04-Choosing-ARIMA-Orders

October 31, 2021

1 Choosing ARIMA Orders

- Goals
  - Understand PDQ terms for ARIMA (slides)
  - Understand how to choose orders manually from ACF and PACF
  - Understand how to use automatic order selection techniques using the functions below

Before we can apply an ARIMA forecasting model, we need to review the components of one. ARIMA, or Autoregressive Independent Moving Average is actually a combination of 3 models: * AR(p) Autoregression - a regression model that utilizes the dependent relationship between a current observation and observations over a previous period. * I(d) Integration - uses differencing of observations (subtracting an observation from an observation at the previous time step) in order to make the time series stationary * MA(q) Moving Average - a model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Related Functions:

pmdarima.auto_arima(y[,start_p,d,start_q, ...]) Returns the optimal order for an ARIMA model

Optional Function (see note below):

stattools.arma_order_select_ic(y[, max_ar, ...]) Returns information criteria for many ARMA models x13.x13_arima_select_order(endog[, ...]) Perform automatic seasonal ARIMA order identification using x12/x13 ARIMA

[1]:
```
import pandas as pd
import numpy as np
from pmdarima import auto_arima
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

[3]:
```
# Load a non-stationary dataset
df1 = pd.read_csv('../Data/airline_passengers.
 ↪csv',index_col='Month',parse_dates=True)
df1.index.freq = 'MS'

# Load a stationary dataset
```

```
df2 = pd.read_csv('../Data/DailyTotalFemaleBirths.
 ↪csv',index_col='Date',parse_dates=True)
df2.index.freq = 'D'
```

[4]:
```
help(auto_arima)
```

```
Help on function auto_arima in module pmdarima.arima.auto:

auto_arima(y, X=None, start_p=2, d=None, start_q=2, max_p=5, max_d=2, max_q=5,
start_P=1, D=None, start_Q=1, max_P=2, max_D=1, max_Q=2, max_order=5, m=1,
seasonal=True, stationary=False, information_criterion='aic', alpha=0.05,
test='kpss', seasonal_test='ocsb', stepwise=True, n_jobs=1, start_params=None,
trend=None, method='lbfgs', maxiter=50, offset_test_args=None,
seasonal_test_args=None, suppress_warnings=True, error_action='trace',
trace=False, random=False, random_state=None, n_fits=10,
return_valid_fits=False, out_of_sample_size=0, scoring='mse', scoring_args=None,
with_intercept='auto', sarimax_kwargs=None, **fit_args)
    Automatically discover the optimal order for an ARIMA model.

    The auto-ARIMA process seeks to identify the most optimal
    parameters for an ``ARIMA`` model, settling on a single fitted ARIMA model.
    This process is based on the commonly-used R function,
    ``forecast::auto.arima`` [3].

    Auto-ARIMA works by conducting differencing tests (i.e.,
    Kwiatkowski-Phillips-Schmidt-Shin, Augmented Dickey-Fuller or
    Phillips-Perron) to determine the order of differencing, ``d``, and then
    fitting models within ranges of defined ``start_p``, ``max_p``,
    ``start_q``, ``max_q`` ranges. If the ``seasonal`` optional is enabled,
    auto-ARIMA also seeks to identify the optimal ``P`` and ``Q`` hyper-
    parameters after conducting the Canova-Hansen to determine the optimal
    order of seasonal differencing, ``D``.

    In order to find the best model, auto-ARIMA optimizes for a given
    ``information_criterion``, one of ('aic', 'aicc', 'bic', 'hqic', 'oob')
    (Akaike Information Criterion, Corrected Akaike Information Criterion,
    Bayesian Information Criterion, Hannan-Quinn Information Criterion, or
    "out of bag"--for validation scoring--respectively) and returns the ARIMA
    which minimizes the value.

    Note that due to stationarity issues, auto-ARIMA might not find a
    suitable model that will converge. If this is the case, a ``ValueError``
    will be thrown suggesting stationarity-inducing measures be taken prior
    to re-fitting or that a new range of ``order`` values be selected. Non-
    stepwise (i.e., essentially a grid search) selection can be slow,
    especially for seasonal data. Stepwise algorithm is outlined in Hyndman and
    Khandakar (2008).
```

```
Parameters
----------
y : array-like or iterable, shape=(n_samples,)
    The time-series to which to fit the ``ARIMA`` estimator. This may
    either be a Pandas ``Series`` object (statsmodels can internally
    use the dates in the index), or a numpy array. This should be a
    one-dimensional array of floats, and should not contain any
    ``np.nan`` or ``np.inf`` values.

X : array-like, shape=[n_obs, n_vars], optional (default=None)
    An optional 2-d array of exogenous variables. If provided, these
    variables are used as additional features in the regression
    operation. This should not include a constant or trend. Note that
    if an ``ARIMA`` is fit on exogenous features, it must be provided
    exogenous features for making predictions.

start_p : int, optional (default=2)
    The starting value of ``p``, the order (or number of time lags)
    of the auto-regressive ("AR") model. Must be a positive integer.

d : int, optional (default=None)
    The order of first-differencing. If None (by default), the value
    will automatically be selected based on the results of the ``test``
    (i.e., either the Kwiatkowski-Phillips-Schmidt-Shin, Augmented
    Dickey-Fuller or the Phillips-Perron test will be conducted to find
    the most probable value). Must be a positive integer or None. Note
    that if ``d`` is None, the runtime could be significantly longer.

start_q : int, optional (default=2)
    The starting value of ``q``, the order of the moving-average
    ("MA") model. Must be a positive integer.

max_p : int, optional (default=5)
    The maximum value of ``p``, inclusive. Must be a positive integer
    greater than or equal to ``start_p``.

max_d : int, optional (default=2)
    The maximum value of ``d``, or the maximum number of non-seasonal
    differences. Must be a positive integer greater than or equal to ``d``.

max_q : int, optional (default=5)
    The maximum value of ``q``, inclusive. Must be a positive integer
    greater than ``start_q``.

start_P : int, optional (default=1)
    The starting value of ``P``, the order of the auto-regressive portion
    of the seasonal model.
```

```
D : int, optional (default=None)
    The order of the seasonal differencing. If None (by default, the value
    will automatically be selected based on the results of the
    ``seasonal_test``. Must be a positive integer or None.

start_Q : int, optional (default=1)
    The starting value of ``Q``, the order of the moving-average portion
    of the seasonal model.

max_P : int, optional (default=2)
    The maximum value of ``P``, inclusive. Must be a positive integer
    greater than ``start_P``.

max_D : int, optional (default=1)
    The maximum value of ``D``. Must be a positive integer greater
    than ``D``.

max_Q : int, optional (default=2)
    The maximum value of ``Q``, inclusive. Must be a positive integer
    greater than ``start_Q``.

max_order : int, optional (default=5)
    Maximum value of p+q+P+Q if model selection is not stepwise.
    If the sum of ``p`` and ``q`` is >= ``max_order``, a model will
    *not* be fit with those parameters, but will progress to the next
    combination. Default is 5. If ``max_order`` is None, it means there
    are no constraints on maximum order.

m : int, optional (default=1)
    The period for seasonal differencing, ``m`` refers to the number of
    periods in each season. For example, ``m`` is 4 for quarterly data, 12
    for monthly data, or 1 for annual (non-seasonal) data. Default is 1.
    Note that if ``m`` == 1 (i.e., is non-seasonal), ``seasonal`` will be
    set to False. For more information on setting this parameter, see
    :ref:`period`.

seasonal : bool, optional (default=True)
    Whether to fit a seasonal ARIMA. Default is True. Note that if
    ``seasonal`` is True and ``m`` == 1, ``seasonal`` will be set to False.

stationary : bool, optional (default=False)
    Whether the time-series is stationary and ``d`` should be set to zero.

information_criterion : str, optional (default='aic')
    The information criterion used to select the best ARIMA model. One of
    ``pmdarima.arima.auto_arima.VALID_CRITERIA``, ('aic', 'bic', 'hqic',
    'oob').
```

```
alpha : float, optional (default=0.05)
    Level of the test for testing significance.

test : str, optional (default='kpss')
    Type of unit root test to use in order to detect stationarity if
    ``stationary`` is False and ``d`` is None. Default is 'kpss'
    (Kwiatkowski-Phillips-Schmidt-Shin).

seasonal_test : str, optional (default='ocsb')
    This determines which seasonal unit root test is used if ``seasonal``
    is True and ``D`` is None. Default is 'OCSB'.

stepwise : bool, optional (default=True)
    Whether to use the stepwise algorithm outlined in Hyndman and Khandakar
    (2008) to identify the optimal model parameters. The stepwise algorithm
    can be significantly faster than fitting all (or a ``random`` subset
    of) hyper-parameter combinations and is less likely to over-fit
    the model.

n_jobs : int, optional (default=1)
    The number of models to fit in parallel in the case of a grid search
    (``stepwise=False``). Default is 1, but -1 can be used to designate
    "as many as possible".

start_params : array-like, optional (default=None)
    Starting parameters for ``ARMA(p,q)``.  If None, the default is given
    by ``ARMA._fit_start_params``.

method : str, optional (default='lbfgs')
    The ``method`` determines which solver from ``scipy.optimize``
    is used, and it can be chosen from among the following strings:

    - 'newton' for Newton-Raphson
    - 'nm' for Nelder-Mead
    - 'bfgs' for Broyden-Fletcher-Goldfarb-Shanno (BFGS)
    - 'lbfgs' for limited-memory BFGS with optional box constraints
    - 'powell' for modified Powell's method
    - 'cg' for conjugate gradient
    - 'ncg' for Newton-conjugate gradient
    - 'basinhopping' for global basin-hopping solver

    The explicit arguments in ``fit`` are passed to the solver,
    with the exception of the basin-hopping solver. Each
    solver has several optional arguments that are not the same across
    solvers. These can be passed as **fit_kwargs

trend : str or None, optional (default=None)
```

```
    The trend parameter. If ``with_intercept`` is True, ``trend`` will be
    used. If ``with_intercept`` is False, the trend will be set to a no-
    intercept value.

maxiter : int, optional (default=50)
    The maximum number of function evaluations. Default is 50.

offset_test_args : dict, optional (default=None)
    The args to pass to the constructor of the offset (``d``) test. See
    ``pmdarima.arima.stationarity`` for more details.

seasonal_test_args : dict, optional (default=None)
    The args to pass to the constructor of the seasonal offset (``D``)
    test. See ``pmdarima.arima.seasonality`` for more details.

suppress_warnings : bool, optional (default=True)
    Many warnings might be thrown inside of statsmodels. If
    ``suppress_warnings`` is True, all of the warnings coming from
    ``ARIMA`` will be squelched. Note that this will not suppress
    UserWarnings created by bad argument combinations.

error_action : str, optional (default='warn')
    If unable to fit an ``ARIMA`` for whatever reason, this controls the
    error-handling behavior. Model fits can fail for linear algebra errors,
    convergence errors, or any number of problems related to stationarity
    or input data.
        - 'warn': Warns when an error is encountered (default)
        - 'raise': Raises when an error is encountered
        - 'ignore': Ignores errors (not recommended)
        - 'trace': Logs the entire error stacktrace and continues the
           search. This is the best option when trying to determine why a
           model is failing.

trace : bool or int, optional (default=False)
    Whether to print status on the fits. A value of False will print no
    debugging information. A value of True will print some. Integer values
    exceeding 1 will print increasing amounts of debug information at each
    fit.

random : bool, optional (default=False)
    Similar to grid searches, ``auto_arima`` provides the capability to
    perform a "random search" over a hyper-parameter space. If ``random``
    is True, rather than perform an exhaustive search or ``stepwise``
    search, only ``n_fits`` ARIMA models will be fit (``stepwise`` must be
    False for this option to do anything).

random_state : int, long or numpy ``RandomState``, optional (default=None)
    The PRNG for when ``random=True``. Ensures replicable testing and
```

```
    results.

n_fits : int, optional (default=10)
    If ``random`` is True and a "random search" is going to be performed,
    ``n_iter`` is the number of ARIMA models to be fit.

return_valid_fits : bool, optional (default=False)
    If True, will return all valid ARIMA fits in a list. If False (by
    default), will only return the best fit.

out_of_sample_size : int, optional (default=0)
    The ``ARIMA`` class can fit only a portion of the data if specified,
    in order to retain an "out of bag" sample score. This is the
    number of examples from the tail of the time series to hold out
    and use as validation examples. The model will not be fit on these
    samples, but the observations will be added into the model's ``endog``
    and ``exog`` arrays so that future forecast values originate from the
    end of the endogenous vector.

    For instance::

        y = [0, 1, 2, 3, 4, 5, 6]
        out_of_sample_size = 2

        > Fit on: [0, 1, 2, 3, 4]
        > Score on: [5, 6]
        > Append [5, 6] to end of self.arima_res_.data.endog values

scoring : str, optional (default='mse')
    If performing validation (i.e., if ``out_of_sample_size`` > 0), the
    metric to use for scoring the out-of-sample data. One of ('mse', 'mae')

scoring_args : dict, optional (default=None)
    A dictionary of key-word arguments to be passed to the ``scoring``
    metric.

with_intercept : bool or str, optional (default="auto")
    Whether to include an intercept term. Default is "auto" which behaves
    like True until a point in the search where the sum of differencing
    terms will explicitly set it to True or False.


sarimax_kwargs : dict or None, optional (default=None)
    Keyword arguments to pass to the ARIMA constructor.


**fit_args : dict, optional (default=None)
    A dictionary of keyword arguments to pass to the :func:`ARIMA.fit`
```

```
        method.

    See Also
    --------
    :func:`pmdarima.arima.ARIMA`

    Notes
    -----
    * Fitting with `stepwise=False` can prove slower, especially when
      `seasonal=True`.

    References
    ----------
    .. [1] https://wikipedia.org/wiki/Autoregressive_integrated_moving_average
    .. [2] R's auto-arima source code:
https://github.com/robjhyndman/forecast/blob/master/R/arima.R
    .. [3] R's auto-arima documentation:
https://www.rdocumentation.org/packages/forecast
```

[10]:
```
'''
So there are a lot of parameters you can pass in to the auto Auto Arima␣
 ↪function.

The most obvious ones should be what the starting point for each parameter is␣
 ↪and what the max value
for each parameter is.

And the first thing you pass in is the entirety of the data set that we want to␣
 ↪predict on

right now we're not actually concerned with a trained test split.
Instead, we're using AIC as our main criterion for judging what orders we␣
 ↪should be using

later on when we actually run into ARIMA based model and we want to evaluate␣
 ↪how well our forecasts will perform
on a test data set, that the model hasn't seen before, then we'll actually be␣
 ↪concerned with a trained
test split.

And then if you want, you can also pass in some starting P and Q values. so we␣
 ↪can say something like

start_p=0,essentially saying there was no AR component, which is unlikely given␣
 ↪this data.
We're going to say start_q=0
```

```
then we get to choose the max value.we can say max_p=6, max_q=3,

And another thing we're going to do is I'm going to go ahead and say seasonal =␣
 ↪false, since
we already ran some descriptive statistics and tests and we saw that this was a␣
 ↪stationary data set.

So I can tell a story to not worry about trying to find higher, more complex␣
 ↪models that fit any sort
of seasonality component.

And finally, what I'm also going to do is I'm going to set Trece equal to true␣
 ↪and this is not true
by default.

Trece will basically show you the first couple of Arima models that auto Arima␣
 ↪is trying to fit.


And what's really interesting here is you'll notice that we're going from P=0␣
 ↪all the way to max
of P=6 and Q =0 all the way to Q=3.

Now it's interesting if you think about this, if we have 6 possible values for␣
 ↪P and 3 possible
values for Q, then at a minimum we should be getting 6X3 or 18 different models␣
 ↪to search
for.


So something to note here is because Auto ARIMA is using AIC as its information␣
 ↪criterion.
Remember, AIC begins to punish more complex models due to their potential to␣
 ↪overfit the data.

So what's really nice about this auto arima function?
It's actually smart enough to understand that it's not going to need to check␣
 ↪all the way to Max P =6
or even Max Q=3.

Eventually, what it's going to realize is even as it begins raising the order␣
 ↪of P and Q higher and
higher, the AIC is essentially staying the same.
So at that point it's going to stop and not waste your time with trying to fit␣
 ↪every single model in
```

```
this grid search.


So we're going to go ahead and say, let's call this stepwise fit.
Since essentially performing a step wise by stepping through the different␣
 ↪combinations here.

And we specified traces equal to true, which will allow us to end up seeing the␣
 ↪different models that
ordinary men tried out


'''

'''
But you'll notice what it's happening is it's fitting these various Arima␣
 ↪models with various different
orders and then reporting back the AIC, the BIC, and then how long it took to␣
 ↪fit this particular model.


And what's really interesting here is notice that a lot of these actually have␣
 ↪extremely similar AIC
values.

And what happens is Auto ARIMA eventually figures out that it's no longer worth␣
 ↪it to continue increasing
the orders of some of these more complex models all the way to your max_P=6␣
 ↪value instead,
since these are essentially staying the same from, it's no longer worthit to␣
 ↪keep increasing AIC since that's
something we're looking to minimize.

So what we realize is probably the most balanced model ends up being ARIM(1,1,1)
'''
stepwise_fit=auto_arima(df2["Births"],
                        start_p=0,
                        start_q=0,
                        max_p=6,
                        max_q=3,
                        seasonal=False,
                        trace=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=2650.760, Time=0.02 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=2565.234, Time=0.04 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=2463.584, Time=0.06 sec
```

```
ARIMA(0,1,0)(0,0,0)[0]             : AIC=2648.768, Time=0.01 sec
ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=2460.154, Time=0.11 sec
ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=2461.271, Time=0.14 sec
ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.27 sec
ARIMA(0,1,2)(0,0,0)[0] intercept   : AIC=2460.722, Time=0.11 sec
ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=2536.154, Time=0.07 sec
ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=2463.050, Time=0.34 sec
ARIMA(1,1,1)(0,0,0)[0]             : AIC=2459.074, Time=0.04 sec
ARIMA(0,1,1)(0,0,0)[0]             : AIC=2462.221, Time=0.02 sec
ARIMA(1,1,0)(0,0,0)[0]             : AIC=2563.261, Time=0.02 sec
ARIMA(2,1,1)(0,0,0)[0]             : AIC=2460.367, Time=0.06 sec
ARIMA(1,1,2)(0,0,0)[0]             : AIC=2460.427, Time=0.12 sec
ARIMA(0,1,2)(0,0,0)[0]             : AIC=2459.571, Time=0.04 sec
ARIMA(2,1,0)(0,0,0)[0]             : AIC=2534.205, Time=0.03 sec
ARIMA(2,1,2)(0,0,0)[0]             : AIC=2462.366, Time=0.18 sec

Best model:  ARIMA(1,1,1)(0,0,0)[0]
Total fit time: 1.680 seconds
```

This shows a recommended (p,d,q) ARIMA Order of (1,1,1), with no seasonal_order component.

We can see how this was determined by looking at the stepwise results. The recommended order is the one with the lowest Akaike information criterion or AIC score. Note that the recommended model may not be the one with the closest fit. The AIC score takes complexity into account, and tries to identify the best forecasting model.

[6]:
```
'''
but we dont have to actually decide that from these results from auto_arima.

Instead, we just call stepwise_fit.summary() and it's going to give you the␣
 ↪summary of the best performing model
that Auto ARIMA thinks you should use.

We think you should use an SARIMAX(1, 1, 1).
It reports back the number of observations, the AIC score for that particular␣
 ↪model and then more information
as far as number of lags.

So we can see here, Auto Regression, lagging one component (ar.L1        ), a␣
 ↪moving average, also lagging one
component(ma.L1), essentially saying they're both order one, which we can see␣
 ↪in SARIMAX(1, 1, 1).

So basically what we do now for the next step is actually create an ARIMA model␣
 ↪using stat's models
of order (1,1,1) and then continue on of our train to split forecasting and so␣
 ↪on.
```

```
But what's really nice is we no longer need to read these complex partial␣
 ↪autocorrelation function plots
or auto correlation function plots.

'''
stepwise_fit.summary()
```

[6]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                               SARIMAX Results
==========================================================================
Dep. Variable:                          y   No. Observations:                  365
Model:                 SARIMAX(1, 1, 1)   Log Likelihood               -1226.537
Date:                  Sun, 31 Oct 2021   AIC                           2459.074
Time:                          01:34:05   BIC                           2470.766
Sample:                               0   HQIC                          2463.721
                                  - 365
Covariance Type:                    opg
==========================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------
ar.L1          0.1252      0.060      2.097      0.036       0.008       0.242
ma.L1         -0.9624      0.017    -56.429      0.000      -0.996      -0.929
sigma2        49.1512      3.250     15.122      0.000      42.781      55.522
==========================================================================
===
Ljung-Box (L1) (Q):                   0.04   Jarque-Bera (JB):
25.33
Prob(Q):                              0.84   Prob(JB):
0.00
Heteroskedasticity (H):               0.96   Skew:
0.57
Prob(H) (two-sided):                  0.81   Kurtosis:
3.60
==========================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
"""
```

[11]:
```
'''

So I'm going to assume that there's probably going to be both  AR and MA␣
 ↪component to this more complicated
data set.
```

```
So we'll say starting points for you, start_p=0,start_q=0,

Then we going to specify some max value of P and Q i.e  max_p=4,max_q=4

As for keep in mind, it may not actually reaches max values if it sees the AIC␣
 ↪starting to converge
to some point.

Now, remember, this is seasonal data, so we're going to want to specify that␣
 ↪seasonal=True,

So I remember that I already know the thousands of passengers is seasonal and␣
 ↪recall that we can perform
things like that Decky fooler test to see if we have stationary or non␣
 ↪stationary data.
And then we can plot it out and we can do an ETS to see if there is actually␣
 ↪seasonality in the decomposition.


And the other thing I want to do here is also trace=True,.
So I can see the trace results of the first couple of different Arima models␣
 ↪that it tried.

And then finally, since I specified that it's seasonal, I should be choosing an␣
 ↪M value.
So what is M?
M is just the number of periods per season.
So that's the period for seasonal differences.
M refers to the number of periods in each season.
So M would it be 4 for quarterly data, 12 for monthly data or 1 for just annual.


OK, so that's the main thing to keep in mind when you're working with seasonal␣
 ↪data sets is make sure
seasonal is true, which again is the default.

And you also want to make sure that you specify how many periods are there per␣
 ↪season in case you want
Auto Arima to perform some seasonal differences.
'''

'''

And notice, what it's also doing here is that you have to order you have your␣
 ↪lower case, p,d,q, you
```

```
and your upper case P,D,Q, so you have your Arima portion and then your␣
 ↪seasonal portion.

So this is technically running a SARIMA model.
And the way we're going to call that and sets models is SRIAMAX.
'''

stepwise_fit=auto_arima(df1["Thousands of Passengers"],
                        start_p=0,
                        start_q=0,
                        max_p=4,
                        max_q=4,
                        seasonal=True,
                        trace=True,
                        m=12)
```

```
Performing stepwise search to minimize aic
 ARIMA(0,1,0)(1,1,1)[12]             : AIC=1032.128, Time=0.10 sec
 ARIMA(0,1,0)(0,1,0)[12]             : AIC=1031.508, Time=0.01 sec
 ARIMA(1,1,0)(1,1,0)[12]             : AIC=1020.393, Time=0.05 sec
 ARIMA(0,1,1)(0,1,1)[12]             : AIC=1021.003, Time=0.07 sec
 ARIMA(1,1,0)(0,1,0)[12]             : AIC=1020.393, Time=0.02 sec
 ARIMA(1,1,0)(2,1,0)[12]             : AIC=1019.239, Time=0.11 sec
 ARIMA(1,1,0)(2,1,1)[12]             : AIC=inf, Time=0.87 sec
 ARIMA(1,1,0)(1,1,1)[12]             : AIC=1020.493, Time=0.14 sec
 ARIMA(0,1,0)(2,1,0)[12]             : AIC=1032.120, Time=0.08 sec
 ARIMA(2,1,0)(2,1,0)[12]             : AIC=1021.120, Time=0.15 sec
 ARIMA(1,1,1)(2,1,0)[12]             : AIC=1021.032, Time=0.22 sec
 ARIMA(0,1,1)(2,1,0)[12]             : AIC=1019.178, Time=0.12 sec
 ARIMA(0,1,1)(1,1,0)[12]             : AIC=1020.425, Time=0.05 sec
 ARIMA(0,1,1)(2,1,1)[12]             : AIC=inf, Time=0.54 sec
 ARIMA(0,1,1)(1,1,1)[12]             : AIC=1020.327, Time=0.16 sec
 ARIMA(0,1,2)(2,1,0)[12]             : AIC=1021.148, Time=0.14 sec
 ARIMA(1,1,2)(2,1,0)[12]             : AIC=1022.805, Time=0.26 sec
 ARIMA(0,1,1)(2,1,0)[12] intercept   : AIC=1021.017, Time=0.20 sec

Best model:  ARIMA(0,1,1)(2,1,0)[12]
Total fit time: 3.325 seconds
```

[8]:
```
'''
the summary is going to turn back the best performing model, which happens to be
this one right here SARIMAX(0, 1, 1)x(2, 1, [], 12)        .


We can see the different orders here.
So it shows that order to for auto regression as well as moving average.
```

```
And we also get to see the seasonal components.
'''
stepwise_fit.summary()
```

[8]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                                     SARIMAX Results
==========================================================================================
Dep. Variable:                                   y   No. Observations:
144
Model:             SARIMAX(0, 1, 1)x(2, 1, [], 12)   Log Likelihood
-505.589
Date:                             Sun, 31 Oct 2021   AIC
1019.178
Time:                                     01:34:08   BIC
1030.679
Sample:                                          0   HQIC
1023.851
                                               - 144
Covariance Type:                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.L1         -0.3634      0.074     -4.945      0.000      -0.508      -0.219
ar.S.L12      -0.1239      0.090     -1.372      0.170      -0.301       0.053
ar.S.L24       0.1911      0.107      1.783      0.075      -0.019       0.401
sigma2       130.4480     15.527      8.402      0.000     100.016     160.880
==============================================================================
===
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):
4.59
Prob(Q):                              0.92   Prob(JB):
0.10
Heteroskedasticity (H):               2.70   Skew:
0.15
Prob(H) (two-sided):                  0.00   Kurtosis:
3.87
==============================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
"""
```

1.1 OPTIONAL: statsmodels ARMA_Order_Select_IC

Statsmodels has a selection tool to find orders for ARMA models on stationary data.

[12]:
```
from statsmodels.tsa.stattools import arma_order_select_ic
```

[13]:
```
help(arma_order_select_ic)
```

```
Help on function arma_order_select_ic in module statsmodels.tsa.stattools:

arma_order_select_ic(y, max_ar=4, max_ma=2, ic='bic', trend='c', model_kw=None,
fit_kw=None)
    Compute information criteria for many ARMA models.

    Parameters
    ----------
    y : array_like
        Array of time-series data.
    max_ar : int
        Maximum number of AR lags to use. Default 4.
    max_ma : int
        Maximum number of MA lags to use. Default 2.
    ic : str, list
        Information criteria to report. Either a single string or a list
        of different criteria is possible.
    trend : str
        The trend to use when fitting the ARMA models.
    model_kw : dict
        Keyword arguments to be passed to the ``ARMA`` model.
    fit_kw : dict
        Keyword arguments to be passed to ``ARMA.fit``.

    Returns
    -------
    Bunch
        Dict-like object with attribute access. Each ic is an attribute with a
        DataFrame for the results. The AR order used is the row index. The ma
        order used is the column index. The minimum orders are available as
        ``ic_min_order``.

    Notes
    -----
    This method can be used to tentatively identify the order of an ARMA
    process, provided that the time series is stationary and invertible. This
    function computes the full exact MLE estimate of each model and can be,
    therefore a little slow. An implementation using approximate estimates
    will be provided in the future. In the meantime, consider passing
    {method : "css"} to fit_kw.
```

```
    Examples
    --------

    >>> from statsmodels.tsa.arima_process import arma_generate_sample
    >>> import statsmodels.api as sm
    >>> import numpy as np

    >>> arparams = np.array([.75, -.25])
    >>> maparams = np.array([.65, .35])
    >>> arparams = np.r_[1, -arparams]
    >>> maparam = np.r_[1, maparams]
    >>> nobs = 250
    >>> np.random.seed(2014)
    >>> y = arma_generate_sample(arparams, maparams, nobs)
    >>> res = sm.tsa.arma_order_select_ic(y, ic=["aic", "bic"], trend="nc")
    >>> res.aic_min_order
    >>> res.bic_min_order
```

[14]:
```
arma_order_select_ic(df2['Births'])
```

[14]:
```
{'bic':              0            1            2
 0  2502.581666  2494.238827  2494.731525
 1  2490.780306  2484.505386  2486.223523
 2  2491.963234  2485.782753  2491.097218
 3  2496.498618  2491.061564  2496.961178
 4  2501.491891  2504.012588  2498.329743,
 'bic_min_order': (1, 1)}
```

[ ]:
```

```