04-Choosing-ARIMA-Orders

October 19, 2022

---

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

1.1 Perform standard imports and load datasets

[1]:
```
import pandas as pd
import numpy as np
%matplotlib inline
```

```
# 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'
```

1.2 pmdarima Auto-ARIMA

This is a third-party tool separate from statsmodels. It should already be installed if you're using our virtual environment. If not, then at a terminal run: pip install pmdarima

[2]:
```
from pmdarima import auto_arima

# Ignore harmless warnings
import warnings
warnings.filterwarnings("ignore")
```

[3]:
```
help(auto_arima)
```

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

auto_arima(y, exogenous=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=10,
m=1, seasonal=True, stationary=False, information_criterion='aic', alpha=0.05,
test='kpss', seasonal_test='ch', stepwise=True, n_jobs=1, start_params=None,
trend=None, method=None, transparams=True, solver='lbfgs', maxiter=50, disp=0,
callback=None, offset_test_args=None, seasonal_test_args=None,
suppress_warnings=False, error_action='warn', 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=True, **fit_args)
    Automatically discover the optimal order for an ARIMA model.

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

    The ``auro_arima`` function 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.

exogenous : 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=10)
    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='ch')
    This determines which seasonal unit root test is used if ``seasonal``
    is True and ``D`` is None. Default is 'ch' (Canova-Hansen).

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``.

transparams : bool, optional (default=True)
    Whether or not to transform the parameters to ensure stationarity.
```

```
    Uses the transformation suggested in Jones (1980).  If False,
    no checking for stationarity or invertibility is done.

method : str, one of {'css-mle','mle','css'}, optional (default=None)
    This is the loglikelihood to maximize.  If "css-mle", the
    conditional sum of squares likelihood is maximized and its values
    are used as starting values for the computation of the exact
    likelihood via the Kalman filter.  If "mle", the exact likelihood
    is maximized via the Kalman Filter.  If "css" the conditional sum
    of squares likelihood is maximized.  All three methods use
    `start_params` as starting parameters.  See above for more
    information. If fitting a seasonal ARIMA, the default is 'lbfgs'

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.

solver : str or None, optional (default='lbfgs')
    Solver to be used.  The default is 'lbfgs' (limited memory
    Broyden-Fletcher-Goldfarb-Shanno).  Other choices are 'bfgs',
    'newton' (Newton-Raphson), 'nm' (Nelder-Mead), 'cg' -
    (conjugate gradient), 'ncg' (non-conjugate gradient), and
    'powell'. By default, the limited memory BFGS uses m=12 to
    approximate the Hessian, projected gradient tolerance of 1e-8 and
    factr = 1e2. You can change these by using kwargs.

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

disp : int, optional (default=0)
    If True, convergence information is printed.  For the default
    'lbfgs' ``solver``, disp controls the frequency of the output during
    the iterations. disp < 0 means no output in this case.

callback : callable, optional (default=None)
    Called after each iteration as callback(xk) where xk is the current
    parameter vector. This is only used in non-seasonal ARIMA models.

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=False)
```

```
    Many warnings might be thrown inside of statsmodels. If
    ``suppress_warnings`` is True, all of the warnings coming from
    ``ARIMA`` will be squelched.

error_action : str, optional (default='warn')
    If unable to fit an ``ARIMA`` due to stationarity issues, whether to
    warn ('warn'), raise the ``ValueError`` ('raise') or ignore ('ignore').
    Note that the default behavior is to warn, and fits that fail will be
    returned as None. This is the recommended behavior, as statsmodels
    ARIMA and SARIMAX models hit bugs periodically that can cause
    an otherwise healthy parameter combination to fail for reasons not
    related to pmdarima.

trace : bool, optional (default=False)
    Whether to print status on the fits. Note that this can be
    very verbose…

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, optional (default=True)
    Whether to include an intercept term. Default is True.

**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: http://bit.ly/2gOh5z2
.. [3] R's auto-arima documentation: http://bit.ly/2wbBvUN
```

Let's look first at the stationary, non-seasonal Daily Female Births dataset:

[4]:
```python
auto_arima(df2['Births'])
```

[4]:
```
ARIMA(callback=None, disp=0, maxiter=50, method=None, order=(1, 1, 1),
   out_of_sample_size=0, scoring='mse', scoring_args={},
   seasonal_order=(0, 0, 0, 1), solver='lbfgs', start_params=None,
   suppress_warnings=False, transparams=True, trend=None,
   with_intercept=True)
```

NOTE: Harmless warnings should have been suppressed, but if you see an error citing unusual behavior you can suppress this message by passing error_action='ignore' into auto_arima(). Also, auto_arima().summary() provides a nicely formatted summary table.

```
[5]: auto_arima(df2['Births'],error_action='ignore').summary()
```

```
[5]: <class 'statsmodels.iolib.summary.Summary'>
"""
                           Statespace Model Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  365
Model:               SARIMAX(1, 1, 1)   Log Likelihood               -1226.077
Date:                Fri, 22 Mar 2019   AIC                           2460.154
Time:                        10:46:05   BIC                           2475.743
Sample:                             0   HQIC                          2466.350
                                - 365
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      0.0132      0.014      0.975      0.330      -0.013       0.040
ar.L1          0.1299      0.059      2.217      0.027       0.015       0.245
ma.L1         -0.9694      0.016    -62.235      0.000      -1.000      -0.939
sigma2        48.9989      3.432     14.279      0.000      42.273      55.725
================================================================================
===
Ljung-Box (Q):                       36.69   Jarque-Bera (JB):
26.17
Prob(Q):                              0.62   Prob(JB):
0.00
Heteroskedasticity (H):               0.97   Skew:
0.58
Prob(H) (two-sided):                  0.85   Kurtosis:
3.62
================================================================================
===

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

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]:
```python
stepwise_fit = auto_arima(df2['Births'], start_p=0, start_q=0,
                          max_p=6, max_q=3, m=12,
                          seasonal=False,
                          d=None, trace=True,
                          error_action='ignore',   # we don't want to know if␣
 ↪an order does not work
                          suppress_warnings=True,  # we don't want convergence␣
 ↪warnings
                          stepwise=True)           # set to stepwise

stepwise_fit.summary()
```

```
Fit ARIMA: order=(0, 1, 0); AIC=2650.760, BIC=2658.555, Fit time=0.018 seconds
Fit ARIMA: order=(1, 1, 0); AIC=2565.234, BIC=2576.925, Fit time=0.139 seconds
Fit ARIMA: order=(0, 1, 1); AIC=2463.584, BIC=2475.275, Fit time=0.047 seconds
Fit ARIMA: order=(1, 1, 1); AIC=2460.154, BIC=2475.742, Fit time=0.105 seconds
Fit ARIMA: order=(1, 1, 2); AIC=2460.515, BIC=2480.000, Fit time=0.320 seconds
Fit ARIMA: order=(2, 1, 2); AIC=2462.045, BIC=2485.428, Fit time=0.373 seconds
Fit ARIMA: order=(2, 1, 1); AIC=2461.271, BIC=2480.757, Fit time=0.169 seconds
Total fit time: 1.175 seconds
```

[6]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                             ARIMA Model Results
==============================================================================
Dep. Variable:                    D.y   No. Observations:                  364
Model:                 ARIMA(1, 1, 1)   Log Likelihood               -1226.077
Method:                       css-mle   S.D. of innovations              7.000
Date:                Fri, 22 Mar 2019   AIC                           2460.154
Time:                        10:46:20   BIC                           2475.742
Sample:                             1   HQIC                          2466.350

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.0152      0.014      1.068      0.286      -0.013       0.043
ar.L1.D.y      0.1299      0.056      2.334      0.020       0.021       0.239
ma.L1.D.y     -0.9694      0.019    -51.415      0.000      -1.006      -0.932
                                    Roots
=============================================================================
                  Real          Imaginary           Modulus         Frequency
-----------------------------------------------------------------------------
AR.1            7.6996           +0.0000j            7.6996            0.0000
MA.1            1.0316           +0.0000j            1.0316            0.0000
-----------------------------------------------------------------------------
"""
```

Now let's look at the non-stationary, seasonal Airline Passengers dataset:

[7]:
```python
stepwise_fit = auto_arima(df1['Thousands of Passengers'], start_p=1, start_q=1,
                          max_p=3, max_q=3, m=12,
                          start_P=0, seasonal=True,
                          d=None, D=1, trace=True,
                          error_action='ignore',   # we don't want to know if␣
 ↪an order does not work
                          suppress_warnings=True,  # we don't want convergence␣
 ↪warnings
                          stepwise=True)           # set to stepwise

stepwise_fit.summary()
```

```
Fit ARIMA: order=(1, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=1024.824,
BIC=1039.200, Fit time=0.574 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(0, 1, 0, 12); AIC=1033.479,
BIC=1039.229, Fit time=0.000 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 1, 0, 12); AIC=1022.316,
BIC=1033.817, Fit time=0.294 seconds
Fit ARIMA: order=(0, 1, 1) seasonal_order=(0, 1, 1, 12); AIC=1022.904,
BIC=1034.405, Fit time=0.251 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(0, 1, 0, 12); AIC=1022.343,
BIC=1030.968, Fit time=0.094 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(2, 1, 0, 12); AIC=1021.142,
BIC=1035.518, Fit time=0.748 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(2, 1, 1, 12); AIC=1016.960,
BIC=1034.211, Fit time=2.258 seconds
Fit ARIMA: order=(0, 1, 0) seasonal_order=(2, 1, 1, 12); AIC=1033.371,
BIC=1047.747, Fit time=1.931 seconds
Fit ARIMA: order=(2, 1, 0) seasonal_order=(2, 1, 1, 12); AIC=1018.094,
BIC=1038.221, Fit time=2.492 seconds
Fit ARIMA: order=(1, 1, 1) seasonal_order=(2, 1, 1, 12); AIC=1017.829,
BIC=1037.955, Fit time=2.464 seconds
Fit ARIMA: order=(2, 1, 1) seasonal_order=(2, 1, 1, 12); AIC=nan, BIC=nan, Fit
time=nan seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(1, 1, 1, 12); AIC=1022.425,
BIC=1036.801, Fit time=0.432 seconds
Fit ARIMA: order=(1, 1, 0) seasonal_order=(2, 1, 2, 12); AIC=1017.410,
BIC=1037.536, Fit time=2.546 seconds
Total fit time: 14.093 seconds
```

[7]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                           Statespace Model Results
==================================================================================
```

```
==========
Dep. Variable:                                  y   No. Observations:
144
Model:             SARIMAX(1, 1, 0)x(2, 1, 1, 12)   Log Likelihood
-502.480
Date:                            Fri, 22 Mar 2019   AIC
1016.960
Time:                                    10:46:39   BIC
1034.211
Sample:                                         0   HQIC
1023.970
                                            - 144
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept      0.0045      0.178      0.025      0.980      -0.345       0.354
ar.L1         -0.3766      0.077     -4.890      0.000      -0.527      -0.226
ar.S.L12       0.6891      0.140      4.918      0.000       0.414       0.964
ar.S.L24       0.3091      0.107      2.883      0.004       0.099       0.519
ma.S.L12      -0.9742      0.511     -1.906      0.057      -1.976       0.028
sigma2       113.2075     48.855      2.317      0.020      17.453     208.961
================================================================================
===
Ljung-Box (Q):                       58.67   Jarque-Bera (JB):
12.12
Prob(Q):                              0.03   Prob(JB):
0.00
Heteroskedasticity (H):               2.70   Skew:
0.10
Prob(H) (two-sided):                  0.00   Kurtosis:
4.48
================================================================================
===

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

1.3 OPTIONAL: statsmodels ARMA_Order_Select_IC

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

[8]:
```python
from statsmodels.tsa.stattools import arma_order_select_ic
```

[9]:
```
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={},
fit_kw={})
    Returns information criteria for many ARMA models

    Parameters
    ----------
    y : array-like
        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
    -------
    obj : Results object
        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``.

    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

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.
```

[10]:
```python
arma_order_select_ic(df2['Births'])
```

[10]:
```
{'bic':             0            1            2
 0  2502.581666  2494.238827  2494.731525
 1  2490.780306  2484.505386          NaN
 2  2491.963234  2485.782753  2491.097206
 3  2496.498618  2491.061564  2496.961178
 4  2501.491891  2504.012579  2498.329743, 'bic_min_order': (1, 1)}
```

[11]:
```python
arma_order_select_ic(df1['Thousands of Passengers'])
```

[11]:
```
{'bic':             0            1            2
 0  1796.307207  1627.771967  1534.002384
 1  1437.088819  1421.627524  1425.899321
 2  1425.518037  1423.098290          NaN
 3  1425.191373          NaN  1400.744437
 4  1427.576572          NaN  1414.310652, 'bic_min_order': (3, 2)}
```

A note about statsmodels.tsa.x13.x13_arima_select_order This utility requires installation of X-13ARIMA-SEATS, a seasonal adjustment tool developed by the U.S. Census Bureau. See https://www.census.gov/srd/www/x13as/ for details. Since the installation requires adding x13as to your PATH settings we've deemed it beyond the scope of this course.