  1.61047260e-01],
        [-3.64087860e-01, -2.43116257e-01],
        [ 1.62825693e-01,  2.83797296e-01],
        [-4.52013719e-01, -3.31042116e-01],
        [ 2.06696419e-01,  3.27668022e-01],
        [-6.04417343e-03,  1.14927430e-01],
        [-2.09636341e-01, -8.86647377e-02],
        [ 4.56262137e-01,  5.77233740e-01],
        [-1.52663631e+00, -1.40566471e+00],
        [-2.34813911e+00, -2.22716750e+00],
        [ 8.39437119e-01,  9.60408723e-01],
        [-2.81238957e+00, -2.69141797e+00],
```

(continues on next page)

(continued from previous page)

```
        [-1.47207203e+00, -1.35110042e+00],
        [ 6.67464820e-01,  7.88436423e-01],
        [-4.21936183e-01, -3.00964579e-01],
        [-1.25455353e-03,  1.19717050e-01],
        [-9.05726907e-02,  3.03989128e-02],
        [-1.85658788e-01, -6.46871844e-02],
        [ 2.22556271e-01,  3.43527874e-01]])},
('O',
 'T',
 'Y'): {'pacf': array([  1.        ,  -0.08267869,   0.17202456,  -0.56644104,
         -0.9541979 ,   0.85189822,   0.1845516 ,   0.94596428,
          0.43918741,  -1.15664315,  -1.64226152,   2.42653871,
          0.96331553,  -0.31849573,   5.24481561,   0.49895362,
         -0.98231553,   0.75347103,   4.98908903,  -0.80730486,
          2.87884606,   0.7683438 ,   0.94560433, -17.85764882,
         -0.97487154,  -1.64867233,   0.88285951,  -0.51857155,
         -0.28254564,   1.1636752 ,   2.94570193,  -1.51600933,
         -1.10973108,   1.65826343,   0.62080675,   2.96339001,
         -0.37477296,  -0.70521211,   0.6992378 ,  -0.3032168 ,
          0.13773625]), 'confidence_intervals': array([[  1.        ,   1.        ],
        [ -0.14316449,  -0.02219289],
        [  0.11153876,   0.23251036],
        [ -0.62692684,  -0.50595524],
        [ -1.0146837 ,  -0.8937121 ],
        [  0.79141242,   0.91238402],
        [  0.1240658 ,   0.2450374 ],
        [  0.88547848,   1.00645008],
        [  0.37870161,   0.49967321],
        [ -1.21712895,  -1.09615734],
        [ -1.70274732,  -1.58177572],
        [  2.36605291,   2.48702451],
        [  0.90282972,   1.02380133],
        [ -0.37898153,  -0.25800993],
        [  5.1843298 ,   5.30530141],
        [  0.43846782,   0.55943942],
        [ -1.04280133,  -0.92182972],
        [  0.69298523,   0.81395683],
        [  4.92860323,   5.04957483],
        [ -0.86779066,  -0.74681906],
        [  2.81836026,   2.93933186],
        [  0.707858  ,   0.8288296 ],
        [  0.88511853,   1.00609014],
        [-17.91813462, -17.79716302],
        [ -1.03535734,  -0.91438574],
        [ -1.70915813,  -1.58818653],
        [  0.82237371,   0.94334532],
        [ -0.57905735,  -0.45808574],
        [ -0.34303144,  -0.22205984],
        [  1.1031894 ,   1.224161  ],
        [  2.88521613,   3.00618773],
        [ -1.57649513,  -1.45552353],
        [ -1.17021688,  -1.04924528],
```

(continues on next page)

(continued from previous page)

```
        [  1.59777763,   1.71874923],
        [  0.56032094,   0.68129255],
        [  2.90290421,   3.02387581],
        [ -0.43525876,  -0.31428716],
        [ -0.76569792,  -0.64472631],
        [  0.638752  ,   0.7597236 ],
        [ -0.3637026 ,  -0.242731  ],
        [  0.07725045,   0.19822205]])},
 ('Y',
  'V',
  'G'): {'pacf': array([ 1.        , -0.06095707,  0.19143118, -0.54121706, -0.84696723,
         0.95805607,  0.21854914,  0.94681651,  0.16707407, -1.51569922,
        -0.54723384, -1.73908393,  1.37645162,  0.39243366,  1.49812903,
        -0.31122597, -1.20178542, -0.84078145, -8.7457162 ,  1.02860706,
         0.3577604 ,  1.95108875, -0.46472826, -1.10813513, -1.61419444,
         3.30494158,  0.98217571,  1.47580919, -5.93462491, -1.14301777,
        -0.92691166,  2.31809935,  0.57009033,  1.03284862, -5.90888654,
        -0.63850567, -1.82253279, -0.85711637,  1.18090889,  0.52745047,
         2.32443609]), 'confidence_intervals': array([[ 1.00000000e+00,  1.
↪00000000e+00],
        [-1.21442872e-01, -4.71268221e-04],
        [ 1.30945374e-01,  2.51916977e-01],
        [-6.01702859e-01, -4.80731256e-01],
        [-9.07453036e-01, -7.86481432e-01],
        [ 8.97570269e-01,  1.01854187e+00],
        [ 1.58063336e-01,  2.79034939e-01],
        [ 8.86330709e-01,  1.00730231e+00],
        [ 1.06588264e-01,  2.27559867e-01],
        [-1.57618503e+00, -1.45521342e+00],
        [-6.07719647e-01, -4.86748043e-01],
        [-1.79956973e+00, -1.67859812e+00],
        [ 1.31596581e+00,  1.43693742e+00],
        [ 3.31947858e-01,  4.52919462e-01],
        [ 1.43764323e+00,  1.55861483e+00],
        [-3.71711773e-01, -2.50740170e-01],
        [-1.26227123e+00, -1.14129962e+00],
        [-9.01267253e-01, -7.80295649e-01],
        [-8.80620200e+00, -8.68523040e+00],
        [ 9.68121254e-01,  1.08909286e+00],
        [ 2.97274601e-01,  4.18246204e-01],
        [ 1.89060295e+00,  2.01157455e+00],
        [-5.25214057e-01, -4.04242454e-01],
        [-1.16862093e+00, -1.04764933e+00],
        [-1.67468024e+00, -1.55370863e+00],
        [ 3.24445578e+00,  3.36542739e+00],
        [ 9.21689904e-01,  1.04266151e+00],
        [ 1.41532339e+00,  1.53629500e+00],
        [-5.99511071e+00, -5.87413911e+00],
        [-1.20350357e+00, -1.08253197e+00],
        [-9.87397461e-01, -8.66425858e-01],
        [ 2.25761354e+00,  2.37858515e+00],
        [ 5.09604531e-01,  6.30576134e-01],
```

(continues on next page)

(continued from previous page)

```
        [ 9.72362822e-01,  1.09333443e+00],
        [-5.96937234e+00, -5.84840074e+00],
        [-6.98991475e-01, -5.78019871e-01],
        [-1.88301860e+00, -1.76204699e+00],
        [-9.17602175e-01, -7.96630572e-01],
        [ 1.12042309e+00,  1.24139470e+00],
        [ 4.66964672e-01,  5.87936275e-01],
        [ 2.26395029e+00,  2.38492189e+00]])}}
```

Start with a simple ARIMA model with explicitly defined (p, d, q) values.

These values were determined by aggregating all of the series into a single representative series, run through AutoARIMA to determine the p, d, q order terms, and applied to each series individually.

Note that this approach will not work if the series have different seasonality attributes (series 'a' has a weekly seasonality while series 'b' has a monthly seasonality), if the series has a complex seasonality (compounding weekly, day of month, and month of year effects), or if the any of the ordering terms (differencing, order, or moving average order) are significantly different for the different independent series.

[11]:
```
arima_base = ARIMA(order=(4, 1, 5), out_of_sample_size=14)
group_arima = GroupedPmdarima(
    model_template=arima_base
)
```

Fit the grouped Pmdarima model

[12]:
```
group_arima_model = group_arima.fit(df=data.df,
                                    group_key_columns=data.key_columns,
                                    y_col="y",
                                    datetime_col="ds",
                                    silence_warnings=True)
```

Let's see what the parameters were from the run.

[13]:
```
group_arima_model.get_model_params()
```

[13]:
```
  grouping_key_columns key2 key1 key0 maxiter method out_of_sample_size  \
0   (key2, key1, key0)    F    E    K      50  lbfgs                 14
1   (key2, key1, key0)    J    X    V      50  lbfgs                 14
2   (key2, key1, key0)    N    X    C      50  lbfgs                 14
3   (key2, key1, key0)    O    T    Y      50  lbfgs                 14
4   (key2, key1, key0)    Y    V    G      50  lbfgs                 14

  scoring scoring_args start_params suppress_warnings trend with_intercept  p  \
0     mse         None         None              False  None           True  4
1     mse         None         None              False  None           True  4
2     mse         None         None              False  None           True  4
3     mse         None         None              False  None           True  4
4     mse         None         None              False  None           True  4
```

(continues on next page)

(continued from previous page)

```
   d  q  P  D  Q  s
0  1  5  0  0  0  0
1  1  5  0  0  0  0
2  1  5  0  0  0  0
3  1  5  0  0  0  0
4  1  5  0  0  0  0
```

Let's take a look at the training metrics

[14]:
```
group_arima_model.get_metrics()
```

[14]:
```
  grouping_key_columns key2 key1 key0         hqic         aicc         oob  \
0   (key2, key1, key0)    F    E    K  7308.479882  7288.064205   26.639746
1   (key2, key1, key0)    J    X    V  8529.693560  8509.277883   73.661043
2   (key2, key1, key0)    N    X    C  7094.464510  7074.048832   33.927776
3   (key2, key1, key0)    O    T    Y  7636.027821  7615.612144  118.098407
4   (key2, key1, key0)    Y    V    G  8408.200891  8387.785213  177.272914

           bic          aic
0  7342.321388  7287.809869
1  8563.535066  8509.023548
2  7128.306016  7073.794497
3  7669.869327  7615.357809
4  8442.042397  8387.530878
```

The out of bounds measure on the 14 day validation period doesn't look too bad. Let's save this model.

Save it to a local directory

[15]:
```
group_arima_model.save("./group_arima.gpmd")
```

Load the saved model and perform a forecast for each group.

[16]:
```
loaded_arima = GroupedPmdarima.load("./group_arima.gpmd")
```

[17]:
```
forecast = loaded_arima.predict(n_periods = 60, alpha=0.5, predict_col="forecast",␣
↪return_conf_int=True)
forecast
```

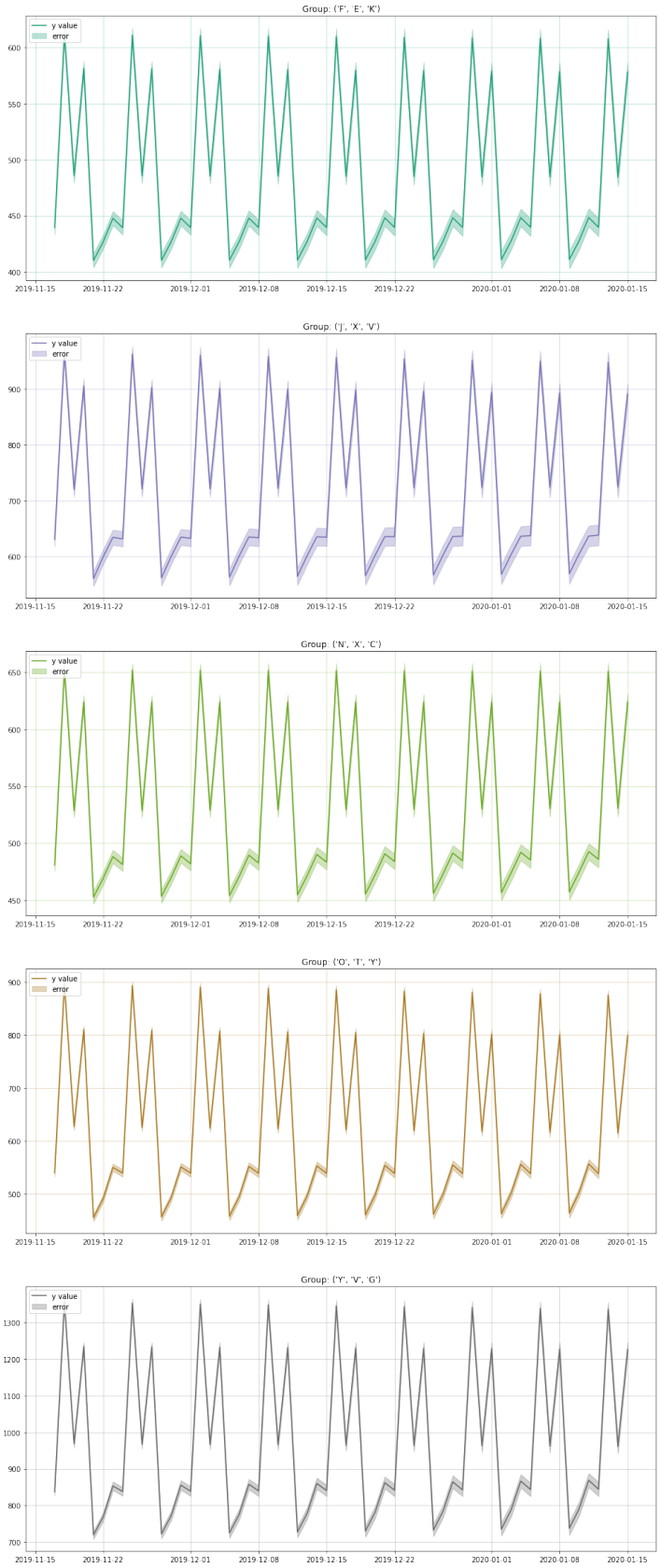
[17]:
```
    grouping_key_columns key2 key1 key0     forecast   yhat_lower  \
0     (key2, key1, key0)    F    E    K   439.455456   433.443671
1     (key2, key1, key0)    F    E    K   611.627138   605.596691
2     (key2, key1, key0)    F    E    K   485.765766   479.734559
3     (key2, key1, key0)    F    E    K   581.559916   575.493820
4     (key2, key1, key0)    F    E    K   410.152148   403.979627
..                   ...  ...  ...  ...          ...          ...
295   (key2, key1, key0)    Y    V    G   868.854019   849.593089
296   (key2, key1, key0)    Y    V    G    844.56244   825.194837
```

(continues on next page)

(continued from previous page)

```
297   (key2, key1, key0)    Y    V    G  1336.358013  1316.951209
298   (key2, key1, key0)    Y    V    G   961.353673   941.932352
299   (key2, key1, key0)    Y    V    G  1226.602318  1207.077806

      yhat_upper                  ds
0     445.467241 2019-11-17 00:01:00
1     617.657586 2019-11-18 00:01:00
2     491.796972 2019-11-19 00:01:00
3     587.626011 2019-11-20 00:01:00
4     416.324670 2019-11-21 00:01:00
..           ...                 ...
295   888.114949 2020-01-11 00:01:00
296   863.930042 2020-01-12 00:01:00
297  1355.764817 2020-01-13 00:01:00
298   980.774994 2020-01-14 00:01:00
299  1246.126830 2020-01-15 00:01:00

[300 rows x 8 columns]
```

[18]:
```
plot_grouped_series_forecast(forecast, data.key_columns, "ds", "forecast", "yhat_lower",
↪"yhat_upper")
```


Group: ('F', 'E', 'K')
y value
error
Group: ('J', 'X', 'V')
y value
error
Group: ('N', 'X', 'C')
y value
error
Group: ('O', 'T', 'Y')
y value
error
Group: ('Y', 'V', 'G')
y value
error
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15

The ordering terms that were used above, `(4, 1, 5)` were determined via using AutoARIMA against the average of all generated series. This approach, a hierarchal optimization of multi series data, is very effective and should be pursued first if at all possible.

3.3.4 AutoARIMA with seasonality components

Since we know that these data sets have a weekly periodicity and that they're generated on a daily basis, let's set m=7. This will apply a seasonality term to the ARIMA model (making it a SARIMA model) and opening up the tuning of the seasonal components (P, D, Q).

Note: This will take **much longer to run**.

[19]:
```
auto_arima = AutoARIMA(out_of_sample_size=14,
                       maxiter=500,
                       max_order=7,
                       d=1,
                       m=7
                      )
auto_arima_model = GroupedPmdarima(model_template=auto_arima).fit(
    df=data.df,
    group_key_columns=data.key_columns,
    y_col="y",
    datetime_col="ds",
    silence_warnings=True)
```

Let's see what the parameters are for this run.

[20]:
```
auto_arima_model.get_model_params()
```

[20]:
```
  grouping_key_columns key2 key1 key0 maxiter method out_of_sample_size  \
0   (key2, key1, key0)    F    E    K     500  lbfgs                 14
1   (key2, key1, key0)    J    X    V     500  lbfgs                 14
2   (key2, key1, key0)    N    X    C     500  lbfgs                 14
3   (key2, key1, key0)    O    T    Y     500  lbfgs                 14
4   (key2, key1, key0)    Y    V    G     500  lbfgs                 14

  scoring scoring_args start_params suppress_warnings trend with_intercept  p  \
0     mse           {}         None               True  None          False  5
1     mse           {}         None               True  None          False  5
2     mse           {}         None               True  None          False  5
3     mse           {}         None               True  None          False  5
4     mse           {}         None               True  None          False  5

   d  q  P  D  Q  s
0  1  0  2  1  0  7
1  1  0  2  1  0  7
2  1  0  2  1  0  7
3  1  0  2  1  0  7
4  1  0  2  1  0  7
```

And the training metrics...

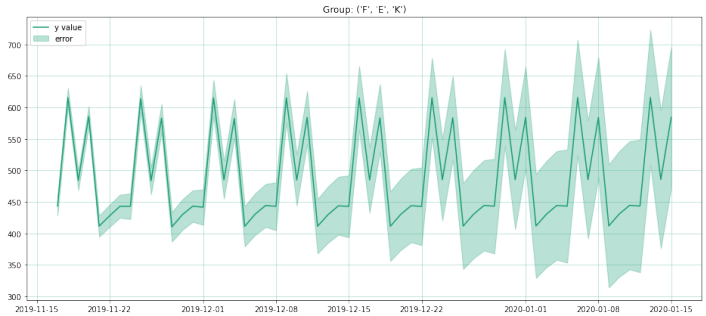
[21]:
```
auto_arima_model.get_metrics()
```

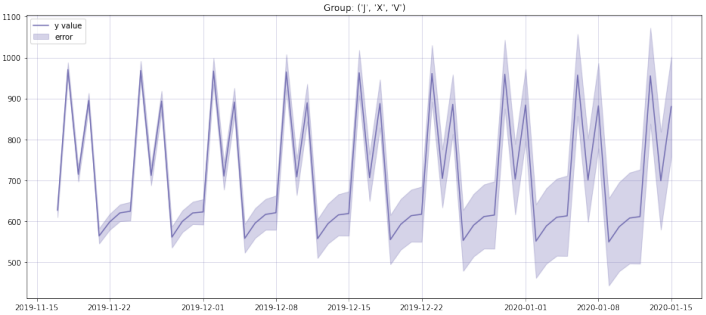
[21]:
```
  grouping_key_columns key2 key1 key0         hqic         aicc         oob  \
0   (key2, key1, key0)    F    E    K  7238.444457  7223.565458   36.179148
1   (key2, key1, key0)    J    X    V  7500.385256  7485.506257  104.947817
2   (key2, key1, key0)    N    X    C  6831.858389  6816.979390   18.105916
3   (key2, key1, key0)    O    T    Y  6355.091450  6340.212451   43.374184
4   (key2, key1, key0)    Y    V    G  6320.132335  6305.253336   23.108154

           bic          aic
0  7263.018307  7223.427129
1  7524.959107  7485.367929
2  6856.432240  6816.841062
3  6379.665300  6340.074122
4  6344.706185  6305.115008
```

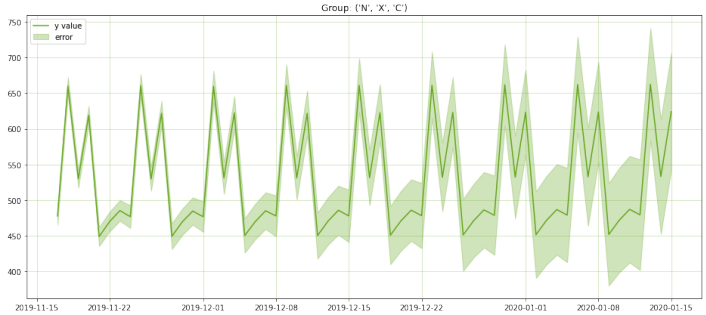
And check the cross validation of each group's model to see what our error metrics are for prediction via backtesting.

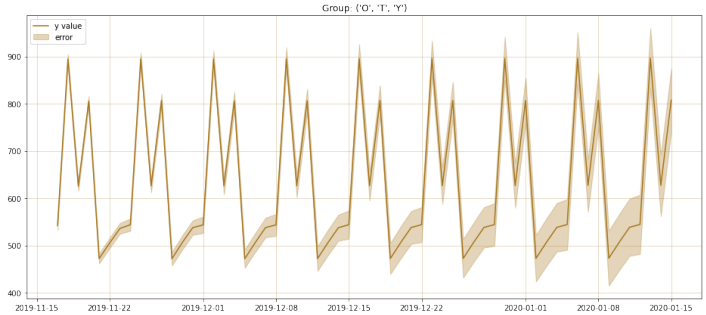
[22]:
```
auto_arima_forecast = auto_arima_model.predict(n_periods = 60, alpha=0.05, return_conf_
↪int=True)
```

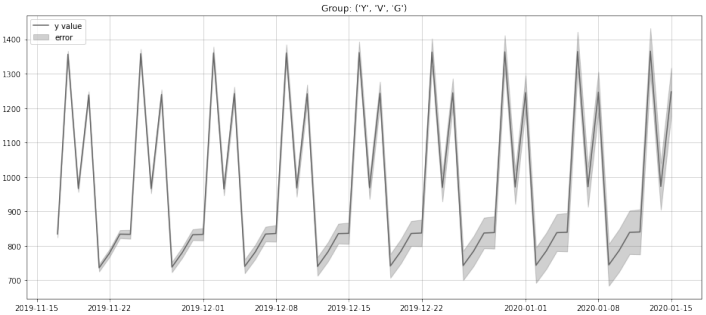
[23]:
```
plot_grouped_series_forecast(auto_arima_forecast, data.key_columns, "ds", "yhat", "yhat_
↪lower", "yhat_upper")
```


Group: ('F', 'E', 'K')
y value
error
700
650
600
550
500
450
400
350
300
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('J', 'X', 'V')
y value
error
1100
1000
900
800
700
600
500
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('N', 'X', 'C')
y value
error
750
700
650
600
550
500
450
400
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('O', 'T', 'Y')
y value
error
900
800
700
600
500
400
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('Y', 'V', 'G')
y value
error
1400
1300
1200
1100
1000
900
800
700
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15

The seasonal components for this example aren't quite as great as the first example. However, this is due to the nature of this generated synthetic example data. For many real-world complex series data, using a seasonal approach with each model getting fit with its own optimal AR terms (p, d, q) and seasonal terms (P, D, Q) can provide better results than manually specifying them.

Cross validation backtesting on the models to get the error metrics

[24]:
```
auto_arima_cv_window = SlidingWindowForecastCV(h=28, step=180, window_size=365)

auto_arima_cv = auto_arima_model.cross_validate(df=data.df,
                                                metrics=["mean_squared_error", "smape",
↪"mean_absolute_error"],
                                                cross_validator=auto_arima_cv_window,
                                                error_score=np.nan,
                                                verbosity=4
                                               )
```

```
[CV] fold=0 ...
fold=0, score=79.808 [time=27.476 sec]
[CV] fold=1 ...
fold=1, score=59.472 [time=41.604 sec]
[CV] fold=2 ...
fold=2, score=232.314 [time=25.329 sec]
[CV] fold=3 ...
fold=3, score=49.415 [time=21.327 sec]
[CV] fold=0 ...
fold=0, score=1.542 [time=26.895 sec]
[CV] fold=1 ...
fold=1, score=1.240 [time=40.395 sec]
[CV] fold=2 ...
fold=2, score=2.711 [time=25.616 sec]
[CV] fold=3 ...
fold=3, score=1.138 [time=21.324 sec]
[CV] fold=0 ...
fold=0, score=7.313 [time=26.776 sec]
[CV] fold=1 ...
fold=1, score=6.078 [time=40.482 sec]
[CV] fold=2 ...
fold=2, score=13.143 [time=25.371 sec]
[CV] fold=3 ...
fold=3, score=5.498 [time=21.812 sec]
[CV] fold=0 ...
fold=0, score=108.753 [time=21.509 sec]
[CV] fold=1 ...
fold=1, score=107.304 [time=24.203 sec]
[CV] fold=2 ...
fold=2, score=142.982 [time=22.971 sec]
[CV] fold=3 ...
fold=3, score=67.764 [time=20.946 sec]
[CV] fold=0 ...
fold=0, score=1.218 [time=21.455 sec]
[CV] fold=1 ...
fold=1, score=1.190 [time=23.443 sec]
```

(continues on next page)

(continued from previous page)

```
[CV] fold=2 ...
fold=2, score=1.365 [time=22.825 sec]
[CV] fold=3 ...
fold=3, score=0.952 [time=21.210 sec]
[CV] fold=0 ...
fold=0, score=8.226 [time=21.380 sec]
[CV] fold=1 ...
fold=1, score=8.362 [time=23.636 sec]
[CV] fold=2 ...
fold=2, score=9.412 [time=22.880 sec]
[CV] fold=3 ...
fold=3, score=6.478 [time=21.000 sec]
[CV] fold=0 ...
fold=0, score=33.936 [time=26.360 sec]
[CV] fold=1 ...
fold=1, score=31.374 [time=26.617 sec]
[CV] fold=2 ...
fold=2, score=31.117 [time=26.668 sec]
[CV] fold=3 ...
fold=3, score=32.456 [time=32.644 sec]
[CV] fold=0 ...
fold=0, score=1.034 [time=2784.458 sec]
[CV] fold=1 ...
fold=1, score=0.923 [time=7289.441 sec]
[CV] fold=2 ...
fold=2, score=0.896 [time=3687.129 sec]
[CV] fold=3 ...
fold=3, score=0.918 [time=7270.761 sec]
[CV] fold=0 ...
fold=0, score=4.857 [time=7284.930 sec]
[CV] fold=1 ...
fold=1, score=4.560 [time=7287.635 sec]
[CV] fold=2 ...
fold=2, score=4.415 [time=3681.372 sec]
[CV] fold=3 ...
fold=3, score=4.672 [time=5103.397 sec]
[CV] fold=0 ...
fold=0, score=22.142 [time=3003.464 sec]
[CV] fold=1 ...
fold=1, score=37.220 [time=907.578 sec]
[CV] fold=2 ...
fold=2, score=36.748 [time=128.197 sec]
[CV] fold=3 ...
fold=3, score=30.669 [time=59.834 sec]
[CV] fold=0 ...
fold=0, score=0.652 [time=22.607 sec]
[CV] fold=1 ...
fold=1, score=0.836 [time=25.936 sec]
[CV] fold=2 ...
fold=2, score=0.827 [time=25.435 sec]
[CV] fold=3 ...
fold=3, score=0.726 [time=21.935 sec]
```

(continues on next page)

(continued from previous page)

```
[CV] fold=0 ...
fold=0, score=4.110 [time=21.966 sec]
[CV] fold=1 ...
fold=1, score=4.930 [time=25.702 sec]
[CV] fold=2 ...
fold=2, score=5.035 [time=27.211 sec]
[CV] fold=3 ...
fold=3, score=4.385 [time=22.635 sec]
[CV] fold=0 ...
fold=0, score=17.119 [time=34.697 sec]
[CV] fold=1 ...
fold=1, score=107.078 [time=17.232 sec]
[CV] fold=2 ...
fold=2, score=19.922 [time=26.995 sec]
[CV] fold=3 ...
fold=3, score=16.925 [time=24.339 sec]
[CV] fold=0 ...
fold=0, score=0.396 [time=36.418 sec]
[CV] fold=1 ...
fold=1, score=1.083 [time=18.280 sec]
[CV] fold=2 ...
fold=2, score=0.419 [time=28.167 sec]
[CV] fold=3 ...
fold=3, score=0.383 [time=23.993 sec]
[CV] fold=0 ...
fold=0, score=3.361 [time=36.009 sec]
[CV] fold=1 ...
fold=1, score=9.168 [time=18.480 sec]
[CV] fold=2 ...
fold=2, score=3.774 [time=27.421 sec]
[CV] fold=3 ...
fold=3, score=3.246 [time=23.926 sec]
```

[25]:
```
auto_arima_cv
```

[25]:
```
  grouping_key_columns key2 key1 key0  mean_squared_error_mean  \
0   (key2, key1, key0)    F    E    K               105.252090
1   (key2, key1, key0)    J    X    V               106.700806
2   (key2, key1, key0)    N    X    C                32.220902
3   (key2, key1, key0)    O    T    Y                31.694749
4   (key2, key1, key0)    Y    V    G                40.261072

   mean_squared_error_stddev  smape_mean  smape_stddev  \
0                  74.171681    1.657663      0.626170
1                  26.631834    1.181143      0.148013
2                   1.110630    0.942818      0.053464
3                   6.090290    0.760451      0.076121
4                  38.594749    0.570198      0.296271

   mean_absolute_error_mean  mean_absolute_error_stddev
0                  8.007901                    3.036208
1                  8.119500                    1.052836
```

(continues on next page)

(continued from previous page)

```
2                  4.625816                    0.161250
3                  4.615029                    0.381769
4                  4.887450                    2.479422
```

3.3.5 AutoARIMA without seasonality components

[26]:
```
auto_arima_no_seasonal = AutoARIMA(out_of_sample_size=14, maxiter=500, d=1, max_
↪order=14) # leaving the 'm' arg out.

auto_arima_no_seasonal_obj = GroupedPmdarima(model_template=auto_arima_no_seasonal)

auto_arima_model_no_seasonal = auto_arima_no_seasonal_obj.fit(df=data.df,
                                                              group_key_columns=data.key_
↪columns,
                                                              y_col="y",
                                                              datetime_col="ds",
                                                              silence_warnings=True
                                                             )
```

[27]:
```
auto_arima_model_no_seasonal.get_model_params()
```

[27]:
```
  grouping_key_columns key2 key1 key0 maxiter method out_of_sample_size  \
0   (key2, key1, key0)    F    E    K     500  lbfgs                 14
1   (key2, key1, key0)    J    X    V     500  lbfgs                 14
2   (key2, key1, key0)    N    X    C     500  lbfgs                 14
3   (key2, key1, key0)    O    T    Y     500  lbfgs                 14
4   (key2, key1, key0)    Y    V    G     500  lbfgs                 14

  scoring scoring_args start_params suppress_warnings trend with_intercept  p  \
0     mse           {}         None               True  None          False  2
1     mse           {}         None               True  None          False  2
2     mse           {}         None               True  None           True  4
3     mse           {}         None               True  None          False  4
4     mse           {}         None               True  None          False  5

   d  q  P  D  Q  s
0  1  2  0  0  0  0
1  1  0  0  0  0  0
2  1  3  0  0  0  0
3  1  5  0  0  0  0
4  1  4  0  0  0  0
```

[28]:
```
auto_arima_model_no_seasonal.get_metrics()
```

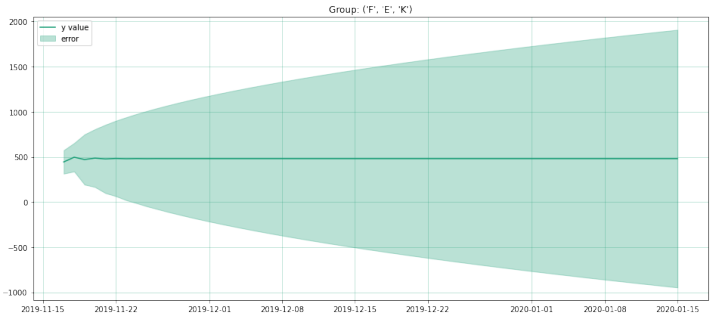
[28]:
```
  grouping_key_columns key2 key1 key0          hqic          aicc  \
0   (key2, key1, key0)    F    E    K  11803.021777  11793.683788
1   (key2, key1, key0)    J    X    V  13775.437281  13769.822949
2   (key2, key1, key0)    N    X    C   7591.362347   7574.623595
3   (key2, key1, key0)    O    T    Y   7181.798445   7163.219266
4   (key2, key1, key0)    Y    V    G   8291.866077   8273.286898
```

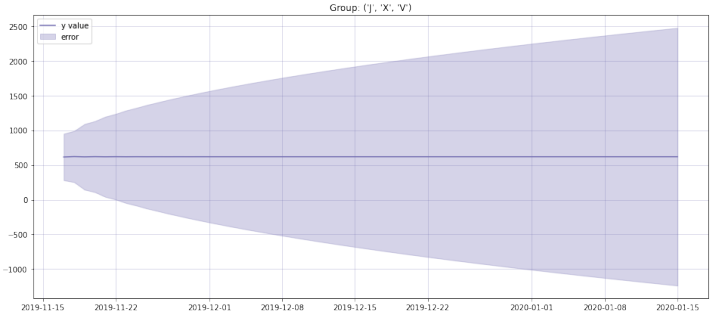
(continues on next page)

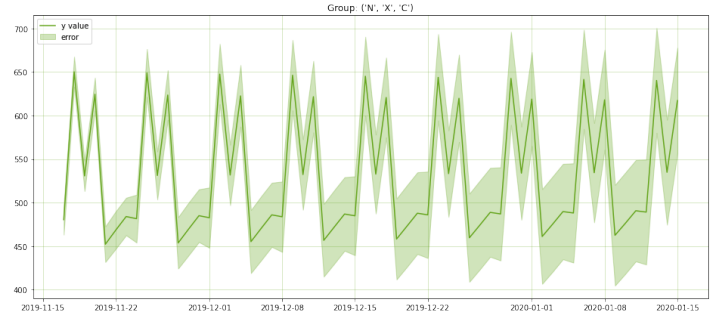
(continued from previous page)

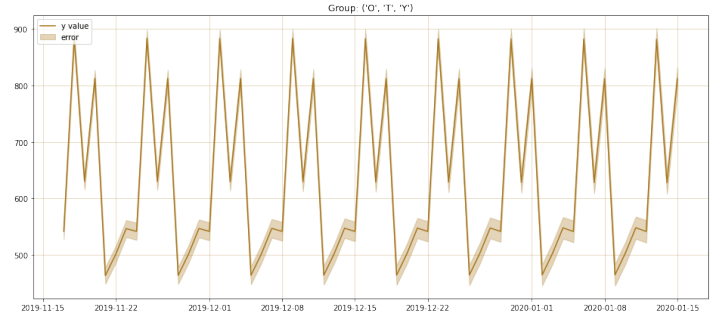
```
            oob           bic           aic
0   4803.128966  11818.404280  11793.626317
1  32134.899126  13784.666782  13769.800005
2     85.834726   7619.050852   7574.450519
3     51.496175   7212.563450   7163.007524
4    108.699561   8322.631082   8273.075156
```

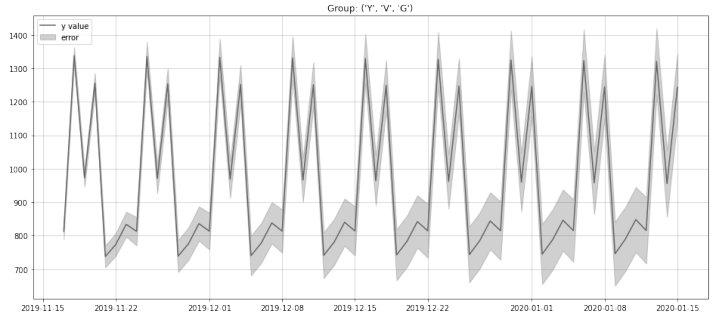
[29]:
```
auto_arima_forecast_no_seasonal = auto_arima_model_no_seasonal.predict(n_periods = 60,
                                                                       alpha=0.05,
                                                                       return_conf_
↪int=True
                                                                      )
plot_grouped_series_forecast(auto_arima_forecast_no_seasonal,
                             data.key_columns,
                             "ds",
                             "yhat",
                             "yhat_lower",
                             "yhat_upper"
                            )
```


Group: ('F', 'E', 'K')
y value
error
2000
1500
1000
500
0
−500
−1000
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('J', 'X', 'V')
y value
error
2500
2000
1500
1000
500
0
−500
−1000
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('N', 'X', 'C')
y value
error
700
650
600
550
500
450
400
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('O', 'I', 'Y')
y value
error
900
800
700
600
500
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('Y', 'V', 'G')
y value
error
1400
1300
1200
1100
1000
900
800
700
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15

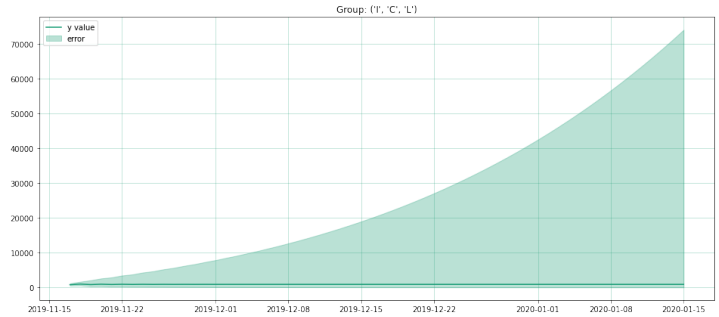
These are the results when allowing AutoARIMA to optimize without specifying the seasonal 'm' value. They're definitely not as great since the generated data has a clear weekly seasonality component and the optimizer will struggle to find appropriate ordering terms for this type of data.

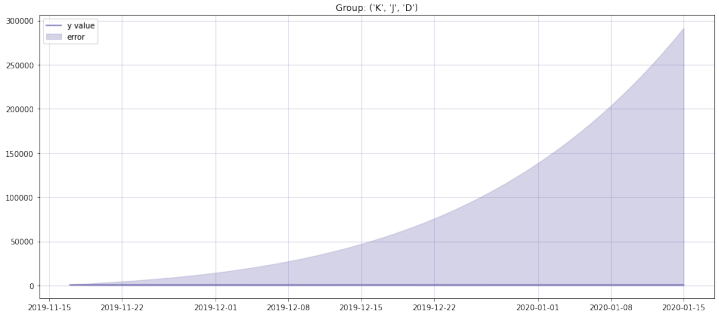
3.3.6 Pipeline orchestration with data preprocessing

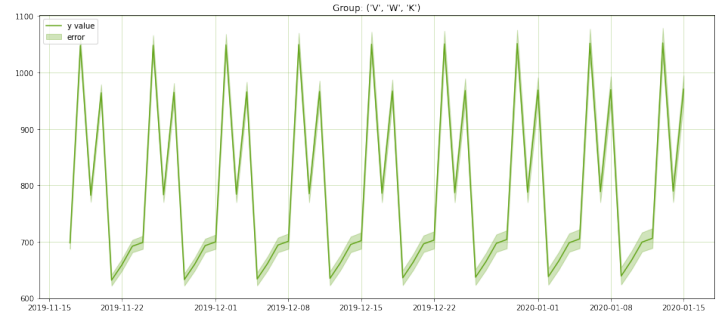
[78]:
```
pipeline_obj = Pipeline(
    steps=[
        (
            "log",
            LogEndogTransformer(lmbda=0.2, neg_action="raise", floor=1e-12),
        ),
        ("arima", AutoARIMA(out_of_sample_size=14, max_order=14, d=1, suppress_
↪warnings=True)),
    ]
)

pipeline_arima = GroupedPmdarima(
    y_col="y", datetime_col="ds", model_template=pipeline_obj
).fit(df=data.df, group_key_columns=data.key_columns, silence_warnings=True)
```

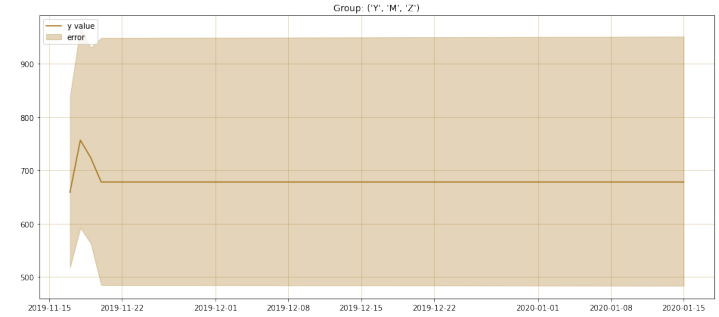
[79]:
```
pipeline_forecast = pipeline_arima.predict(n_periods = 60,
                                           alpha=0.05,
                                           return_conf_int=True
                                          )
```

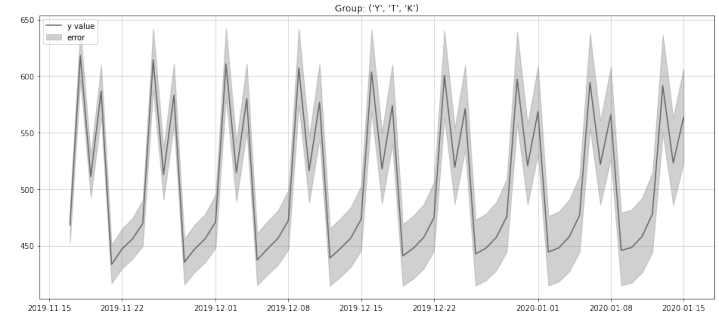
[89]:
```
plot_grouped_series_forecast(pipeline_forecast,
                             data.key_columns,
                             "ds",
                             "forecast",
                             "yhat_lower",
                             "yhat_upper"
                            )
```


Group: ('I', 'C', 'L')
y value
error
70000
60000
50000
40000
30000
20000
10000
0
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('K', 'J', 'D')
y value
error
300000
250000
200000
150000
100000
50000
0
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('V', 'W', 'K')
y value
error
1100
1000
900
800
700
600
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('Y', 'M', 'Z')
y value
error
900
800
700
600
500
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15


Group: ('Y', 'T', 'K')
y value
error
650
600
550
500
450
2019-11-15
2019-11-22
2019-12-01
2019-12-08
2019-12-15
2019-12-22
2020-01-01
2020-01-08
2020-01-15

These series are definitely not in need of log transformation to build effective ARIMA models. That being said, your data may benefit from having an endogenous transformation applied to enforce stationarity.

3.4 GroupedPmdarima Example Scripts

The scripts included below show the various options available for utilizing the `GroupedPmdarima` API. For an alternative view of these examples with data visualizations, see the *notebooks here*

Scripts

- *GroupedPmdarima ARIMA*
- *GroupedPmdarima AutoARIMA*
- *GroupedPmdarima Pipeline Example*
- *GroupedPmdarima Group Subset Prediction Example*
- *GroupedPmdarima Series Analysis Example*
- *GroupedPmdarima Differencing Term Manual Calculation Example*
- *Supplementary*

3.4.1 GroupedPmdarima ARIMA

This example shows using a manually-configured (order values provided for a non-seasonal collection of series) ARIMA model that is applied to each group.

Using this approach (a static order configuration) can be useful for homogenous collections of series. If each member of the grouped collection of series shares a common characteristic in the residuals (i.e., the differencing terms for both an auto-correlation and partial auto-correlation analysis shows similar relationships for all groups), this approach will be faster and less expensive to fit a model than any other means.

Listing 4: GroupedPmdarima manually configured ARIMA model

```
import numpy as np
from pmdarima.arima.arima import ARIMA
from pmdarima.model_selection import SlidingWindowForecastCV
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedPmdarima


def get_and_print_model_metrics_params(grouped_model):
    fit_metrics = grouped_model.get_metrics()
    fit_params = grouped_model.get_model_params()

    print(f"\nModel Fit Metrics:\n{fit_metrics.to_string()}")
    print(f"\nModel Fit Params:\n{fit_params.to_string()}")


if __name__ == "__main__":

```

(continues on next page)

(continued from previous page)

```
    # Generate a few years of daily data across 4 different groups, defined by 3 columns␣
↪that
    # define each group
    generated_data = generate_example_data(
        column_count=3,
        series_count=4,
        series_size=365 * 4,
        start_dt="2019-01-01",
        days_period=1,
    )

    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

    # Build a GroupedPmdarima model by specifying an ARIMA model
    arima_obj = ARIMA(order=(2, 1, 3), out_of_sample_size=60)
    base_arima = GroupedPmdarima(model_template=arima_obj).fit(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
        silence_warnings=True,
    )

    # Save to local directory
    save_dir = "/tmp/group_pmdarima/arima.gpmd"
    base_arima.save(save_dir)

    # Load from saved model
    loaded_model = GroupedPmdarima.load(save_dir)

    print("\nARIMA results:\n", "-" * 40)
    get_and_print_model_metrics_params(loaded_model)

    prediction = loaded_model.predict(
        n_periods=30, alpha=0.02, predict_col="forecast", return_conf_int=True
    )
    print("\nPredictions:\n", "-" * 40)
    print(prediction.to_string())

    print("\nCross validation metric results:\n", "-" * 40)
    cross_validator = SlidingWindowForecastCV(h=90, step=365, window_size=730)
    cv_results = loaded_model.cross_validate(
        df=training_data,
        metrics=["mean_squared_error", "smape", "mean_absolute_error"],
        cross_validator=cross_validator,
        error_score=np.nan,
        verbosity=4,
    )

    print(cv_results.to_string())
```

3.4.2 GroupedPmdarima AutoARIMA

For projects that do not have homogeneous relationships amongst groups of series, using the AutoARIMA functionality of pmdarima is advised. This will allow for individualized optimation of the order terms (p, d, q) and, for seasonal series, the (P, D, Q) seasonal order terms as well.

Note: If using a seasonal approach, the parameter `m` must be set to an integer value that represents the seasonal periodicity. In this mode, with `m` set, the ARIMA terms (p, d, q) will be optimized along with (P, D, Q). Due to the complexity of optimizing these terms, this execution mode will take far longer than an optimization of a non-seasonal model.

Listing 5: GroupedPmdarima non-seasonal AutoARIMA model

```
import numpy as np
from pmdarima.arima.auto import AutoARIMA
from pmdarima.model_selection import SlidingWindowForecastCV
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedPmdarima


def get_and_print_model_metrics_params(grouped_model):
    fit_metrics = grouped_model.get_metrics()
    fit_params = grouped_model.get_model_params()

    print(f"\nModel Fit Metrics:\n{fit_metrics.to_string()}")
    print(f"\nModel Fit Params:\n{fit_params.to_string()}")


if __name__ == "__main__":

    # Generate a few years of daily data across 4 different groups, defined by 3 columns␣
↪that
    # define each group
    generated_data = generate_example_data(
        column_count=3,
        series_count=4,
        series_size=365 * 3,
        start_dt="2019-01-01",
        days_period=1,
    )

    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

    # Utilize pmdarima's AutoARIMA to auto-tune the ARIMA order values
    auto_arima_obj = AutoARIMA(out_of_sample_size=60, maxiter=100)
    base_auto_arima = GroupedPmdarima(model_template=auto_arima_obj).fit(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
        silence_warnings=True,
```

(continues on next page)

(continued from previous page)

```
    )

    # Save to local directory
    save_dir = "/tmp/group_pmdarima/autoarima.gpmd"
    base_auto_arima.save(save_dir)

    # Load from saved model
    loaded_model = GroupedPmdarima.load(save_dir)

    print("\nAutoARIMA results:\n", "-" * 40)
    get_and_print_model_metrics_params(loaded_model)

    print("\nPredictions:\n", "-" * 40)
    prediction = loaded_model.predict(n_periods=30, alpha=0.1, return_conf_int=True)
    print(prediction.to_string())

    print("\nCross validation metric results:\n", "-" * 40)
    cross_validator = SlidingWindowForecastCV(h=30, step=180, window_size=365)
    cv_results = loaded_model.cross_validate(
        df=training_data,
        metrics=["mean_squared_error", "smape", "mean_absolute_error"],
        cross_validator=cross_validator,
        error_score=np.nan,
        verbosity=3,
    )

    print(cv_results.to_string())
```

3.4.3 GroupedPmdarima Pipeline Example

This example shows the utilization of a `pmdarima.pipeline.Pipeline`, incorporating preprocessing operations to each series. In the example below, a Box Cox transformation is applied to each series to force stationarity.

Note: The data set used for these examples is a randomly generated non-deterministic group of series data. As such, The relevance of utilizing a normalcy transform on this data is somewhere between 'unlikely' and 'zero'. Using a BoxCox transform here is used as an API example only.

Listing 6: GroupedPmdarima with Pipeline model

```
import numpy as np
from pmdarima.arima.auto import AutoARIMA
from pmdarima.pipeline import Pipeline
from pmdarima.preprocessing import BoxCoxEndogTransformer
from pmdarima.model_selection import RollingForecastCV
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedPmdarima


def get_and_print_model_metrics_params(grouped_model):
```

(continues on next page)

(continued from previous page)

```
    fit_metrics = grouped_model.get_metrics()
    fit_params = grouped_model.get_model_params()

    print(f"\nModel Fit Metrics:\n{fit_metrics.to_string()}")
    print(f"\nModel Fit Params:\n{fit_params.to_string()}")


if __name__ == "__main__":

    # Generate a few years of daily data across 2 different groups, defined by 3 columns␣
↪that
    # define each group
    generated_data = generate_example_data(
        column_count=3,
        series_count=2,
        series_size=365 * 3,
        start_dt="2019-01-01",
        days_period=1,
    )

    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

    pipeline_obj = Pipeline(
        steps=[
            (
                "box",
                BoxCoxEndogTransformer(lmbda2=0.4, neg_action="ignore", floor=1e-12),
            ),
            ("arima", AutoARIMA(out_of_sample_size=60, max_p=4, max_q=4, max_d=4)),
        ]
    )
    pipeline_arima = GroupedPmdarima(model_template=pipeline_obj).fit(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
        silence_warnings=True,
    )

    # Save to local directory
    save_dir = "/tmp/group_pmdarima/pipeline.gpmd"
    pipeline_arima.save(save_dir)

    # Load from saved model
    loaded_model = GroupedPmdarima.load(save_dir)

    print("\nPipeline AutoARIMA results:\n", "-" * 40)
    get_and_print_model_metrics_params(loaded_model)

    print("\nPredictions:\n", "-" * 40)
    prediction = loaded_model.predict(
```

(continues on next page)

(continued from previous page)

```
        n_periods=30, alpha=0.2, predict_col="predictions", return_conf_int=True
    )
    print(prediction.to_string())

    print("\nCross validation metric results:\n", "-" * 40)
    cross_validator = RollingForecastCV(h=30, step=365, initial=730)
    cv_results = loaded_model.cross_validate(
        df=training_data,
        metrics=["mean_squared_error"],
        cross_validator=cross_validator,
        error_score=np.nan,
        verbosity=3,
    )

    print(cv_results.to_string())
```

3.4.4 GroupedPmdarima Group Subset Prediction Example

This example shows a subset prediction of groups by using the *predict_groups <diviner.GroupedPmdarima.predict_groups>* method.

Listing 7: GroupedPmdarima Subset Groups Prediction

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima
from diviner.utils.example_utils.example_data_generator import generate_example_data

if __name__ == "__main__":

    generated_data = generate_example_data(
        column_count=2,
        series_count=6,
        series_size=365 * 4,
        start_dt="2019-01-01",
        days_period=1,
    )

    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

    arima_obj = ARIMA(order=(2, 1, 3), out_of_sample_size=60)
    base_arima = GroupedPmdarima(model_template=arima_obj).fit(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
        silence_warnings=True,
    )

    # Get a subset of group keys to generate forecasts for
    group_df = training_data.copy()
```

(continues on next page)

(continued from previous page)

```
    group_df["groups"] = list(zip(*[group_df[c] for c in group_key_columns]))
    distinct_groups = group_df["groups"].unique()
    groups_to_predict = list(distinct_groups[:3])

    print("-" * 65)
    print(f"Unique groups that have been modeled: {distinct_groups}")
    print(f"Subset of groups to generate predictions for: {groups_to_predict}")
    print("-" * 65)

    forecasts = base_arima.predict_groups(
        groups=groups_to_predict,
        n_periods=60,
        predict_col="forecast_values",
        on_error="warn",
    )

    print(f"\nForecast values:\n{forecasts.to_string()}")
```

3.4.5 GroupedPmdarima Series Analysis Example

The below script illustrates how to perform analytics on a grouped series data set. Applying the results of these utilities can aid in determining appropriate order values (p, d, q) and seasonal order values (P, D, Q) for the example shown in *the ARIMA example*.

Listing 8: GroupedPmdarima series exploration and analysis

```
import pprint
from diviner import PmdarimaAnalyzer

from diviner.utils.example_utils.example_data_generator import generate_example_data


def _print_dict(data, name):
    print("\n" + "-" * 100)
    print(f"{name} values for the groups")
    print("-" * 100, "\n")
    pprint.PrettyPrinter(indent=2).pprint(data)


if __name__ == "__main__":

    generated_data = generate_example_data(
        column_count=4,
        series_count=3,
        series_size=365 * 12,
        start_dt="2010-01-01",
        days_period=1,
    )
    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

```

(continues on next page)

(continued from previous page)

```
    # Create a utility object for performing analyses
    # We reuse this object because the grouped data set collection is lazily evaluated and can be
    # reused for subsequent analytics operations on the data set.
    analyzer = PmdarimaAnalyzer(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
    )

    # Decompose the trends of each group
    decomposed_trends = analyzer.decompose_groups(m=7, type_="additive")

    print("Decomposed trend data for the groups")
    print("-" * 100, "\n")
    print(decomposed_trends[:50].to_string())

    # Calculate optimal differencing for ARMA terms
    ndiffs = analyzer.calculate_ndiffs(alpha=0.1, test="kpss", max_d=5)

    _print_dict(ndiffs, "Differencing")

    # Calculate seasonal differencing
    nsdiffs = analyzer.calculate_nsdiffs(m=365, test="ocsb", max_D=5)

    _print_dict(nsdiffs, "Seasonal Differencing")

    # Get the autocorrelation function for each group
    group_acf = analyzer.calculate_acf(
        unbiased=True, nlags=120, qstat=True, fft=True, alpha=0.05, adjusted=True
    )

    _print_dict(group_acf, "Autocorrelation function")

    # Get the partial autocorrelation function for each group
    group_pacf = analyzer.calculate_pacf(nlags=120, method="yw", alpha=0.05)

    _print_dict(group_pacf, "Partial Autocorrelation function")

    # Perform a diff operation on each group
    group_diff = analyzer.generate_diff(lag=7, differences=1)

    _print_dict(group_diff, "Differencing")

    # Invert the diff operation on each group
    group_diff_inv = analyzer.generate_diff_inversion(
        group_diff, lag=7, differences=1, recenter=True
    )

    _print_dict(group_diff_inv, "Differencing Inversion")
```

3.4.6 GroupedPmdarima Differencing Term Manual Calculation Example

This script below shows a means of dramatically reducing the optimization time of AutoARIMA through the manual calculation of the differencing term `'d'` for each series in the grouped series data set. By manually setting this argument (which can be either unique for each group or homogenous across all groups), the optimization algorithm can reduce the total number of iterative validation tests.

Listing 9: GroupedPmdarima manual differencing term extraction and application to AutoARIMA

```
from diviner.utils.example_utils.example_data_generator import generate_example_data
from diviner import GroupedPmdarima, PmdarimaAnalyzer
from pmdarima.pipeline import Pipeline
from pmdarima import AutoARIMA
from pmdarima.model_selection import SlidingWindowForecastCV


def get_and_print_model_metrics_params(grouped_model):
    fit_metrics = grouped_model.get_metrics()
    fit_params = grouped_model.get_model_params()

    print(f"\nModel Fit Metrics:\n{fit_metrics.to_string()}")
    print(f"\nModel Fit Params:\n{fit_params.to_string()}")


if __name__ == "__main__":

    # Generate 6 years of daily data across 4 different groups, defined by 3 columns that
    # define each group
    generated_data = generate_example_data(
        column_count=3,
        series_count=3,
        series_size=365 * 3,
        start_dt="2019-01-01",
        days_period=1,
    )

    training_data = generated_data.df
    group_key_columns = generated_data.key_columns

    pipeline = Pipeline(
        steps=[
            (
                "arima",
                AutoARIMA(
                    max_order=14,
                    out_of_sample_size=90,
                    suppress_warnings=True,
                    error_action="ignore",
                ),
            )
        ]
    )
```

(continues on next page)

(continued from previous page)

```

    diff_analyzer = PmdarimaAnalyzer(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
    )
    ndiff = diff_analyzer.calculate_ndiffs(
        alpha=0.05,
        test="kpss",
        max_d=4,
    )

    grouped_model = GroupedPmdarima(model_template=pipeline).fit(
        df=training_data,
        group_key_columns=group_key_columns,
        y_col="y",
        datetime_col="ds",
        ndiffs=ndiff,
        silence_warnings=True,
    )

    # Save to local directory
    save_dir = "/tmp/group_pmdarima/pipeline_override.gpmd"
    grouped_model.save(save_dir)

    # Load from saved model
    loaded_model = GroupedPmdarima.load(save_dir)

    print("\nAutoARIMA results:\n", "-" * 40)
    get_and_print_model_metrics_params(loaded_model)

    print("\nPredictions:\n", "-" * 40)
    prediction = loaded_model.predict(
        n_periods=30, alpha=0.1, predict_col="forecasted_values", return_conf_int=True
    )
    print(prediction.to_string())

    cv_evaluator = SlidingWindowForecastCV(h=90, step=120, window_size=180)
    cross_validation = loaded_model.cross_validate(
        df=training_data,
        metrics=["smape", "mean_squared_error", "mean_absolute_error"],
        cross_validator=cv_evaluator,
    )

    print("\nCross validation metrics:\n", "-" * 40)
    print(cross_validation.to_string())
```

3.4.7 Supplementary

Note: To run these examples for yourself with the data generator example, utlize the following code:

Listing 10: Synthetic Data Generator

```
import itertools
import pandas as pd
import numpy as np
import string
import random
from datetime import timedelta, datetime
from collections import namedtuple


def _generate_time_series(series_size: int):
    residuals = np.random.lognormal(
        mean=np.random.uniform(low=0.5, high=3.0),
        sigma=np.random.uniform(low=0.6, high=0.98),
        size=series_size,
    )
    trend = [
        np.polyval([23.0, 1.0, 5], x)
        for x in np.linspace(start=0, stop=np.random.randint(low=0, high=4), num=series_
↪size)
    ]
    seasonality = [
        90 * np.sin(2 * np.pi * 1000 * (i / (series_size * 200))) + 40
        for i in np.arange(0, series_size)
    ]

    return residuals + trend + seasonality + np.random.uniform(low=20.0, high=1000.0)


def _generate_grouping_columns(column_count: int, series_count: int):
    candidate_list = list(string.ascii_uppercase)
    candidates = random.sample(
        list(itertools.permutations(candidate_list, column_count)), series_count
    )
    column_names = sorted([f"key{x}" for x in range(column_count)], reverse=True)
    return [dict(zip(column_names, entries)) for entries in candidates]


def _generate_raw_df(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int,
):
    candidates = _generate_grouping_columns(column_count, series_count)
```

(continues on next page)

(continued from previous page)

```
    start_date = datetime.strptime(start_dt, "%Y-%M-%d")
    dates = np.arange(
        start_date,
        start_date + timedelta(days=series_size * days_period),
        timedelta(days=days_period),
    )
    df_collection = []
    for entry in candidates:
        generated_series = _generate_time_series(series_size)
        series_dict = {"ds": dates, "y": generated_series}
        series_df = pd.DataFrame.from_dict(series_dict)
        for column, value in entry.items():
            series_df[column] = value
        df_collection.append(series_df)
    return pd.concat(df_collection)


def generate_example_data(
    column_count: int,
    series_count: int,
    series_size: int,
    start_dt: str,
    days_period: int = 1,
):

    Structure = namedtuple("Structure", "df key_columns")
    data = _generate_raw_df(column_count, series_count, series_size, start_dt, days_
↪period)
    key_columns = list(data.columns)

    for key in ["ds", "y"]:
        key_columns.remove(key)

    return Structure(data, key_columns)
```

CHAPTER

FOUR

GROUPED PROPHET

The Grouped Prophet model is a multi-series orchestration framework for building multiple individual models of related, but isolated series data. For example, a project that required the forecasting of airline passengers at major airports around the world would historically require individual orchestration of data acquisition, hyperparameter definitions, model training, metric validation, serialization, and registration of thousands of individual models.

This API consolidates the many thousands of models that would otherwise need to be implemented, trained individually, and managed throughout their frequent retraining and forecasting lifecycles to a single high-level API that simplifies these common use cases that rely on the Prophet forecasting library.

Table of Contents

4.1 Grouped Prophet API

The following sections provide a basic overview of using the *GroupedProphet* API, from fitting of the grouped models, predicting forecasted data, saving, loading, and customization of the underlying `Prophet` instances.

To see working end-to-end examples, you can go to *Tutorials and Examples*. The examples will allow you to explore the data structures required for training, how to extract forecasts for each group, and demonstrations of the saving and loading of trained models.

4.1.1 Model fitting

In order to fit a *GroupedProphet* model instance, the *fit* method is used. Calling this method will process the input `DataFrame` to create a grouped execution collection, fit a `Prophet` model on each individual series, and persist the trained state of each group's model to the object instance.

The arguments for the *fit* method are:

df A 'normalized' DataFrame that contains an endogenous regressor column (the 'y' column), a date (or datetime) column (that defines the ordering, periodicity, and frequency of each series (if this column is a string, the frequency will be inferred)), and grouping column(s) that define the discrete series to be modeled. For further information on the structure of this `DataFrame`, see the *quickstart guide*

group_key_columns The names of the columns within `df` that, when combined (in order supplied) define distinct series. See the *quickstart guide* for further information.

kwargs *[Optional]* Arguments that are used for overrides to the `Prophet` pystan optimizer. Details of what parameters are available and how they might affect the optimization of the model can be found by running `help(pystan.StanModel.optimizing)` from a Python REPL.

Example:

```
grouped_prophet_model = GroupedProphet().fit(df, ["country", "region"])
```

4.1.2 Forecast

The *forecast* method is the 'primary means' of generating future forecast predictions. For each group that was trained in the *Model fitting* of the grouped model, a value of time periods is predicted based upon the last event date (or datetime) from each series' temporal termination.

Usage of this method requires providing two arguments:

horizon The number of events to forecast (supplied as a positive integer)

frequency The periodicity between each forecast event. Note that this value does not have to match the periodicity of the training data (i.e., training data can be in days and predictions can be in months, minutes, hours, or years).

The frequency abbreviations that are allowed can be found at the following link for pandas timeseries.

Note: The generation of error estimates (*yhat_lower* and *yhat_upper*) in the output of a forecast are controlled through the use of the `Prophet` argument `uncertainty_samples` during class instantiation, prior to *Model fitting* being called. Setting this value to *0* will eliminate error estimates and will dramatically increase the speed of training, prediction, and cross validation.

The return data structure for this method will be of a 'stacked' `pandas DataFrame`, consisting of the grouping keys defined (in the order in which they were generated), the grouping columns, elements of the prediction values (deconstructed; e.g. 'weekly', 'yearly', 'daily' seasonality terms and the 'trend'), the date (datetime) values, and the prediction itself (labeled *yhat*).

4.1.3 Predict

A 'manual' method of generating predictions based on discrete date (or datetime) values for each group specified. This method accepts a `DataFrame` as input having columns that define discrete dates to generate predictions for and the grouping key columns that match those supplied when the model was fit. For example, a model trained with the grouping key columns of 'city' and 'country' that included New York City, US and Toronto, Canada as series would generate predictions for both of these cities if the provided `df` argument were supplied:

```
predict_config = pd.DataFrame.from_records(
    {
        "country": ["us", "us", "ca", "ca"],
        "city": ["nyc", "nyc", "toronto", "toronto"],
        "ds": ["2022-01-01", "2022-01-08", "2022-01-01", "2022-01-08"],
    }
)

grouped_prophet_model.predict(predict_config)
```

The structure of this submitted `DataFrame` for the above use case is:

Table 1: Predict *df* Structure

country	city	ds
us	nyc	2022-01-01
us	nyc	2022-01-08
ca	toronto	2022-01-01
ca	toronto	2022-01-08

Usage of this method with the above specified df would generate 4 individual predictions; one for each row.

Note: The *Forecast* method is more appropriate for most use cases as it will continue immediately after the training period of data terminates.

4.1.4 Predict Groups

The *predict_groups* method generates forecast data for a subset of groups that a *diviner.GroupedProphet* model was trained upon.

Example:

```
from diviner import GroupedProphet

model = GroupedProphet().fit(df, ["country", "region"])

subset_forecasts = model.predict_groups(groups=[("US", "NY"), ("FR", "Paris"), ("UA",
↪"Kyiv")],
```

(continues on next page)

(continued from previous page)

```
                                    horizon=90,
                                    frequency="D",
                                    on_error="warn"
                                    )
```

The arguments for the *predict_groups* method are:

groups A collection of one or more groups for which to generate a forecast. The collection of groups must be submitted as a `List[Tuple[str]]` to identify the order-specific group values to retrieve the correct model. For instance, if the model was trained with the specified `group_key_columns` of `["country", "city"]`, a valid `groups` entry would be: `[("US", "LosAngeles"), ("CA", "Toronto")]`. Changing the order within the tuples will not resolve (e.g. `[("NewYork", "US")]` would not find the appropriate model).

Note: Groups that are submitted for prediction that are not present in the trained model will, by default, cause an Exception to be raised. This behavior can be changed to a warning or ignore status with the argument `on_error`.

---

horizon The number of events to forecast (supplied as a positive integer)

frequency The periodicity between each forecast event. Note that this value does not have to match the periodicity of the training data (i.e., training data can be in days and predictions can be in months, minutes, hours, or years).

The frequency abbreviations that are allowed can be found here.

predict_col *[Optional]* The name to use for the generated column containing forecasted data. Default: `"yhat"`

on_error *[Optional]* [Default -> `"raise"`] Dictates the behavior for handling group keys that have been submitted in the `groups` argument that do not match with a group identified and registered during training (`fit`). The modes are:

- **`"raise"`** A `DivinerException` is raised if any supplied groups do not match to the fitted groups.
- **`"warn"`** A warning is emitted (printed) and logged for any groups that do not match to those that the model was fit with.
- **`"ignore"`** Invalid groups will silently fail prediction.

Note: A `DivinerException` will still be raised even in `"ignore"` mode if there are no valid fit groups to match the provided `groups` provided to this method.

4.1.5 Save

Supports saving a *GroupedProphet* model that has been *fit*. The serialization of the model instance does not rely on pickle or cloudpickle, rather a straight-forward json serialization.

```
save_location = "/path/to/store/model"
grouped_prophet_model.save(save_location)
```

4.1.6 Load

Loading a saved *GroupedProphet* model is done through the use of a class method. The *load* method is called as below:

```
load_location = "/path/to/stored/model"
grouped_prophet_model = GroupedProphet.load(load_location)
```

Note: The PyStan backend optimizer instance used to fit the model is not saved (this would require compilation of PyStan on the same machine configuration that was used to fit it in order for it to be valid to reuse) as it is not useful to store and would require additional dependencies that are not involved in cross validation, parameter extraction, forecasting, or predicting. If you need access to the PyStan backend, retrain the model and access the underlying solver prior to serializing to disk.

4.1.7 Overriding Prophet settings

In order to create a *GroupedProphet* instance, there are no required attributes to define. Utilizing the default values will, as with the underlying Prophet library, utilize the default values to perform model fitting. However, there are arguments that can be overridden which are pass-through values to the individual Prophet instances that are created for each group. Since these are **kwargs entries, the names will be argument names for the respective arguments in Prophet.

To see a full listing of available arguments for the given version of Prophet that you are using, the simplest (as well as the recommended manner in the library documentation) is to run a help() command in a Python REPL:

```
from prophet import Prophet
help(Prophet)
```

An example of overriding many of the arguments within the underlying Prophet model for the GroupedProphet API is shown below.

```
grouped_prophet_model = GroupedProphet(
    growth='linear',
    changepoints=None,
    n_changepoints=90,
    changepoint_range=0.8,
    yearly_seasonality='auto',
    weekly_seasonality='auto',
    daily_seasonality='auto',
    holidays=None,
    seasonality_mode='additive',
    seasonality_prior_scale=10.0,
    holidays_prior_scale=10.0,
    changepoint_prior_scale=0.05,
    mcmc_samples=0,
    interval_width=0.8,
    uncertainty_samples=1000,
    stan_backend=None
)
```

4.2 Utilities

4.2.1 Parameter Extraction

The method *extract_model_params* is a utility that extracts the tuning parameters from each individual model from within the *model's* container and returns them as a single DataFrame. Columns are the parameters from the models, while each row is an individual group's Prophet model's parameter values. Having a single consolidated extraction data structure eases the historical registration of model performance and enables a simpler approach to the design of frequent retraining through passive retraining systems (allowing for an easier means by which to acquire priors hyperparameter values on frequently retrained forecasting models).

An example extract from a 2-group model (cast to a dictionary from the `Pandas DataFrame` output) is shown below:

```
{'changepoint_prior_scale': {0: 0.05, 1: 0.05},
 'changepoint_range': {0: 0.8, 1: 0.8},
 'component_modes': {0: {'additive': ['yearly',
                                      'weekly',
                                      'additive_terms',
                                      'extra_regressors_additive',
                                      'holidays'],
                         'multiplicative': ['multiplicative_terms',
                                            'extra_regressors_multiplicative']},
                     1: {'additive': ['yearly',
                                      'weekly',
                                      'additive_terms',
                                      'extra_regressors_additive',
                                      'holidays'],
                         'multiplicative': ['multiplicative_terms',
                                            'extra_regressors_multiplicative']}},
 'country_holidays': {0: None, 1: None},
 'daily_seasonality': {0: 'auto', 1: 'auto'},
 'extra_regressors': {0: OrderedDict(), 1: OrderedDict()},
 'fit_kwargs': {0: {}, 1: {}},
 'grouping_key_columns': {0: ('key2', 'key1', 'key0'),
                          1: ('key2', 'key1', 'key0')},
 'growth': {0: 'linear', 1: 'linear'},
 'holidays': {0: None, 1: None},
 'holidays_prior_scale': {0: 10.0, 1: 10.0},
 'interval_width': {0: 0.8, 1: 0.8},
 'key0': {0: 'T', 1: 'M'},
 'key1': {0: 'A', 1: 'B'},
 'key2': {0: 'C', 1: 'L'},
 'logistic_floor': {0: False, 1: False},
 'mcmc_samples': {0: 0, 1: 0},
 'n_changepoints': {0: 90, 1: 90},
 'seasonality_mode': {0: 'additive', 1: 'additive'},
 'seasonality_prior_scale': {0: 10.0, 1: 10.0},
 'specified_changepoints': {0: False, 1: False},
 'stan_backend': {0: <prophet.models.PyStanBackend object at 0x7f900056d2e0>,
                  1: <prophet.models.PyStanBackend object at 0x7f9000523eb0>},
 'start': {0: Timestamp('2018-01-02 00:02:00'),
           1: Timestamp('2018-01-02 00:02:00')},
 't_scale': {0: Timedelta('1459 days 00:00:00'),
```

(continues on next page)

(continued from previous page)

```
            1: Timedelta('1459 days 00:00:00')},
 'train_holiday_names': {0: None, 1: None},
 'uncertainty_samples': {0: 1000, 1: 1000},
 'weekly_seasonality': {0: 'auto', 1: 'auto'},
 'y_scale': {0: 1099.9530489951537, 1: 764.727400507604},
 'yearly_seasonality': {0: 'auto', 1: 'auto'}}
```

4.2.2 Cross Validation and Scoring

The primary method of evaluating model performance across all groups is by using the method *cross_validate_and_score*. Using this method from a `GroupedProphet` instance that has been fit will perform backtesting of each group's model using the training data set supplied when the *fit* method was called.

The return type of this method is a single consolidated `Pandas DataFrame` that contains metrics as columns with each row representing a distinct grouping key. For example, below is a sample of 3 groups' cross validation metrics.

```
{'coverage': {0: 0.21839080459770113,
              1: 0.057471264367816084,
              2: 0.5114942528735632},
 'grouping_key_columns': {0: ('key2', 'key1', 'key0'),
                          1: ('key2', 'key1', 'key0'),
                          2: ('key2', 'key1', 'key0')},
 'key0': {0: 'T', 1: 'M', 2: 'K'},
 'key1': {0: 'A', 1: 'B', 2: 'S'},
 'key2': {0: 'C', 1: 'L', 2: 'Q'},
 'mae': {0: 14.230668998203283, 1: 34.62100210053155, 2: 46.17014668092673},
 'mape': {0: 0.015166533573997266,
          1: 0.05578282899646585,
          2: 0.047658812366283436},
 'mdape': {0: 0.013636314354422746,
           1: 0.05644041426067295,
           2: 0.039153745874603914},
 'mse': {0: 285.42142900120183, 1: 1459.7746527190932, 2: 3523.9281809854906},
 'rmse': {0: 15.197908800171147, 1: 35.520537302480314, 2: 55.06313841955681},
 'smape': {0: 0.015327226830099487,
           1: 0.0577464576758301 8,
           2: 0.0494437278595581}}
```

Method arguments:

horizon A `pandas.Timedelta` string consisting of two parts: an integer and a periodicity. For example, if the training data is daily, consists of 5 years of data, and the end-use for the project is to predict 14 days of future values every week, a plausible horizon value might be `"21 days"` or `"28 days"`. See pandas documentation for information on the allowable syntax and format for `pandas.Timedelta` values.

metrics A list of metrics that will be calculated following the back-testing cross validation. By default, all of the following will be tested:

- "mae" (mean absolute error)
- "mape" (mean absolute percentage error)
- "mdape" (median absolute percentage error)
- "mse" (mean squared error)

- "rmse" (root mean squared error)
- "smape" (symmetric mean absolute percentage error)

To restrict the metrics computed and returned, a subset of these tests can be supplied to the `metrics` argument.

period The frequency at which each windowed collection of back testing cross validation will be conducted. If the argument `cutoffs` is left as `None`, this argument will determine the spacing between training and validation sets as the cross validation algorithm steps through each series. Smaller values will increase cross validation execution time.

initial The size of the initial training period to use for cross validation windows. The default derived value, if not specified, is `horizon` * 3 with cutoff values for each window set at `horizon` / 2.

parallel Mode of operation for calculating cross validation windows. `None` for serial execution, `'processes'` for multiprocessing pool execution, and `'threads'` for thread pool execution.

cutoffs Optional control mode that allows for defining specific datetime values in `pandas.Timestamp` format to determine where to conduct train and test split boundaries for validation of each window.

kwargs Individual optional overrides to `prophet.diagnostics.cross_validation()` and `prophet.diagnostics.performance_metrics()` functions. See the prophet docs for more information.

4.2.3 Cross Validation

The *`diviner.GroupedProphet.cross_validate()`* method is a wrapper around the `Prophet` function `prophet.diagnostics.cross_validation()`. It is intended to be used as a debugging tool for the 'automated' metric calculation method, see *Cross Validation and Scoring*. The arguments for this method are:

horizon A timedelta formatted string in the `Pandas.Timedelta` format that defines the amount of time to utilize for generating a validation dataset that is used for calculating loss metrics per each cross validation window iteration. Example horizons: (`"30 days"`, `"24 hours"`, `"16 weeks"`). See the pandas Timedelta docs for more information on supported formats and syntax.

period The periodicity of how often a windowed validation will be constructed. Smaller values here will take longer as more 'slices' of the data will be made to calculate error metrics. The format is the same as that of the horizon (i.e. `"60 days"`).

initial The minimum size of data that will be used to build the cross validation window. Values that are excessively small may cause issues with the effectiveness of the estimated overall prediction error and lead to long cross validation runtimes. This argument is in the same format as `horizon` and `period`, a `pandas.Timedelta` format string.

parallel Selection on how to execute the cross validation windows. Supported modes: (`None`, `'processes'`, or `'threads'`). Due to the reuse of the originating dataset for window slice selection, a shared memory instance mode `'threads'` is recommended over using `'processes'` mode.

cutoffs Optional arguments for specified `pandas.Timestamp` values to define where boundaries should be within the group series values. If this is specified, the `period` and `initial` arguments are not used.

Note: For information on how cross validation works within the `Prophet` library, see this link.

The return type of this method is a dictionary of `{<group_key>: <pandas DataFrame>}`, the `DataFrame` containing the cross validation window scores across time horizon splits.

4.2.4 Performance Metrics

The *calculate_performance_metrics* method is a debugging tool that wraps the function performance_metrics from `Prophet`. Usage of this method will generate the defined metric scores for each cross validation window, returning a dictionary of `{<group_key>: <DataFrame of metrics for each window>}`

Method arguments:

cv_results The output of *cross_validate*.

metrics Optional subset list of metrics. See the signature for *cross_validate_and_score()* for supported metrics.

rolling_window Defines the fractional amount of data to use in each rolling window to calculate the performance metrics. Must be in the range of {0: 1}.

monthly Boolean value that, if set to `True`, will collate the windows to ensure that horizons are computed as a factor of months of the year from the cutoff date. This is only useful if the data has a yearly seasonality component to it that relates to day of month.

4.3 Class Signature

class diviner.**GroupedProphet**(***kwargs*)

A class for executing multiple Prophet models from a single submitted DataFrame. The structure of the input DataFrame to the `fit` method must have defined grouping columns that are used to build the per-group processing dataframes and models for each group. The names of these columns, passed in as part of the `fit` method are required to be present in the DataFrame schema. Any parameters that are needed to override Prophet parameters can be submitted as kwargs to the `fit` and `predict` methods. These settings will be overridden globally for all grouped models within the submitted DataFrame.

For the Prophet initialization constructor, showing which arguments are available to be passed in as `kwargs` in this class constructor, see: https://github.com/facebook/prophet/blob/main/python/prophet/forecaster.py

calculate_performance_metrics(*cv_results*, *metrics=None*, *rolling_window=0.1*, *monthly=False*)

Model debugging utility function for evaluating performance metrics from the grouped cross validation extract. This will output a metric table for each time boundary from cross validation, for each model. note: This output will be very large and is intended to be used as a debugging tool only.

Parameters

- **cv_results** – The output return of `group_cross_validation`
- **metrics** – (Optional) overrides (subtractive) for metrics to generate for this function's output. note: see supported metrics in Prophet documentation: (https://facebook.github.io/prophet/docs/diagnostics. html#cross-validation) note: any model in the collection that was fit with the argument `uncertainty_samples` set to `0` will have the metric `'coverage'` removed from evaluation due to the fact that `` `yhat_error` `` values are not calculated with that configuration of that parameter.
- **rolling_window** – Defines how much data to use in each rolling window as a range of `[0, 1]` for computing the performance metrics.
- **monthly** – If set to true, will collate the windows to ensure that horizons are computed as number of months from the cutoff date. Only useful for date data that has yearly seasonality associated with calendar day of month.

Returns Dictionary of {`'group_key'`: <performance metrics per window pandas DataFrame>}

cross_validate(*horizon*, *period=None*, *initial=None*, *parallel=None*, *cutoffs=None*)

Utility method for generating the cross validation dataset for each grouping key. This is a wrapper around `prophet.diagnostics.cross_validation` and uses the same signature arguments as that function. It applies each globally to all groups. note: the output of this will be a Pandas DataFrame for each grouping key per cutoff boundary in the datetime series. The output of this function will be many times larger than the original input data utilized for training of the model.

Parameters

- **horizon** – pandas Timedelta formatted string (i.e. `'14 days'` or `'18 hours'`) to define the amount of time to utilize for a validation set to be created.
- **period** – the periodicity of how often a windowed validation will occur. Default is 0.5 * horizon value.
- **initial** – The minimum amount of training data to include in the first cross validation window.
- **parallel** – mode of computing cross validation statistics. One of: (`None`, `'processes'`, or `'threads'`)
- **cutoffs** – List of pandas Timestamp values that specify cutoff overrides to be used in conducting cross validation.

Returns Dictionary of {`'group_key'`: <cross validation Pandas DataFrame>}

cross_validate_and_score(*horizon*, *period=None*, *initial=None*, *parallel=None*, *cutoffs=None*, *metrics=None*, ***kwargs*)

Metric scoring method that will run backtesting cross validation scoring for each time series specified within the model after a `fit` has been performed.

Note: If the configuration overrides for the model during `fit` set `uncertainty_samples=0`, the metric `coverage` will be removed from metrics calculation, saving a great deal of runtime overhead since the prediction errors `(yhat_upper, yhat_lower)` will not be calculated.

Note: overrides to functionality of both `cross_validation` and `performance_metrics` within Prophet's `diagnostics` module are handled here as `kwargs`. These arguments in this method's signature are directly passed, per model, to prophet's `cross_validation` function.

Parameters

- **horizon** – String pandas `Timedelta` format that defines the length of forecasting values to generate in order to acquire error metrics. examples: `'30 days'`, `'1 year'`
- **metrics** – Specific subset list of metrics to calculate and return. note: see supported metrics in Prophet documentation: https://facebook.github.io/prophet/docs/diagnostics.html#cross-validation note: The `coverage` metric will be removed if error estiamtes are not configured to be calculated as part of the Prophet `fit` method by setting `uncertainty_samples=0` within the GroupedProphet `fit` method.
- **period** – the periodicity of how often a windowed validation will occur. Default is 0.5 * horizon value.
- **initial** – The minimum amount of training data to include in the first cross validation window.
- **parallel** – mode of computing cross validation statistics. Supported modes: (`None`, `'processes'`, or `'threads'`)
- **cutoffs** – List of pandas `Timestamp` values that specify cutoff overrides to be used in conducting cross validation.

- **kwargs** – cross validation overrides to Prophet's `prophet.diagnostics.cross_validation` and `prophet.diagnostics.performance_metrics` functions

Returns A consolidated Pandas DataFrame containing the specified metrics to test as columns with each row representing a group.

extract_model_params()

Utility method for extracting all model parameters from each model within the processed groups.

Returns A consolidated pandas DataFrame containing the model parameters as columns with each row entry representing a group.

fit(*df*, *group_key_columns*, *y_col='y'*, *datetime_col='ds'*, ***kwargs*)

Main `fit` method for executing a Prophet `fit` on the submitted DataFrame, grouped by the `group_key_columns` submitted. When initiated, the input DataFrame `df` will be split into an iterable collection that represents a core series to be fit against. This `fit` method is a per-group wrapper around Prophet's `fit` implementation. See: https://facebook.github.io/prophet/docs/quick_start.html for information on the basic API, as well as links to the source code that will demonstrate all of the options available for overriding default functionality. For a full description of all parameters that are available to the optimizer, run the following in a shell:

Listing 1: Retrieving pystan parameters

```
import pystan

help(pystan.StanModel.optimizing)
```

Parameters

- **df** – Normalized pandas DataFrame containing `group_key_columns`, a `'ds'` column, and a target `'y'` column. An example normalized data set to be used in this method:

region	zone	ds	y
northeast	1	'2021-10-01'	1234.5
northeast	2	'2021-10-01'	3255.6
northeast	1	'2021-10-02'	1255.9

- **group_key_columns** – The columns in the `df` argument that define, in aggregate, a unique time series entry. For example, with the DataFrame referenced in the `df` param, group_key_columns could be: (`'region'`, `'zone'`) Specifying an incomplete grouping collection, while valid through this API (i.e., ('region')), can cause serious problems with any forecast that is built with this API. Ensure that all relevant keys are defined in the input *df* and declared in this param to ensure that the appropriate per-univariate series data is used to train each model.
- **y_col** – The name of the column within the DataFrame input to any method within this class that contains the endogenous regressor term (the raw data that will be used to train and use as a basis for forecasting).
- **datetime_col** – The name of the column within the DataFrame input that defines the datetime or date values associated with each row of the endogenous regressor (`y_col`) data.
- **kwargs** – overrides for underlying `Prophet .fit() **kwargs` (i.e., optimizer backend library configuration overrides) for further information, see: (https://facebook.github.io/prophet/docs/diagnostics.html #hyperparameter-tuning).

Returns object instance (self) of GroupedProphet

forecast(*horizon: int*, *frequency: str*)

Forecasting method that will automatically generate forecasting values where the `'ds'` datetime value from the `fit` DataFrame left off. For example: If the last datetime value in the training data is `'2021-01-01 00:01:00'`, with a specified `frequency` of `'1 day'`, the beginning of the forecast value will be `'2021-01-02 00:01:00'` and will continue at a 1 day frequency for `horizon` number of entries. This implementation wraps the Prophet library's `prophet.forecaster.Prophet.make_future_dataframe` method.

Note: This will generate a forecast for each group that was present in the `fit` input DataFrame `df` argument. Time horizon values are dependent on the per-group `'ds'` values for each group, which may result in different datetime values if the source fit DataFrame did not have consistent datetime values within the `'ds'` column for each group.

Note: For full listing of supported periodicity strings for the `frequency` parameter, see: https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#offset-aliases

Parameters

- **horizon** – The number of row events to forecast
- **frequency** – The frequency (periodicity) of Pandas date_range format (i.e., `'D'`, `'M'`, `'Y'`)

Returns A consolidated (unioned) single DataFrame of forecasts for all groups

classmethod load(*path: str*)

Load the model from the specified path, deserializing it from its JSON string representation and returning a `GroupedProphet` instance.

Parameters **path** – File system path of a saved `GroupedProphet` model.

Returns An instance of GroupedProphet with `fit` attributes applied.

predict(*df*, *predict_col: str = 'yhat'*)

Main prediction method for generating forecast values based on the group keys and dates for each that are passed in to this method. The structure of the DataFrame submitted to this method is the same normalized format that `fit` takes as a DataFrame argument. i.e.:

region	zone	ds
northeast	1	'2021-10-01'
northeast	2	'2021-10-01'
northeast	1	'2021-10-02'

Parameters

- **df** – Normalized DataFrame consisting of grouping key entries and the dates to forecast for each group.
- **predict_col** – The name of the column in the output `DataFrame` that contains the forecasted series data.

Returns A consolidated (unioned) single DataFrame of all groups forecasts

predict_groups(*groups: List[Tuple[str]]*, *horizon: int*, *frequency: str*, *predict_col: str = 'yhat'*, *on_error: str = 'raise'*)

This is a prediction method that allows for generating a subset of forecasts based on the collection of keys.

Parameters

- **groups** – `List[Tuple[str]]` the collection of group(s) to generate forecast predictions. The group definitions must be the values within the `group_key_columns` that were used during the `fit` of the model in order to return valid forecasts.

Note: The positional ordering of the values are important and must match the order of `group_key_columns` for the `fit` argument to provide correct prediction forecasts.

- **horizon** – The number of row events to forecast
- **frequency** – The frequency (periodicity) of Pandas date_range format (i.e., `'D'`, `'M'`, `'Y'`)
- **predict_col** – The name of the column in the output `DataFrame` that contains the forecasted series data. Default: `"yhat"`
- **on_error** – Alert level setting for handling mismatched group keys. Default: `"raise"` The valid modes are:
  - "ignore" - no logging or exception raising will occur if a submitted group key in the `groups` argument is not present in the model object.

    Note: This is a silent failure mode and will not present any indication of a failure to generate forecast predictions.

  - "warn" - any keys that are not present in the fit model will be recorded as logged warnings.
  - "raise" - any keys that are not present in the fit model will cause a `DivinerException` to be raised.

Returns A consolidated (unioned) single DataFrame of forecasts for all groups specified in the `groups` argument.

save(*path: str*)

Serialize the model as a JSON string and write it to the provided path. note: The model must be fit in order to save it.

Parameters **path** – Location on the file system to store the model.

Returns None

CHAPTER

FIVE

GROUPED PMDARIMA

The Grouped `pmdarima` API is a multi-series orchestration framework for building multiple individual models of related, but isolated series data. For example, a project that required the forecasting of inventory demand at regional warehouses around the world would historically require individual orchestration of data acquisition, hyperparameter definitions, model training, metric validation, serialization, and registration of tens of thousands of individual models based on the permutations of SKU and warehouse location.

This API consolidates the many thousands of models that would otherwise need to be implemented, trained individually, and managed throughout their frequent retraining and forecasting lifecycles to a single high-level API that simplifies these common use cases that rely on the pmdarima forecasting library.

Table of Contents

5.1 Grouped pmdarima API

The following sections provide a basic overview of using the *GroupedPmdarima* API, from fitting of the grouped models, predicting forecasted data, saving, loading, and customization of the underlying `pmdarima` instances.

To see working end-to-end examples, you can go to *Tutorials and Examples*. The examples will allow you to explore the data structures required for training, how to extract forecasts for each group, and demonstrations of the saving and loading of trained models.

5.1.1 Base Estimators and API interface

The usage of the *GroupedPmdarima* API is slightly different from the other grouped forecasting library wrappers within `Diviner`. This is due to the ability of `pmdarima` to support multiple modes of configuration.

These modes that are available to construct a model are:

- Passing an `ARIMA` model template (wrapper around statsmodels ARIMA)
- Using the native `pmdarima` `AutoARIMA` model template
- Constructing a `pmdarima` Pipeline template

The *GroupedPmdarima* implementation requires the submission of one of these 3 model templates to set the base configured model architecture for each group.

For example:

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima

# Define the base ARIMA with a preset ordering parameter
base_arima_model = ARIMA(order=(1, 0, 2))

# Define the model template in the GroupedPmdarima constructor
grouped_arima = GroupedPmdarima(model_template=base_arima_model)
```

The above example is intended only to showcase the interface between a base estimator (`base_arima_model`) and the instance constructor for GroupedPmdarima. For a more in-depth and realistic example of utilizing an ARIMA model manually, see the additional statistical validation steps that would be required for this in the *Tutorials and Examples* section of the docs.

5.1.2 Model fitting

In order to fit a *GroupedPmdarima* model instance, the *fit* method is used. Calling this method will process the input DataFrame to create a grouped execution collection, fit a pmdarima model type on each individual series, and persist the trained state of each group's model to the object instance.

The arguments for the *fit* method are:

df A 'normalized' DataFrame that contains an endogenous regressor column (the 'y' column), a date (or datetime) column (that defines the ordering, periodicity, and frequency of each series (if this column is a string, the frequency will be inferred)), and grouping column(s) that define the discrete series to be modeled. For further information on the structure of this DataFrame, see the *quickstart guide*

group_key_columns The names of the columns within df that, when combined (in order supplied) define distinct series. See the *quickstart guide* for further information.

y_col Name of the endogenous regressor term within the DataFrame argument df. This column contains the values of the series that are used during training.

datetime_col Name of the column within the df argument DataFrame that defines the datetime ordering of the series data.

exog_cols *[Optional]* A collection of column names within the submitted data that contain exogenous regressor elements to use as part of model fitting and predicting. The data within each column will be assembled into a 2D array for use in the regression.

Note: pmdarima currently has exogeneous regressor support marked as a future deprecated feature. Usage of this functionality is not recommended except for existing legacy implementations.

---

ndiffs *[Optional]* A dictionary of {<group_key>: <d value>} for the differencing term for each group. This is intended to function alongside the output from the *diviner.PmdarimaAnalyzer.calculate_ndiffs()* method, serving to reduce the search space for AutoARIMA by supplying fixed d values to each group's model.

nsdiffs *[Optional]* A dictionary of {<group_key>: <D value>} for the seasonal differencing term for each group. This is intended to function alongside the output from the *diviner.PmdarimaAnalyzer.calculate_nsdiffs()* method, serving to reduce the search space for AutoARIMA by supplying fixed D values to each group's model.

Note: These values will only be used if the models being fit are seasonal models. The value m must be set on the underlying ARIMA or AutoARIMA model for seasonality order components to be used.

---

silence_warnings *[Optional]* Whether to silence stdout reporting of the underlying pmdarima fit process. Default: False.

fit_kwargs *[Optional]* fit_kwargs for pmdarima ARIMA, AutoARIMA, or Pipeline stages overrides. For more information, see the pmdarima docs

Example:

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima

base_arima_model = ARIMA(order=(1, 0, 2))

grouped_arima = GroupedPmdarima(model_template=base_arima_model)
```

(continues on next page)

(continued from previous page)

```
grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")
```

5.1.3 Predict

The *predict* method generates forecast data for each grouped series within the meta *diviner.GroupedPmdarima* model.

Example:

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima

base_arima_model = ARIMA(order=(1, 0, 2))

grouped_arima = GroupedPmdarima(model_template=base_arima_model)

grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

forecasts = grouped_arima_model.predict(n_periods=30)
```

The arguments for the *predict* method are:

n_periods The number of future periods to generate from the end of each group's series. The first value of the prediction forecast series will begin at one periodicity value after the end of the training series. For example, if the training series was of daily data from 2019-10-01 to 2021-10-02, the start of the prediction series output would be 2021-10-03 and continue for `n_periods` days from that point.

predict_col *[Optional]* The name to use for the generated column containing forecasted data. Default: `"yhat"`

alpha *[Optional]* Confidence interval significance value for error estimates. Default: `0.05`.

Note: `alpha` is only used if the boolean flag `return_conf_int` is set to `True`.

---

return_conf_int *[Optional]* Boolean flag for whether or not to calculate confidence intervals for the predicted forecasts. If `True`, the columns `"yhat_upper"` and `"yhat_lower"` will be added to the output `DataFrame` for the upper and lower confidence intervals for the predictions.

inverse_transform *[Optional]* Used exclusively for `Pipeline` based models that include an endogeneous transformer such as `BoxCoxEndogTransformer` or `LogEndogTransformer`. Default: `True` (although it only applies *if* the `model_template` type passed in is a `Pipeline` that contains a transformer). An inversion of the endogeneous regression term can be helpful for distributions that are highly non-normal. For further reading on what the purpose of these functions are, why they are used, and how they might be applicable to a given time series, see data transformation.

exog *[Optional]* If the original model was trained with an exogeneous regressor elements, the prediction will require these 2D arrays at prediction time. This argument is used to hold the 2D array of future exogeneous regressor values to be used in generating the prediction for the regressor.

predict_kwargs *[Optional]* Extra `kwarg` arguments for any of the transform stages of a `Pipeline` or for additional `predict` `kwargs` to the model instance. `Pipeline` `kwargs` are specified in the manner of `sklearn` `Pipeline` format (i.e., `<stage_name>__<arg name>=<value>`. e.g., to change the values of a fourier transformer at prediction time, the override would be: `{'fourier__n_periods': 45}`)

5.1.4 Predict Groups

The *predict_groups* method generates forecast data for a subset of groups that a *diviner.GroupedPmdarima* model was trained upon.

Example:

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima

base_model = ARIMA(order=(2, 1, 2))

grouped_arima = GroupedPmdarima(model_template=base_model)

model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

subset_forecasts = model.predict_groups(groups=[("US", "NY"), ("FR", "Paris"), ("UA",
↪"Kyiv")], n_periods=90)
```

The arguments for the *predict_groups* method are:

groups A collection of one or more groups for which to generate a forecast. The collection of groups must be submitted as a `List[Tuple[str]]` to identify the order-specific group values to retrieve the correct model. For instance, if the model was trained with the specified `group_key_columns` of `["country", "city"]`, a valid `groups` entry would be: `[("US", "LosAngeles"), ("CA", "Toronto")]`. Changing the order within the tuples will not resolve (e.g. `[("NewYork", "US")]` would not find the appropriate model).

Note: Groups that are submitted for prediction that are not present in the trained model will, by default, cause an Exception to be raised. This behavior can be changed to a warning or ignore status with the argument `on_error`.

---

n_periods The number of future periods to generate from the end of each group's series. The first value of the prediction forecast series will begin at one periodicity value after the end of the training series. For example, if the training series was of daily data from 2019-10-01 to 2021-10-02, the start of the prediction series output would be 2021-10-03 and continue for `n_periods` days from that point.

predict_col *[Optional]* The name to use for the generated column containing forecasted data. Default: `"yhat"`

alpha *[Optional]* Confidence interval significance value for error estimates. Default: `0.05`.

Note: `alpha` is only used if the boolean flag `return_conf_int` is set to `True`.

---

return_conf_int *[Optional]* Boolean flag for whether or not to calculate confidence intervals for the predicted forecasts. If `True`, the columns `"yhat_upper"` and `"yhat_lower"` will be added to the output `DataFrame` for the upper and lower confidence intervals for the predictions.

inverse_transform *[Optional]* Used exclusively for `Pipeline` based models that include an endogeneous transformer such as `BoxCoxEndogTransformer` or `LogEndogTransformer`. Default: `True` (although it only applies *if* the `model_template` type passed in is a `Pipeline` that contains a transformer). An inversion of the endogeneous regression term can be helpful for distributions that are highly non-normal. For further reading on what the purpose of these functions are, why they are used, and how they might be applicable to a given time series, see this link.

exog *[Optional]* If the original model was trained with an exogeneous regressor elements, the prediction will require these 2D arrays at prediction time. This argument is used to hold the 2D array of future exogeneous regressor values to be used in generating the prediction for the regressor.

on_error *[Optional]* [Default -> `"raise"`] Dictates the behavior for handling group keys that have been submitted in the `groups` argument that do not match with a group identified and registered during training (`fit`). The modes are:

- **`"raise"`** A `DivinerException` is raised if any supplied groups do not match to the fitted groups.
- **`"warn"`** A warning is emitted (printed) and logged for any groups that do not match to those that the model was fit with.
- **`"ignore"`** Invalid groups will silently fail prediction.

Note: A `DivinerException` will still be raised even in `"ignore"` mode if there are no valid fit groups to match the provided `groups` provided to this method.

---

predict_kwargs *[Optional]* Extra `kwarg` arguments for any of the transform stages of a `Pipeline` or for additional `predict` `kwargs` to the model instance. `Pipeline` `kwargs` are specified in the manner of `sklearn` `Pipeline` format (i.e., `<stage_name>__<arg name>=<value>`. e.g., to change the values of a fourier transformer at prediction time, the override would be: `{'fourier__n_periods': 45}`)

5.1.5 Save

Saves a *GroupedPmdarima* instance that has been *fit*. The serialization of the model instance uses a base64 encoding of the pickle serialization of each model instance within the grouped structure.

Example:

```
from pmdarima.arima.arima import ARIMA
from diviner import GroupedPmdarima

base_arima_model = ARIMA(order=(1, 0, 2))

grouped_arima = GroupedPmdarima(model_template=base_arima_model)

grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

save_location = "/path/to/saved/model"

grouped_arima_model.save(save_location)
```

5.1.6 Load

Loads a *GroupedPmdarima* serialized model from a storage location.

Example:

```
from diviner import GroupedPmdarima

load_location = "/path/to/saved/model"

loaded_model = GroupedPmdarima.load(load_location)
```

Note: The *load* method is a class method. As such, the initialization argument `model_template` does not need to be provided. It will be set on the loaded object based on the template that was provided during initial training before serialization.

5.2 Utilities

5.2.1 Parameter Extraction

To extract the parameters that are either explicitly (or, in the case of `AutoARIMA`, selectively) set during the fitting of each individual model contained within the grouped collection, the method *get_model_params* is used to extract the per-group parameters for each model into an output `Pandas DataFrame`.

Note: The parameters can only be extracted from a `GroupedPmdarima` model that has been fit.

Example:

```
from pmdarima.arima.auto import AutoARIMA
from diviner import GroupedPmdarima

base_arima_model = AutoARIMA(max_order=7, d=1, m=7, max_iter=1000)

grouped_arima = GroupedPmdarima(model_template=base_arima_model)

grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

fit_parameters = grouped_arima_model.get_model_params()
```

5.2.2 Metrics Extraction

This functionality allows for the retrieval of fitting metrics that are attached to the underlying `SARIMA` model. These are not the typical loss metrics that can be calculated through cross validation backtesting.

The metrics that are returned from fitting are:

- hqic (Hannan-Quinn information criterion)
- aicc (Corrected Akaike information criterion; aic for small sample sizes)
- oob (out of bag error)
- bic (Bayesian information criterion)
- aic (Akaike information criterion)

Note: Out of bag error metric (oob) is only calculated if the underlying `ARIMA` model has a value set for the argument `out_of_sample_size`. See out-of-bag-error for more information.

Example:

```
from pmdarima.arima.auto import AutoARIMA
from diviner import GroupedPmdarima

base_arima_model = AutoARIMA(max_order=7, d=1, m=7, max_iter=1000)

grouped_arima = GroupedPmdarima(model_template=base_arima_model)

grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

fit_metrics = grouped_arima_model.get_metrics()
```

5.2.3 Cross Validation Backtesting

Cross validation utilizing backtesting is the primary means of evaluating whether a given model will perform robustly in generating forecasts based on time period horizon events throughout the historical series.

In order to use the cross validation functionality in the method *diviner.GroupedPmdarima.cross_validate()*, one of two windowing split objects must be passed into the method signature:

- RollingForecastCV
- SlidingWindowForecastCV

Arguments to the *diviner.GroupedPmdarima.cross_validate()* method:

df The original source `DataFrame` that was used during *diviner.GroupedPmdarima.fit()* that contains the endogenous series data. This `DataFrame` must contain the columns that define the constructed groups (i.e., missing group data will not be scored and groups that are not present in the model object will raise an Exception).

metrics A collection of metric names to be used for evaluation. Submitted metrics must be one or more of:

- smape
- mean_absolute_error
- mean_squared_error

Default: `{"smape", "mean_absolute_error", "mean_squared_error"}`

cross_validator The cross validation object (either `RollingForecastCV` or `SlidingWindowForecastCV`). See the example below for how to submit this object to the *diviner.GroupedPmdarima.cross_validate()* method.

error_score The default value to assign to a window evaluation if an error occurs during loss calculation.

Default: `np.nan`

In order to throw an Exception, a str value of "raise" can be provided. Otherwise, supply a float.

verbosity Level of verbosity to print during training and cross validation. The lower the integer value, the fewer lines of debugging text is printed to stdout.

Default: `0` (no printing)

Example:

```
from pmdarima.arima.auto import AutoARIMA
from pmdarima.model_selection import SlidingWindowForecastCV
from diviner import GroupedPmdarima

base_arima_model = AutoARIMA(max_order=7, d=1, m=7, max_iter=1000)
```

(continues on next page)

(continued from previous page)

```
grouped_arima = GroupedPmdarima(model_template=base_arima_model)

grouped_arima_model = grouped_arima.fit(df, ["country", "region"], "sales", "date")

cv_window = SlidingWindowForecastCV(h=28, step=180, window_size=365)

grouped_arima_cv = grouped_arima_model.cross_validate(df=df,
                                                      metrics=["mean_squared_error",
↪"smape"],
                                                      cross_validator=cv_window,
                                                      error_score=np.nan,
                                                      verbosity=1
                                                     )
```

5.3 Class Signature of GroupedPmdarima

class diviner.**GroupedPmdarima**(*model_template*)

cross_validate(*df*, *metrics*, *cross_validator*, *error_score=nan*, *verbosity=0*)

Method for performing cross validation on each group of the fit model. The supplied cross_validator to this method will be used to perform either rolling or shifting window prediction validation throughout the data set. Windowing behavior for the cross validation must be defined and configured through the cross_validator that is submitted. See: https://alkaline-ml.com/pmdarima/modules/classes.html#cross-validation-split-utilities for details on the underlying implementation of cross validation with `pmdarima`.

Parameters

- **df** – A `DataFrame` that contains the endogenous series and the grouping key columns that were defined during training. Any missing key entries will not be scored. Note that each group defined within the model will be retrieved from this `DataFrame`. keys that do not exist will raise an Exception.
- **metrics** – A list of metric names or string of single metric name to use for cross validation metric calculation.
- **cross_validator** – A cross validator instance from `pmdarima.model_selection` (`RollingForecastCV` or `SlidingWindowForecastCV`). Note: setting low values of `h` or `step` will dramatically increase execution time).
- **error_score** – Default value to assign to a score calculation if an error occurs in a given window iteration.

  Default: `np.nan` (a silent ignore of the failure)
- **verbosity** – print verbosity level for `pmdarima`'s cross validation stages.

  Default: `0` (no printing to stdout)

Returns `Pandas DataFrame` containing the group information and calculated cross validation metrics for each group.

fit(*df*, *group_key_columns*, *y_col: str*, *datetime_col: str*, *exog_cols: Optional[List[str]] = None*, *ndiffs: Optional[Dict] = None*, *nsdiffs: Optional[Dict] = None*, *silence_warnings: bool = False*, ***fit_kwargs*)

Fit method for training a `pmdarima` model on the submitted normalized DataFrame. When initial-

ized, the input DataFrame will be split into an iterable collection of grouped data sets based on the `group_key_columns` arguments, which is then used to fit individual `pmdarima` models (or a supplied `Pipeline`) upon the templated object supplied as a class instance argument *model_template*. For API information for `pmdarima`'s `ARIMA`, `AutoARIMA`, and `Pipeline` APIs, see: https://alkaline-ml.com/pmdarima/modules/classes.html#api-ref

Parameters

- **df** – A normalized group data set consisting of a datetime column that defines ordering of the series, an endogenous regressor column that specifies the series data for training (e.g. `y_col`), and column(s) that define the grouping of the series data.

  An example normalized data set:

region	zone	country	ds	y
'northeast'	1	"US"	"2021-10-01"	1234.5
'northeast'	2	"US"	"2021-10-01"	3255.6
'northeast'	1	"US"	"2021-10-02"	1255.9

  Wherein the grouping_key_columns could be one, some, or all of `['region', 'zone', 'country']`, the datetime_col would be the *'ds'* column, and the series `y_col` (endogenous regressor) would be *'y'*.

- **group_key_columns** – The columns in the `df` argument that define, in aggregate, a unique time series entry. For example, with the DataFrame referenced in the `df` param, group_key_columns could be: `('region', 'zone')` or `('region')` or `('country', 'region', 'zone')`

- **y_col** – The name of the column within the DataFrame input to any method within this class that contains the endogenous regressor term (the raw data that will be used to train and use as a basis for forecasting).

- **datetime_col** – The name of the column within the DataFrame input that defines the datetime or date values associated with each row of the endogenous regressor (`y_col`) data.

- **exog_cols** – An optional collection of column names within the submitted data to class methods that contain exogenous regressor elements to use as part of model fitting and predicting.

  Default: `None`

- **ndiffs** – optional overrides to the `d` `ARIMA` differencing term for stationarity enforcement. The structure of this argument is a dictionary in the form of: `{<group_key>: <d_term>}`. To calculate, use `diviner.PmdarimaAnalyzer.calculate_ndiffs()`

  Default: `None`

- **nsdiffs** – optional overrides to the `D` SARIMAX seasonal differencing term for seasonal stationarity enforcement. The structure of this argument is a dictionary in the form of: `{<group_key>: <D_term>}`. To calculate, use :py:meth:`diviner.PmdarimaAnalyzer.calculate_nsdiffs`

  Default: `None`

- **silence_warnings** – If `True`, removes `SARIMAX` and underlying optimizer warning message from stdout printing. With a sufficiently large nubmer of groups to process, the volume of these messages to stdout may become very large.

  Default: `False`

- **fit_kwargs** – `fit_kwargs` for `pmdarima`'s `ARIMA`, `AutoARIMA`, or `Pipeline` stage overrides. For more information, see the `pmdarima` docs: https://alkaline-ml.com/pmdarima/index.html

Returns object instance of `GroupedPmdarima` with the persisted fit model attached.

`get_metrics()`

Retrieve the `ARIMA` fit metrics that are generated during the `AutoARIMA` or `ARIMA` training event. Note: These metrics are not validation metrics. Use the `cross_validate()` method for retrieving back-testing error metrics.

Returns `Pandas DataFrame` with metrics provided as columns and a row entry per group.

`get_model_params()`

Retrieve the parameters from the `fit` `model_template` that was passed in and return them in a denormalized `Pandas DataFrame`. Parameters in the return `DataFrame` are columns with a row for each group defined during `fit()`.

Returns `Pandas DataFrame` with `fit` parameters for each group.

`classmethod load(`*path: str*`)`

Load a `GroupedPmdarima` instance from a saved serialized version. Note: This is a class instance and as such, a `GroupedPmdarima` instance does not need to be initialized in order to load a saved model. For example: `loaded_model = GroupedPmdarima.load(<location>)`

Parameters **path** – The path to a serialized instance of `GroupedPmdarima`

Returns The `GroupedPmdarima` instance that was saved.

`predict(`*n_periods: int*, *predict_col: str = 'yhat'*, *alpha: float = 0.05*, *return_conf_int: bool = False*, *inverse_transform: bool = True*, *exog=None*, ***predict_kwargs*`)`

Prediction method for generating forecasts for each group that has been trained as part of a call to `fit()`. Note that `pmdarima`'s API does not support predictions outside of the defined datetime frequency that was validated during training (i.e., if the series endogenous data is at an hourly frequency, the generated predictions will be at an hourly frequency and cannot be modified from within this method).

Parameters

- **n_periods** – The number of future periods to generate. The start of the generated predictions will be 1 frequency period after the maximum datetime value per group during training. For example, a data set used for training that has a datetime frequency in days that ends on 7/10/2021 will, with a value of `n_periods=7`, start its prediction on 7/11/2021 and generate daily predicted values up to and including 7/17/2021.
- **predict_col** – The name to be applied to the column containing predicted data.

  Default: `'yhat'`
- **alpha** – Optional value for setting the confidence intervals for error estimates. Note: this is only utilized if `return_conf_int` is set to `True`.

  Default: `0.05` (representing a 95% CI)
- **return_conf_int** – Boolean flag for whether to calculate confidence interval error estimates for predicted values. The intervals of `yhat_upper` and `yhat_lower` are based on the `alpha` parameter.

  Default: `False`
- **inverse_transform** – Optional argument used only for `Pipeline` models that include either a `BoxCoxEndogTransformer` or a `LogEndogTransformer`.

  Default: `True`

- **exog** – Exogenous regressor components as a 2-D array. Note: if the model is trained with exogenous regressor components, this argument is required.

  Default: None

- **predict_kwargs** – Extra kwarg arguments for any of the transform stages of a Pipeline or for additional predict kwargs to the model instance. Pipeline kwargs are specified in the manner of sklearn Pipeline format (i.e., <stage_name>__<arg name>=<value>. e.g., to change the values of a fourier transformer at prediction time, the override would be: {'fourier__n_periods': 45})

Returns A consolidated (unioned) single DataFrame of predictions per group.

predict_groups(*groups: List[Tuple[str]]*, *n_periods: int*, *predict_col: str = 'yhat'*, *alpha: float = 0.05*, *return_conf_int: bool = False*, *inverse_transform: bool = False*, *exog=None*, *on_error: str = 'raise'*, ***predict_kwargs*)

This is a prediction method that allows for generating a subset of forecasts based on the collection of keys. By specifying individual groups in the groups argument, a limited scope forecast can be performed without incurring the runtime costs associated with predicting all groups.

Parameters

- **groups** – List[Tuple[str]] the collection of group (s) to generate forecast predictions. The group definitions must be the values within the group_key_columns that were used during the fit of the model in order to return valid forecasts.

  Note: The positional ordering of the values are important and must match the order of group_key_columns for the fit argument to provide correct prediction forecasts.

- **n_periods** – The number of row events to forecast

- **predict_col** – The name of the column in the output DataFrame that contains the forecasted series data. Default: "yhat"

- **alpha** – Optional value for setting the confidence intervals for error estimates. Note: this is only utilized if return_conf_int is set to True.

  Default: 0.05 (representing a 95% CI)

- **return_conf_int** – Boolean flag for whether to calculate confidence interval error estimates for predicted values. The intervals of yhat_upper and yhat_lower are based on the alpha parameter.

  Default: False

- **inverse_transform** – Optional argument used only for Pipeline models that include either a BoxCoxEndogTransformer or a LogEndogTransformer.

  Default: False

- **exog** – Exogenous regressor components as a 2-D array. Note: if the model is trained with exogenous regressor components, this argument is required.

  Default: None

- **predict_kwargs** – Extra kwarg arguments for any of the transform stages of a Pipeline or for additional predict kwargs to the model instance. Pipeline kwargs are specified in the manner of sklearn Pipeline format (i.e., <stage_name>__<arg name>=<value>. e.g., to change the values of a fourier transformer at prediction time, the override would be: {'fourier__n_periods': 45})

- **on_error** – Alert level setting for handling mismatched group keys. Default: `"raise"` The valid modes are:
  - "ignore" - no logging or exception raising will occur if a submitted group key in the `groups` argument is not present in the model object.

    Note: This is a silent failure mode and will not present any indication of a failure to generate forecast predictions.

  - "warn" - any keys that are not present in the fit model will be recorded as logged warnings.
  - "raise" - any keys that are not present in the fit model will cause a `DivinerException` to be raised.

Returns A consolidated (unioned) single DataFrame of forecasts for all groups specified in the `groups` argument.

save(*path: str*)

Serialize and write the instance of this class (if it has been fit) to the path specified. Note: The serialized model is base64 encoded for top-level items and `pickle`'d for `pmdarima` individual group models and any `Pandas DataFrame`.

Parameters **path** – Path to write this model's instance to.

Returns None

5.4 Grouped pmdarima Analysis tools

Warning: The `PmdarimaAnalyzer` module is in experimental mode. The methods and signatures are subject to change in the future with no deprecation warnings.

As a companion to `Diviner`'s *diviner.GroupedPmdarima* class, an analysis toolkit class is provided. Contained within this class, *PmdarimaAnalyzer*, are the following utility methods:

- decompose_groups
- calculate_ndiffs
- calculate_nsdiffs
- calculate_is_constant
- calculate_acf
- calculate_pacf

See below for a brief description of each of these utility methods that are available for group processing through the `PmdarimaAnalyzer` API.

Object instantiation:

```
from diviner import PmdarimaAnalyzer

analyzer = PmdarimaAnalyzer(
    df=df,
```

(continues on next page)

(continued from previous page)

```
    group_key_columns=["country", "region"],
    y_col="orders",
    datetime_col="date"
)
```

5.4.1 Decompose Trends

The *diviner.PmdarimaAnalyzer.decompose_groups()* method will decompose each series into its component parts:

- trend
- seasonal
- random (also known as 'residuals')

The output of this method is a union of each group's decomposed trends in a single `DataFrame` that retains the group key information in columns along with the extracted components from the series data.

This method is mainly used for validation of a new project.

Example:

```
decomposed_trends = analyzer.decompose_groups(m=7, type="additive")
```

Arguments to the *diviner.PmdarimaAnalyzer.decompose_groups()* method:

m The frequency value of the endogenous series data. The integer supplied is a measure of the repeatable pattern of the estimated seasonality effect. For instance, `7` would be appropriate for daily measured data, `24` would be a good starting point for hourly data, and `52` would be a good initial validation value for weekly data.

type (`'type_'`) The type of decomposition to perform. One of: `"additive"` or `"multiplicative"`. A good rule of thumb for determining which of these to choose is to determine whether the seasonality effects either stay constant as a function of the trend (which would be "additive") or, if the seasonality effect is a function of the baseline trend value, "multiplicative" would be more appropriate. For further explanation, see here.

filter (`'filter_'`) *[Optional]* Reverse-sorted Array for performing convolution on the coefficients of either the `MA` terms or the `AR` terms.

Default: `None`

5.4.2 Calculate Differencing Term

Isolating the differencing term `'d'` can provide significant performance improvements if `AutoARIMA` is used as the underlying estimator for each series. This method provides a means of estimating these per-group differencing terms.

The output is returned as a dictionary of `{<group_key>: d}`

Note: This utility method is intended to be used as an input to the *diviner.GroupedPmdarima.fit()* method when using `AutoARIMA` as a base group estimator. It will set *per-group* values of `d` so that the AutoARIMA optimizer does not need to search for values of the differencing term, saving a great deal of computation time.

Example:

```
diffs = analyzer.calculate_ndiffs(alpha=0.1, test="kpss", max_d=5)
```

Arguments to the *diviner.PmdarimaAnalyzer.calculate_ndiffs()* method:

alpha The significance value used in determining if a pvalue for a test of an estimated `d` term is significant or not. Default: `0.05`

test The stationarity unit test used to determine significance for a tested `d` term.

Allowable values:

- "kpss"
- "pp"
- "adf"

Default: `"kpss"`

max_d The maximum allowable differencing term to test. Default: `2`

5.4.3 Calculate Seasonal Differencing Term

Isolating the seasonal differencing term `D` can provide a significant performance improvement to seasonal models which are activated by setting the `m` term in the base group estimator. The functionality of this *diviner.PmdarimaAnalyzer.calculate_nsdiffs()* method is similar to that of `calculate_ndiffs`, except for the seasonal differencing term.

Example:

```
seasonal_diffs = analyzer.calculate_nsdiffs(m=7, test="ocsb", max_D=5)
```

Arguments to the `calculate_nsdiffs` method:

m The frequency of seasonal periods within the endogenous series. The integer supplied is a measure of the repeatable pattern of the estimated seasonality effect. For instance, `7` would be appropriate for daily measured data, `24` would be a good starting point for hourly data, and `52` would be a good initial validation value for weekly data.

test The seasonality unit test used to determine an optimal seasonal differencing `D` term.

Allowable tests:

- "ocsb"
- "ch"

Default: `"ocsb"`

max_D The maximum allowable seasonal differencing term to test.

Default: `2`

5.4.4 Calculate Constancy

The constancy check is a data set utility validation tool that operates on each grouped series, determining whether or not it can be modeled.

The output of this validation check method *diviner.PmdarimaAnalyzer.calculate_is_constant()* is a dictionary of `{<group_key>: <Boolean constancy check>}`. Any group with a `True` result **is ineligible for modeling** as this indicates that the group has only a single constant value for each datetime period.

Example:

```
constancy_checks = analyzer.calculate_is_constant()
```

5.4.5 Calculate Auto Correlation Function

The *diviner.PmdarimaAnalyzer.calculate_acf()* method is used for calculating the auto-correlation function for each series group. The auto-correlation function values can be used (in conjunction with the result of partial auto-correlation function results) to select restrictive search values for the ordering terms for `AutoARIMA` or to manually set the ordering terms (`(p, d, q)`) for `ARIMA`.

Note: The general rule to determine whether to use an AR, MA, or ARMA configuration for `ARIMA` or `AutoARIMA` is as follows:

- ACF gradually trend to significance, PACF significance achieved after 1 lag -> AR model
- ACF significance after 1 lag, PACF gradually trend to significance -> MA model
- ACF gradually trend to significance, PACF gradually trend to significance -> ARMA model

These results can help to set the order terms of an ARIMA model (p and q) or, for AutoARIMA, set restrictions on maximum search space terms to assist in faster optimization of the model.

Arguments to the `calculate_acf` method:

unbiased auto-covariance denominator flag with values of:

- `True` -> denominator = `n - k`
- `False` -> denominator = `n`

nlags The number of auto-correlation lags to calculate and return.

Default: `40`

qstat Boolean flag to calculate and return the Q statistic from the Ljung-Box test.

Default: `False`

fft Whether to perform a fast fourier transformation of the series to calculate the auto-correlation function. For large time series, it is highly recommended to set this to `True`. Allowable values: `True`, `False`, or `None`.

Default: `None`

alpha If specified as a float, calculates and returns confidence intervals at this certainty level for the auto-correlation function values. For example, if alpha=0.1, 90% confidence intervals are calculated and returned wherein the standard deviation is computed according to Bartlett's formula.

Default: `None`

missing Handling of `NaN` values in series data. Available options are:

- `None` - no validation checks are performed.
- `'raise'` - an Exception is raised if a missing value is detected.
- `'conservative'` - `NaN` values are removed from the mean and cross-product calculations but are not removed from the series data.
- `'drop'` - `NaN` values are removed from the series data.

Default: `None`

adjusted Deprecation handler for the underlying `statsmodels` arguments that have become the `unbiased` argument. This is a duplicated value for the denominator mode of calculation for the autocovariance of the series.

Default: `False`

5.4.6 Calculate Partial Auto Correlation Function

The *`diviner.PmdarimaAnalyzer.calculate_pacf()`* method is used for determining the partial auto-correlation function for each series group. When combined with *Calculate Auto Correlation Function* results, ordering values can be estimated (or controlled in search space scope for `AutoARIMA`). See the notes in *Calculate Auto Correlation Function* for how to use the results from these two methods.

Arguments to the `calculate_pacf` method:

nlags The number of partial auto-correlation lags to calculate and return.

Default: `40`

method The method employed for calculating the partial auto-correlation function. Methods and their explanations are listed in the pmdarima docs.

Default: `'ywadjusted'`

alpha If specified as a float, calculates and returns confidence intervals at this certainty level for the auto-correlation function values. For example, if alpha=0.1, 90% confidence intervals are calculated and returned wherein the standard deviation is computed according to Bartlett's formula.

Default: `None`

5.4.7 Generate Diff

The utility method *`diviner.PmdarimaAnalyzer.generate_diff()`* will generate lag differences for each group. While not applicable to most timeseries modeling problems, it can prove to be useful in certain situations or as a diagnostic tool to investigate why a particular series is not fitting properly.

Arguments for this method:

lag The magnitude of the lag used in calculating the differencing. Default: `1`

differences The order of the differencing to be performed. Default: `1`

For an illustrative example, see the diff example.

5.4.8 Generate Diff Inversion

The utility method *diviner.PmdarimaAnalyzer.generate_diff_inversion()* will invert a previously differenced grouped series.

Arguments for this method:

group_diff_data The differenced data from the usage of *diviner.PmdarimaAnalyzer.generate_diff()*.

lag The magnitude of the lag that was used in the differencing function in order to revert the diff.

Default: `1`

differences The order of the differencing that was performed using *diviner.PmdarimaAnalyzer.generate_diff()* so that the series data can be reverted.

Default: `1`

recenter If `True` and `'series_start'` exists in `group_diff_data` dict, will restore the original series range for each group based on the series start value calculated through the `generate_diff()` method. If the `group_diff_data` does not contain the starting values, the data will not be re-centered.

Default: `False`

5.5 Class Signature of PmdarimaAnalyzer

`class diviner.PmdarimaAnalyzer(`*df*, *group_key_columns*, *y_col*, *datetime_col*`)`

`calculate_acf(`*unbiased=False*, *nlags=None*, *qstat=False*, *fft=None*, *alpha=None*, *missing='none'*, *adjusted=False*`)`

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility for calculating the autocorrelation function for each group. Combined with a partial autocorrelation function calculation, the return values can greatly assist in setting AR, MA, or ARMA terms for a given model.

The general rule to determine whether to use an AR, MA, or ARMA configuration for ARIMA (or AutoARIMA) is as follows:

- ACF gradually trend to significance, PACF significance achieved after 1 lag -> AR model
- ACF significance after 1 lag, PACF gradually trend to significance -> MA model
- ACF gradually trend to significance, PACF gradually trend to significance -> ARMA model

These results can help to set the order terms of an ARIMA model (p and q) or, for AutoARIMA, set restrictions on maximum search space terms to assist in faster optimization of the model.

Parameters

- **unbiased** – Boolean flag that sets the autocovariance denominator to `'n-k'` if `True` and `n` if `False`.

  Note: This argument is deprecated and removed in versions of pmdarima > 2.0.0

  Default: `False`

- **nlags** – The count of autocorrelation lags to calculate and return.

  Default: `40`

- **qstat** – Boolean flag to calculate and return the Ljung-Box statistic for each lag.

  Default: `False`

- **fft** – Boolean flag for whether to use fast fourier transformation (fft) for computing the autocorrelation function. FFT is recommended for large time series data sets.

  Default: `None`

- **alpha** – If specified, calculates and returns the confidence intervals for the acf values at the level set (i.e., for 90% confidence, an alpha of 0.1 would be set)

  Default: `None`

- **missing** – handling of NaN values in the series data.

  Available options:

  `['none', 'raise', 'conservative', 'drop']`.

  `none`: no checks are performed.

  `raise`: an Exception is raised if NaN values are in the series.

  `conservative`: the autocovariance is calculated by removing NaN values from the mean and cross-product calculations but are not eliminated from the series.

  `drop`: `NaN` values are removed from the series and adjacent values to `NaN`'s are treated as contiguous (which may invalidate the results in certain situations).

  Default: `'none'`

- **adjusted** – Deprecation handler for the underlying `statsmodels` arguments that have become the `unbiased` argument. This is a duplicated value for the denominator mode of calculation for the autocovariance of the series.

Returns Dictionary of `{<group_key>: {<acf terms>: <values as array>}}`

calculate_is_constant()

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility method for determining whether or not a series is composed of all of the same elements or not. (e.g. a series of {1, 2, 3, 4, 5, 1, 2, 3} will return 'False', while a series of {1, 1, 1, 1, 1, 1, 1, 1, 1} will return 'True')

Returns Dictionary of `{<group_key>: <Boolean constancy check>}`

calculate_ndiffs(*alpha=0.05*, *test='kpss'*, *max_d=2*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility method for determining the optimal `d` value for ARIMA ordering. Calculating this as a fixed value can dramatically increase the tuning time for `pmdarima` models.

Parameters

- **alpha** – significance level for determining if a pvalue used for testing a value of `'d'` is significant or not.

  Default: `0.05`

- **test** – Type of unit test for stationarity determination to use. Supported values: `['kpss', 'adf', 'pp']` See:

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.KPSSTest.html#pmdarima.arima.KPSSTest

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.PPTest.html#pmdarima.arima.PPTest

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.ADFTest.html#pmdarima.arima.ADFTest

  Default: `'kpss'`

- **max_d** – The max value for `d` to test.

Returns Dictionary of `{<group_key>: <optimal 'd' value>}`

calculate_nsdiffs(*m*, *test='ocsb'*, *max_D=2*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility method for determining the optimal `D` value for seasonal `SARIMAX` ordering of `('P', 'D', 'Q')`.

Parameters

- **m** – The number of seasonal periods in the series.
- **test** – Type of unit test for seasonality. Supported tests: `['ocsb', 'ch']` See:

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.OCSBTest.html#pmdarima.arima.OCSBTest

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.CHTest.html#pmdarima.arima.CHTest

  Default: `'ocsb'`

- **max_D** – Maximum number of seasonal differences to test for.

  Default: 2

Returns Dictionary of `{<group_key>: <optimal 'D' value>}`

calculate_pacf(*nlags=None*, *method='ywadjusted'*, *alpha=None*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility for calculating the partial autocorrelation function for each group. In conjunction with the autocorrelation function `calculate_acf`, the values returned from a pacf calculation can assist in setting values or bounds on AR, MA, and ARMA terms for an ARIMA model.

The general rule to determine whether to use an AR, MA, or ARMA configuration for `ARIMA` (or `AutoARIMA`) is as follows:

- ACF gradually trend to significance, PACF significance achieved after 1 lag -> AR model
- ACF significance after 1 lag, PACF gradually trend to significance -> MA model

- ACF gradually trend to significance, PACF gradually trend to significance -> ARMA model

These results can help to set the order terms of an ARIMA model (`p` and `q`) or, for `AutoARIMA`, set restrictions on maximum search space terms to assist in faster optimization of the model.

Parameters

- **`nlags`** – The count of partial autocorrelation lags to calculate and return.

  Default: `40`

- **`method`** – The method used for pacf calculation. See the `pmdarima` docs for full listing of methods:

  https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.utils.pacf.html

  Default: `'ywadjusted'`

- **`alpha`** – If specified, returns confidence intervals based on the alpha value supplied.

  Default: `None`

Returns Dictionary of `{<group_key>: {<pacf terms>: <values as array>}}`

`decompose_groups`(*m*, *type_*, *filter_=None*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

Utility method that wraps `pmdarima.arima.decompose()` for each group within the passed-in DataFrame. Note: decomposition works best if the total number of entries within the series being decomposed is a multiple of the *m* parameter value.

Parameters

- **`m`** – The frequency of the endogenous series. (i.e., for daily data, an `m` value of `'7'` would be appropriate for estimating a weekly seasonality, while setting `m` to `'365'` would be effective for yearly seasonality effects.)

- **`type`** – The type of decomposition to perform. One of: `['additive', 'multiplicative']`

  See: https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.decompose.html

- **`filter`** – Optional Array for performing convolution. This is specified as a filter for coefficients (the Moving Average and/or Auto Regressor coefficients) in reverse time order in order to filter out a seasonal component.

  Default: None

Returns Pandas DataFrame with the decomposed trends for each group.

`generate_diff`(*lag=1*, *differences=1*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

A utility for generating the array diff (lag differences) for each group. To support invertability, this method will return the starting value of each array as well as the differenced values.

Parameters

- **lag** – Determines the magnitude of the lag to calculate the differencing function for.

  Default: 1

- **differences** – The order of the differencing to be performed. Note that values > 1 will generate n fewer results.

  Default: 1

Returns Dictionary of `{<group_key>: {"series_start": <float>, "diff": <diff_array>}}`

static generate_diff_inversion(*group_diff_data*, *lag=1*, *differences=1*, *recenter=False*)

Note: Experimental: This method may change, be moved, or removed in a future release with no prior warning.

A utility for inverting a previously differenced group of timeseries data. This utility supports returning each group's series data to the original range of the data if the recenter argument is set to *True* and the start conditions are contained within the `group_diff_data` argument's dictionary structure.

Parameters

- **group_diff_data** – Differenced payload consisting of a dictionary of `{<group_key>: {'diff': <differenced data>, [optional]'series_start': float}}`
- **lag** – The lag to use to perform the differencing inversion.

  Default: 1

- **differences** – The order of differencing to be used during the inversion.

  Default: 1

- **recenter** – If `True` and `'series_start'` exists in `group_diff_data` dict, will restore the original series range for each group based on the series start value calculated through the `generate_diff()` method. If the `group_diff_data` does not contain the starting values, the data will not be re-centered.

  Default: `False`

Returns Dictionary of `{<group_key>: <series_inverted_data>}`

CHAPTER

SIX

DATA PROCESSING

The data manipulation APIs are a key component of the utility of this library. While they are largely obfuscated by the main entry point APIs for each forecasting library, they can be useful for custom implementations and for performing validation of source data sets.

6.1 Pandas DataFrame Group Processing API

Class signature:

class diviner.data.pandas_group_generator.**PandasGroupGenerator**(*group_key_columns: Tuple*, *datetime_col: str*, *y_col: str*)

This class is used to convert a normalized collection of time series data within a single `DataFrame`, e.g.:

region	zone	ds	y
'northeast'	1	"2021-10-01"	1234.5
'northeast'	2	"2021-10-01"	3255.6
'northeast'	1	"2021-10-02"	1255.9

With the grouping keys `['region', 'zone']` define the unique series of the target `y` indexed by `ds`.

This class will

1. Generate a *master group key* that is a tuple zip of the grouping key arguments specified by the user, preserving the order of declaration of these keys.
2. Group the `DataFrame` by these master grouping keys and generate a collection of tuples of the form `(master_grouping_key, <series DataFrame>)` which is used for iterating over to generate the individualized forecasting models for each master key group.

__init__(*group_key_columns: Tuple*, *datetime_col: str*, *y_col: str*)

Parameters

- **group_key_columns** – Grouping columns that a combination of which designates a combination of `ds` and `y` that represent a distinct series.
- **datetime_col** – The name of the column that contains the `datetime` values for each series.
- **y_col** – The endogenous regressor element of the series. This is the value that is used for training and is the element that is intending to be forecast.

_get_df_with_master_key_column(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Method for creating the 'master_group_key' column that defines a unique group. The master_group_key column is generated from the concatenation (within a tuple) of the values in each of the individual *_group_key_columns*, serving as an aggregation grouping key to define a unique collection of datetime series values. For example:

region	zone	ds	y
'northeast'	1	"2021-10-01"	1234.5
'northeast'	2	"2021-10-01"	3255.6
'northeast'	1	"2021-10-02"	1255.9

With the above dataset, the `group_key_columns` passed in would be: `('region', 'zone')` This method will modify the input `DataFrame` by adding the `master_group_key` as follows:

region	zone	ds	y	grouping_key
'northeast'	1	"2021-10-01"	1234.5	('northeast', 1)
'northeast'	2	"2021-10-01"	3255.6	('northeast', 2)
'northeast'	1	"2021-10-02"	1255.9	('northeast', 1)

Parameters **df** – The normalized `DataFrame`

Returns A copy of the passed-in `DataFrame` with a master grouping key column added that contains the group definitions per row of the input `DataFrame`.

generate_prediction_groups(*df: pandas.core.frame.DataFrame*)

Method for generating the data set collection required to run a manual per `datetime` prediction for arbitrary datetime and key groupings.

Parameters **df** – Normalized `DataFrame` that contains the columns defined in instance attribute `_group_key_columns` within its schema and the dates for prediction within the `datetime_col` field.

Returns `List(tuple(master_group_key, df))` the processing collection of `DataFrame` coupled with their group identifier.

generate_processing_groups(*df: pandas.core.frame.DataFrame*)

Method for generating the collection of `[(master_grouping_key, <group DataFrame>)]`

This method will call `_create_master_key_column()` to generate a column containing the tuple of the values within the `_group_key_columns` fields, then generate an iterable collection of `key` -> `DataFrame` representation.

For example, after adding the `grouping_key` column from `_create_master_key_column()`, the `DataFrame` will look like this

region	zone	ds	y	grouping_key
'northeast'	1	"2021-10-01"	1234.5	('northeast', 1)
'northeast'	2	"2021-10-01"	3255.6	('northeast', 2)
'northeast'	1	"2021-10-02"	1255.9	('northeast', 1)

This method will translate this structure to

```
[(('northeast', 1),
```

ds	y
“2021-10-01”	1234.5
“2021-10-02”	1255.9

```
), (('northeast', 2),
```

ds	y
“2021-10-01”	3255.6
“2021-10-02”	1255.9

```
)]
```

Parameters **df** – Normalized `DataFrame` that contains the columns defined in instance attribute `_group_key_columns` within its schema.

Returns `List(tuple(master_group_key, df))` the processing collection of `DataFrame` coupled with their group identifier.

6.2 Developer API for Data Processing

Abstract Base Class for grouped processing of a fully normalized `DataFrame` :

Abstract Base Class for defining the API contract for group generator operations. This base class is a template for package-specific implementations that function to convert a normalized representation of grouped time series into per-group collections of discrete time series so that forecasting models can be trained on each group.

class diviner.data.base_group_generator.**BaseGroupGenerator**(*group_key_columns: Tuple*, *datetime_col: str*, *y_col: str*)

Abstract class for defining the basic elements of performing a group processing collection generation operation.

__init__(*group_key_columns: Tuple*, *datetime_col: str*, *y_col: str*)

Grouping key columns must be defined to serve in the construction of a consolidated single unique key that is used to identify a particular unique time series. The unique combinations of these provided fields define and control the grouping of univariate series data in order to train (fit) a particular model upon each of the unique series (that are defined by the combination of the values within these supplied columns).

The primary purpose of the children of this class is to generate a dictionary of: `{<group_key> : <DataFrame with unique univariate series>}`. The `` `group_key `` element is constructed as a tuple of the values within the columns specified by `_group_key_columns` in this class constructor.

For example, with a normalized data set provided of:

ds	y	group1	group2
2021-09-02	11.1	“a”	“z”
2021-09-03	7.33	“a”	“z”
2021-09-02	31.1	“b”	“q”
2021-09-03	44.1	“b”	“q”

There are two separate univariate series: `("a", "z")` and `("b", "q")`. The group generator’s function is to convert this unioned `DataFrame` into the following:

```
{ ("a", "z"):
```

ds	y	group1	group2
2021-09-02	11.1	“a”	“z”
2021-09-03	7.33	“a”	“z”

,("b", "q"):

ds	y	group1	group2
2021-09-02	31.1	“b”	“q”
2021-09-03	44.1	“b”	“q”

}

This grouping allows for a model to be fit to each of these series in isolation.

Parameters

- **group_key_columns** – Tuple[str] of column names that determine which elements of the submitted DataFrame determine uniqueness of a particular time series.
- **datetime_col** – The name of the column that contains the datetime values for each series.
- **y_col** – The endogenous regressor element of the series. This is the value that is used for training and is the element that is intending to be forecast.

__weakref__
list of weak references to the object (if defined)

abstract generate_prediction_groups(*df*)
Abstract method for generating the data set collection required for manual prediction for arbitrary datetime and key groupings.

Parameters **df** – Normalized DataFrame that contains the columns defined in instance attribute _group_key_columns within its schema and the dates for prediction within the datetime_col field.

Returns List(tuple(master_group_key, df)) the processing collection of DataFrame coupled with their group identifier.

abstract generate_processing_groups(*df*)
Abstract method for the generation of processing execution groups for individual models. Implementations of this method should generate a processing collection that is a relation between the unique combinations of _group_key_columns values, generated as a _master_group_key entry that defines a specific datetime series for forecasting.

For example, with a normalized dataframe input of

ds	region	country	y
2020-01-01	SW	USA	42
2020-01-02	SW	USA	11
2020-01-01	NE	USA	31
2020-01-01	Ontario	CA	12

The output structure should be, with the group_keys value specified as:

("country", "region"):[{ ("USA", "SW"):

ds	region	country	y
2020-01-01	SW	USA	42
2020-01-02	SW	USA	11

}. {("USA", "NE"):

ds	region	country	y
2020-01-01	NE	USA	31

}, {("CA", "Ontario"):

ds	region	country	y
2020-01-01	Ontario	CA	12

}]

The list wrapper around dictionaries is to allow for multiprocessing support without having to contend with encapsulating the entire dictionary for the processing of a single key and value pair.

Parameters **df** – The user-input normalized DataFrame with _group_key_columns

Returns A list of dictionaries of `{group_key: <group's univariate series data>}` structure for isolated processing by the model APIs.

CHAPTER

SEVEN

CONTRIBUTING GUIDE

We happily welcome contributions to the Diviner package. We use GitHub Issues to track community reported issues and GitHub Pull Requests for accepting changes. Contributions are licensed on a license-in/license-out basis.

7.1 Communication

Before starting work on a major feature, please reach out to us via GitHub by filing an issue. We will make sure no one else is already working on it and ask you to open a GitHub issue. A "major feature" is defined as any change that is > 100 LOC altered (not including tests), or changes any user-facing behavior, API signature, or introduces a new wrapped grouped model implementation. We will use the GitHub issue to discuss the feature and come to an agreement on the design approach and help to answer any questions you may have on implementation details or designs. The GitHub review process for major features is also important so that organizations with commit access can come to agreement on design. If it is appropriate to write a design document, the document must be hosted either in the GitHub tracking issue, or linked to from the issue and hosted in a world-readable location. Small patches, examples, documentation updates, and bug fixes don't need prior communication and can be submitted as a PR directly for review.

7.2 Coding Style

We follow PEP 8 with one exception: lines can be up to 100 characters in length, not 79.

7.3 Code formatting

We use the Python formatting plugin Black to ensure that code formatting throughout this package is consistent. To ensure that any submissions are consistent, ensure that you have installed the required version in dev-requirements. Prior to committing changes to your local branch, simply run `black .` from the root of the repository. It is also strongly recommended to run pylint prior to commits to minimize the number of requested changes you may see in a pull request. To install all required code formatting tools, simply run `pip install -r "dev-requirements.txt"` from within your containerized development environment to ensure that you have the required versions of all tools.

7.4 Sign your work

The sign-off is a simple line at the end of the explanation for the patch. Your signature certifies that you wrote the patch or otherwise have the right to pass it on as an open-source patch. The rules are pretty simple: if you can certify the below (from developercertificate.org):

```
Developer Certificate of Origin
Version 1.1

Copyright (C) 2004, 2006 The Linux Foundation and its contributors.
1 Letterman Drive
Suite D4700
San Francisco, CA, 94129

Everyone is permitted to copy and distribute verbatim copies of this
license document, but changing it is not allowed.


Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the best
    of my knowledge, is covered under an appropriate open source
    license and I have the right under that license to submit that
    work with modifications, whether created in whole or in part
    by me, under the same open source license (unless I am
    permitted to submit under a different license), as indicated
    in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including all
    personal information I submit with it, including my sign-off) is
    maintained indefinitely and may be redistributed consistent with
    this project or the open source license(s) involved.
```

Then you just add a line to every git commit message as shown below.

```
Signed-off-by: Joe Smith <joe.smith@email.com>
Use your real name (sorry, no pseudonyms or anonymous contributions.)
```

If you set your `user.name` and `user.email` git configs, you can sign your commit automatically with: `git commit -s`.

CHAPTER

EIGHT

CHANGELOG

8.1 0.1.1 - Support pmdarima 2.x

Diviner 0.1.1 consists of minor changes to support functionality with the recently released pmdarima 2.x suite.

8.2 0.1.0 - Initial release

Diviner 0.1.0 is the initial release of the grouped forecasting API.

Features:

- Introduced GroupedProphet API (#1, @BenWilson2)
- Introduced GroupedPmdarima API (#3, @BenWilson2)

PYTHON MODULE INDEX

d

INDEX

Symbols

B

C

D

E

F

G

L

M

P

S