Maximum Likelihood Estimation and Ordinary Least Squares on Simple Linear Regression

August 6, 2019

[3]:

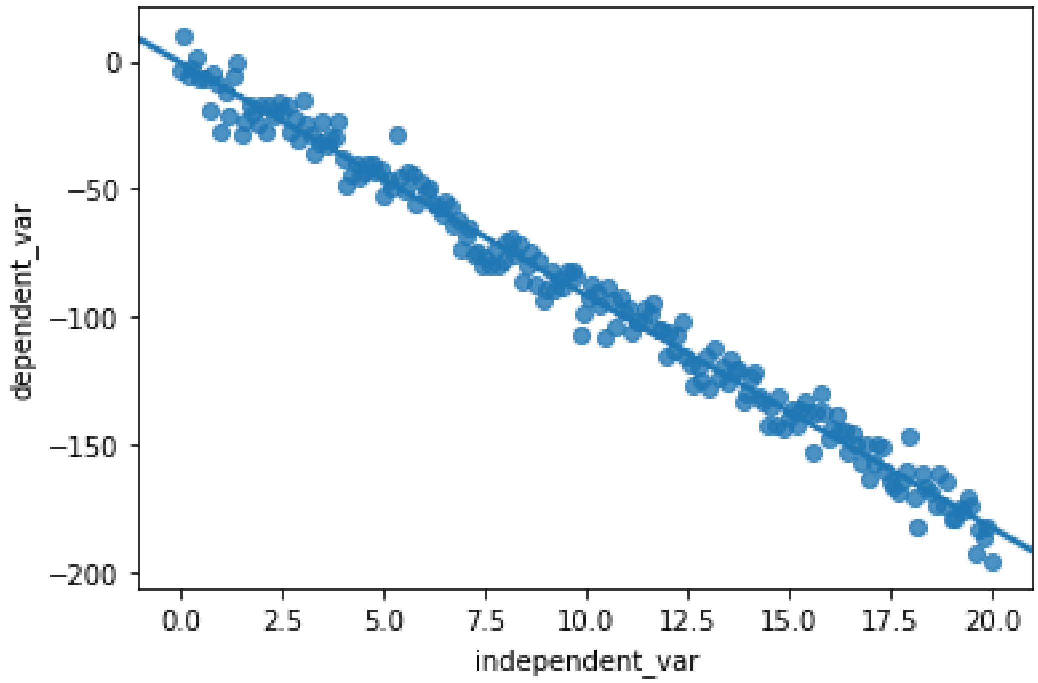
```
#!pip install --upgrade pip --user
#!pip install numdifftools --user
#!pip install scipy --upgrade --user

!pip uninstall statsmodels -y --user
!pip install statsmodels==0.10.0rc2 --pre --user
```

```
Usage:
  pip uninstall [options] <package> ...
  pip uninstall [options] -r <requirements file> ...

no such option: --user
Collecting statsmodels==0.10.0rc2
  Downloading https://files.pythonhosted.org/packages/ab/f5/6bb191bb31574f
59b42ee10278a8d002e5be4055d41f9af86d682b8c63b6/statsmodels-0.10.0rc2-cp36-cp36m-
manylinux1_x86_64.whl (8.1MB)
     || 8.1MB 8.3MB/s eta 0:00:01
|                | 3.9MB 2.6MB/s eta 0:00:02
Requirement already satisfied: pandas>=0.19 in
/home/data/.local/lib/python3.6/site-packages (from statsmodels==0.10.0rc2)
(0.24.2)
Requirement already satisfied: patsy>=0.4.0 in
/home/data/.local/lib/python3.6/site-packages (from statsmodels==0.10.0rc2)
(0.5.1)
Requirement already satisfied: scipy>=0.18 in
/home/data/.local/lib/python3.6/site-packages (from statsmodels==0.10.0rc2)
(1.3.0)
Requirement already satisfied: numpy>=1.11 in
/home/data/.local/lib/python3.6/site-packages (from statsmodels==0.10.0rc2)
(1.17.0)
Requirement already satisfied: python-dateutil>=2.5.0 in /usr/lib/python3/dist-
packages (from pandas>=0.19->statsmodels==0.10.0rc2) (2.6.1)
Requirement already satisfied: pytz>=2011k in /usr/lib/python3/dist-packages
(from pandas>=0.19->statsmodels==0.10.0rc2) (2018.3)
```

```
Requirement already satisfied: six in /usr/lib/python3/dist-packages (from
patsy>=0.4.0->statsmodels==0.10.0rc2) (1.11.0)
Installing collected packages: statsmodels
  Found existing installation: statsmodels 0.9.0
    Uninstalling statsmodels-0.9.0:
      Successfully uninstalled statsmodels-0.9.0
Successfully installed statsmodels-0.10.0rc2
```

[20]:
```
# import libraries
import numpy as np, pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from scipy.optimize import minimize
import scipy.stats as stats
import pymc3 as pm3
import numdifftools as ndt
import statsmodels.api as sm
from statsmodels.base.model import GenericLikelihoodModel


# generate data
N = 200
x = np.linspace(0,20,N)
e = np.random.normal(loc = 0.0, scale = 7.0, size = N)
y = -9*x + e
df = pd.DataFrame({'dependent_var':y, 'independent_var':x})
df['constant'] = 1
```

[21]:
```
# plot
sns.regplot(df.independent_var, df.dependent_var);
```



[4]:
```
df.head()
```

[4]:
```
           y         x  constant
0  11.600880  0.000000         1
1  -3.151675  0.202020         1
2   2.477100  0.404040         1
3  -4.506962  0.606061         1
4  -1.766583  0.808081         1
```

Before we go into the implementation of log likelihood, please take a quick look at the documentation Scipy norm documentation

From the documentation we can see that given the location and scale, we can find the log probabilities of the corresponding normal distribution density

[17]:
```
print(f"The density values for data points are {stats.norm.pdf([1,-1], loc=0,␣
↪scale=1)}")
print(f"The natural log of the density values for the data points are {stats.
↪norm.logpdf([1,-1], loc=0, scale=1)}")
```

```
The density values for data points are [0.24197072 0.24197072]
The natural log of the density values for the data points are [-1.41893853
-1.41893853]
```

Verify the result yourself

0.1 Ordinary Least Squares Fit Using Statsmodels

[24]:
```python
# split features and target
X = df[['constant', 'independent_var']]
# fit model and summarize
statsmodels_ols_summary = sm.OLS(y,X).fit().summary()
statsmodels_ols_summary
```

[24]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.985
Model:                            OLS   Adj. R-squared:                  0.985
Method:                 Least Squares   F-statistic:                 1.281e+04
Date:                Tue, 06 Aug 2019   Prob (F-statistic):          6.42e-182
Time:                        11:30:02   Log-Likelihood:                -660.15
No. Observations:                 200   AIC:                             1324.
Df Residuals:                     198   BIC:                             1331.
Df Model:                           1
Covariance Type:            nonrobust
================================================================================
===
                      coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
---
constant           -0.5107      0.930     -0.549      0.583      -2.344
1.323
independent_var    -9.1022      0.080   -113.197      0.000      -9.261
-8.944
==============================================================================
Omnibus:                        0.705   Durbin-Watson:                   1.747
Prob(Omnibus):                  0.703   Jarque-Bera (JB):                0.606
Skew:                          -0.135   Prob(JB):                        0.739
Kurtosis:                       3.003   Cond. No.                         23.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

0.2 Implementing the Likelihood Function

[25]:
```
# define likelihood function
def neg_log_likelihood(params):
    """ Computes the """
    intercept, beta, sd = params[0], params[1], params[2] # inputs are guesses␣
 ↪at our parameters
    yhat = intercept + beta*x # predictions
    # to find the negative log likelihood, sum the log densities as according␣
 ↪to the likelihood function formula
    negative_log_likelihood = -np.sum( stats.norm.logpdf(y, loc=yhat, scale=sd)␣
 ↪)
    return(negative_log_likelihood)

# let's start with some random coefficient guesses and optimize
guess = np.array([5,5,2])
results = minimize(neg_log_likelihood, guess, method =␣
 ↪'Nelder-Mead',options={'disp': True})

# drop results into df and round to match statsmodels
resultsdf = pd.DataFrame({'coef':results['x']})
resultsdf.index=['constant','x','sigma']
```

```
Optimization terminated successfully.
         Current function value: 660.147313
         Iterations: 186
         Function evaluations: 339
```

[19]:
```
# Let's also see if there are other Scipy options available for optimization
print(minimize.__doc__)
```

```
Minimization of scalar function of one or more variables.

    Parameters
    ----------
    fun : callable
        The objective function to be minimized.

            ``fun(x, *args) -> float``

        where x is an 1-D array with shape (n,) and `args`
        is a tuple of the fixed parameters needed to completely
        specify the function.
    x0 : ndarray, shape (n,)
        Initial guess. Array of real elements of size (n,),
        where 'n' is the number of independent variables.
    args : tuple, optional
```

```
        Extra arguments passed to the objective function and its
        derivatives (`fun`, `jac` and `hess` functions).
    method : str or callable, optional
        Type of solver.  Should be one of

            - 'Nelder-Mead' :ref:`(see here) <optimize.minimize-neldermead>`
            - 'Powell'      :ref:`(see here) <optimize.minimize-powell>`
            - 'CG'          :ref:`(see here) <optimize.minimize-cg>`
            - 'BFGS'        :ref:`(see here) <optimize.minimize-bfgs>`
            - 'Newton-CG'   :ref:`(see here) <optimize.minimize-newtoncg>`
            - 'L-BFGS-B'    :ref:`(see here) <optimize.minimize-lbfgsb>`
            - 'TNC'         :ref:`(see here) <optimize.minimize-tnc>`
            - 'COBYLA'      :ref:`(see here) <optimize.minimize-cobyla>`
            - 'SLSQP'       :ref:`(see here) <optimize.minimize-slsqp>`
            - 'trust-constr':ref:`(see here) <optimize.minimize-trustconstr>`
            - 'dogleg'      :ref:`(see here) <optimize.minimize-dogleg>`
            - 'trust-ncg'   :ref:`(see here) <optimize.minimize-trustncg>`
            - 'trust-exact' :ref:`(see here) <optimize.minimize-trustexact>`
            - 'trust-krylov' :ref:`(see here) <optimize.minimize-trustkrylov>`
            - custom - a callable object (added in version 0.14.0),
              see below for description.

        If not given, chosen to be one of ``BFGS``, ``L-BFGS-B``, ``SLSQP``,
        depending if the problem has constraints or bounds.
    jac : {callable,  '2-point', '3-point', 'cs', bool}, optional
        Method for computing the gradient vector. Only for CG, BFGS,
        Newton-CG, L-BFGS-B, TNC, SLSQP, dogleg, trust-ncg, trust-krylov,
        trust-exact and trust-constr. If it is a callable, it should be a
        function that returns the gradient vector:

            ``jac(x, *args) -> array_like, shape (n,)``

        where x is an array with shape (n,) and `args` is a tuple with
        the fixed parameters. Alternatively, the keywords
        {'2-point', '3-point', 'cs'} select a finite
        difference scheme for numerical estimation of the gradient. Options
        '3-point' and 'cs' are available only to 'trust-constr'.
        If `jac` is a Boolean and is True, `fun` is assumed to return the
        gradient along with the objective function. If False, the gradient
        will be estimated using '2-point' finite difference estimation.
    hess : {callable, '2-point', '3-point', 'cs', HessianUpdateStrategy},
optional
        Method for computing the Hessian matrix. Only for Newton-CG, dogleg,
        trust-ncg,  trust-krylov, trust-exact and trust-constr. If it is
        callable, it should return the  Hessian matrix:

            ``hess(x, *args) -> {LinearOperator, spmatrix, array}, (n, n)``
```

```
    where x is a (n,) ndarray and `args` is a tuple with the fixed
    parameters. LinearOperator and sparse matrix returns are
    allowed only for 'trust-constr' method. Alternatively, the keywords
    {'2-point', '3-point', 'cs'} select a finite difference scheme
    for numerical estimation. Or, objects implementing
    `HessianUpdateStrategy` interface can be used to approximate
    the Hessian. Available quasi-Newton methods implementing
    this interface are:

        - `BFGS`;
        - `SR1`.

    Whenever the gradient is estimated via finite-differences,
    the Hessian cannot be estimated with options
    {'2-point', '3-point', 'cs'} and needs to be
    estimated using one of the quasi-Newton strategies.
    Finite-difference options {'2-point', '3-point', 'cs'} and
    `HessianUpdateStrategy` are available only for 'trust-constr' method.
hessp : callable, optional
    Hessian of objective function times an arbitrary vector p. Only for
    Newton-CG, trust-ncg, trust-krylov, trust-constr.
    Only one of `hessp` or `hess` needs to be given.  If `hess` is
    provided, then `hessp` will be ignored.  `hessp` must compute the
    Hessian times an arbitrary vector:

        ``hessp(x, p, *args) ->  ndarray shape (n,)``

    where x is a (n,) ndarray, p is an arbitrary vector with
    dimension (n,) and `args` is a tuple with the fixed
    parameters.
bounds : sequence or `Bounds`, optional
    Bounds on variables for L-BFGS-B, TNC, SLSQP and
    trust-constr methods. There are two ways to specify the bounds:

        1. Instance of `Bounds` class.
        2. Sequence of ``(min, max)`` pairs for each element in `x`. None
           is used to specify no bound.

constraints : {Constraint, dict} or List of {Constraint, dict}, optional
    Constraints definition (only for COBYLA, SLSQP and trust-constr).
    Constraints for 'trust-constr' are defined as a single object or a
    list of objects specifying constraints to the optimization problem.
    Available constraints are:

        - `LinearConstraint`
        - `NonlinearConstraint`

    Constraints for COBYLA, SLSQP are defined as a list of dictionaries.
```

```
    Each dictionary with fields:

        type : str
            Constraint type: 'eq' for equality, 'ineq' for inequality.
        fun : callable
            The function defining the constraint.
        jac : callable, optional
            The Jacobian of `fun` (only for SLSQP).
        args : sequence, optional
            Extra arguments to be passed to the function and Jacobian.

    Equality constraint means that the constraint function result is to
    be zero whereas inequality means that it is to be non-negative.
    Note that COBYLA only supports inequality constraints.
tol : float, optional
    Tolerance for termination. For detailed control, use solver-specific
    options.
options : dict, optional
    A dictionary of solver options. All methods accept the following
    generic options:

        maxiter : int
            Maximum number of iterations to perform.
        disp : bool
            Set to True to print convergence messages.

    For method-specific options, see :func:`show_options()`.
callback : callable, optional
    Called after each iteration. For 'trust-constr' it is a callable with
    the signature:

        ``callback(xk, OptimizeResult state) -> bool``

    where ``xk`` is the current parameter vector. and ``state``
    is an `OptimizeResult` object, with the same fields
    as the ones from the return.  If callback returns True
    the algorithm execution is terminated.
    For all the other methods, the signature is:

        ``callback(xk)``

    where ``xk`` is the current parameter vector.

Returns
-------
res : OptimizeResult
    The optimization result represented as a ``OptimizeResult`` object.
    Important attributes are: ``x`` the solution array, ``success`` a
```

```
    Boolean flag indicating if the optimizer exited successfully and
    ``message`` which describes the cause of the termination. See
    `OptimizeResult` for a description of other attributes.

See also
--------
minimize_scalar : Interface to minimization algorithms for scalar
    univariate functions
show_options : Additional options accepted by the solvers

Notes
-----
This section describes the available solvers that can be selected by the
'method' parameter. The default method is *BFGS*.

**Unconstrained minimization**

Method :ref:`Nelder-Mead <optimize.minimize-neldermead>` uses the
Simplex algorithm [1]_, [2]_. This algorithm is robust in many
applications. However, if numerical computation of derivative can be
trusted, other algorithms using the first and/or second derivatives
information might be preferred for their better performance in
general.

Method :ref:`Powell <optimize.minimize-powell>` is a modification
of Powell's method [3]_, [4]_ which is a conjugate direction
method. It performs sequential one-dimensional minimizations along
each vector of the directions set (`direc` field in `options` and
`info`), which is updated at each iteration of the main
minimization loop. The function need not be differentiable, and no
derivatives are taken.

Method :ref:`CG <optimize.minimize-cg>` uses a nonlinear conjugate
gradient algorithm by Polak and Ribiere, a variant of the
Fletcher-Reeves method described in [5]_ pp.  120-122. Only the
first derivatives are used.

Method :ref:`BFGS <optimize.minimize-bfgs>` uses the quasi-Newton
method of Broyden, Fletcher, Goldfarb, and Shanno (BFGS) [5]_
pp. 136. It uses the first derivatives only. BFGS has proven good
performance even for non-smooth optimizations. This method also
returns an approximation of the Hessian inverse, stored as
`hess_inv` in the OptimizeResult object.

Method :ref:`Newton-CG <optimize.minimize-newtoncg>` uses a
Newton-CG algorithm [5]_ pp. 168 (also known as the truncated
Newton method). It uses a CG method to the compute the search
direction. See also *TNC* method for a box-constrained
```

```
minimization with a similar algorithm. Suitable for large-scale
problems.

Method :ref:`dogleg <optimize.minimize-dogleg>` uses the dog-leg
trust-region algorithm [5]_ for unconstrained minimization. This
algorithm requires the gradient and Hessian; furthermore the
Hessian is required to be positive definite.

Method :ref:`trust-ncg <optimize.minimize-trustncg>` uses the
Newton conjugate gradient trust-region algorithm [5]_ for
unconstrained minimization. This algorithm requires the gradient
and either the Hessian or a function that computes the product of
the Hessian with a given vector. Suitable for large-scale problems.

Method :ref:`trust-krylov <optimize.minimize-trustkrylov>` uses
the Newton GLTR trust-region algorithm [14]_, [15]_ for unconstrained
minimization. This algorithm requires the gradient
and either the Hessian or a function that computes the product of
the Hessian with a given vector. Suitable for large-scale problems.
On indefinite problems it requires usually less iterations than the
`trust-ncg` method and is recommended for medium and large-scale problems.

Method :ref:`trust-exact <optimize.minimize-trustexact>`
is a trust-region method for unconstrained minimization in which
quadratic subproblems are solved almost exactly [13]_. This
algorithm requires the gradient and the Hessian (which is
*not* required to be positive definite). It is, in many
situations, the Newton method to converge in fewer iteraction
and the most recommended for small and medium-size problems.

**Bound-Constrained minimization**

Method :ref:`L-BFGS-B <optimize.minimize-lbfgsb>` uses the L-BFGS-B
algorithm [6]_, [7]_ for bound constrained minimization.

Method :ref:`TNC <optimize.minimize-tnc>` uses a truncated Newton
algorithm [5]_, [8]_ to minimize a function with variables subject
to bounds. This algorithm uses gradient information; it is also
called Newton Conjugate-Gradient. It differs from the *Newton-CG*
method described above as it wraps a C implementation and allows
each variable to be given upper and lower bounds.

**Constrained Minimization**

Method :ref:`COBYLA <optimize.minimize-cobyla>` uses the
Constrained Optimization BY Linear Approximation (COBYLA) method
[9]_, [10]_, [11]_. The algorithm is based on linear
approximations to the objective function and each constraint. The
```

method wraps a FORTRAN implementation of the algorithm. The
constraints functions 'fun' may return either a single number
or an array or list of numbers.

Method :ref:`SLSQP <optimize.minimize-slsqp>` uses Sequential
Least SQuares Programming to minimize a function of several
variables with any combination of bounds, equality and inequality
constraints. The method wraps the SLSQP Optimization subroutine
originally implemented by Dieter Kraft [12]_. Note that the
wrapper handles infinite values in bounds by converting them into
large floating values.

Method :ref:`trust-constr <optimize.minimize-trustconstr>` is a
trust-region algorithm for constrained optimization. It swiches
between two implementations depending on the problem definition.
It is the most versatile constrained minimization algorithm
implemented in SciPy and the most appropriate for large-scale problems.
For equality constrained problems it is an implementation of Byrd-Omojokun
Trust-Region SQP method described in [17]_ and in [5]_, p. 549. When
inequality constraints are imposed as well, it swiches to the trust-region
interior point method described in [16]_. This interior point algorithm,
in turn, solves inequality constraints by introducing slack variables
and solving a sequence of equality-constrained barrier problems
for progressively smaller values of the barrier parameter.
The previously described equality constrained SQP method is
used to solve the subproblems with increasing levels of accuracy
as the iterate gets closer to a solution.

Finite-Difference Options

For Method :ref:`trust-constr <optimize.minimize-trustconstr>`
the gradient and the Hessian may be approximated using
three finite-difference schemes: {'2-point', '3-point', 'cs'}.
The scheme 'cs' is, potentially, the most accurate but it
requires the function to correctly handles complex inputs and to
be differentiable in the complex plane. The scheme '3-point' is more
accurate than '2-point' but requires twice as much operations.

Custom minimizers

It may be useful to pass a custom minimization method, for example
when using a frontend to this method such as `scipy.optimize.basinhopping`
or a different library. You can simply pass a callable as the ``method``
parameter.

The callable is called as ``method(fun, x0, args, **kwargs, **options)``
where ``kwargs`` corresponds to any other parameters passed to `minimize`
(such as `callback`, `hess`, etc.), except the `options` dict, which has

its contents also passed as `method` parameters pair by pair. Also, if `jac` has been passed as a bool type, `jac` and `fun` are mangled so that `fun` returns just the function values and `jac` is converted to a function returning the Jacobian. The method shall return an `OptimizeResult` object.

The provided `method` callable must be able to accept (and possibly ignore) arbitrary parameters; the set of parameters accepted by `minimize` may expand in future versions and then these parameters will be passed to the method. You can find an example in the scipy.optimize tutorial.

.. versionadded:: 0.11.0

References

.. [1] Nelder, J A, and R Mead. 1965. A Simplex Method for Function Minimization. The Computer Journal 7: 308-13.
.. [2] Wright M H. 1996. Direct search methods: Once scorned, now respectable, in Numerical Analysis 1995: Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis (Eds. D F Griffiths and G A Watson). Addison Wesley Longman, Harlow, UK. 191-208.
.. [3] Powell, M J D. 1964. An efficient method for finding the minimum of a function of several variables without calculating derivatives. The Computer Journal 7: 155-162.
.. [4] Press W, S A Teukolsky, W T Vetterling and B P Flannery. Numerical Recipes (any edition), Cambridge University Press.
.. [5] Nocedal, J, and S J Wright. 2006. Numerical Optimization. Springer New York.
.. [6] Byrd, R H and P Lu and J. Nocedal. 1995. A Limited Memory Algorithm for Bound Constrained Optimization. SIAM Journal on Scientific and Statistical Computing 16 (5): 1190-1208.
.. [7] Zhu, C and R H Byrd and J Nocedal. 1997. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software 23 (4): 550-560.
.. [8] Nash, S G. Newton-Type Minimization Via the Lanczos Method. 1984. SIAM Journal of Numerical Analysis 21: 770-778.
.. [9] Powell, M J D. A direct search optimization method that models the objective and constraint functions by linear interpolation. 1994. Advances in Optimization and Numerical Analysis, eds. S. Gomez and J-P Hennart, Kluwer Academic (Dordrecht), 51-67.
.. [10] Powell M J D. Direct search algorithms for optimization calculations. 1998. Acta Numerica 7: 287-336.
.. [11] Powell M J D. A view of algorithms for optimization without derivatives. 2007.Cambridge University Technical Report DAMTP 2007/NA03
.. [12] Kraft, D. A software package for sequential quadratic

```
   programming. 1988. Tech. Rep. DFVLR-FB 88-28, DLR German Aerospace
   Center -- Institute for Flight Mechanics, Koln, Germany.
.. [13] Conn, A. R., Gould, N. I., and Toint, P. L.
   Trust region methods. 2000. Siam. pp. 169-200.
.. [14] F. Lenders, C. Kirches, A. Potschka: "trlib: A vector-free
   implementation of the GLTR method for iterative solution of
   the trust region problem", https://arxiv.org/abs/1611.04718
.. [15] N. Gould, S. Lucidi, M. Roma, P. Toint: "Solving the
   Trust-Region Subproblem using the Lanczos Method",
   SIAM J. Optim., 9(2), 504--525, (1999).
.. [16] Byrd, Richard H., Mary E. Hribar, and Jorge Nocedal. 1999.
    An interior point algorithm for large-scale nonlinear  programming.
    SIAM Journal on Optimization 9.4: 877-900.
.. [17] Lalee, Marucha, Jorge Nocedal, and Todd Plantega. 1998. On the
    implementation of an algorithm for large-scale equality constrained
    optimization. SIAM Journal on Optimization 8.3: 682-706.

Examples
--------
Let us consider the problem of minimizing the Rosenbrock function. This
function (and its respective derivatives) is implemented in `rosen`
(resp. `rosen_der`, `rosen_hess`) in the `scipy.optimize`.

>>> from scipy.optimize import minimize, rosen, rosen_der

A simple application of the *Nelder-Mead* method is:

>>> x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
>>> res = minimize(rosen, x0, method='Nelder-Mead', tol=1e-6)
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])

Now using the *BFGS* algorithm, using the first derivative and a few
options:

>>> res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
...                options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 26
         Function evaluations: 31
         Gradient evaluations: 31
>>> res.x
array([ 1.,  1.,  1.,  1.,  1.])
>>> print(res.message)
Optimization terminated successfully.
>>> res.hess_inv
array([[ 0.00749589,  0.01255155,  0.02396251,  0.04750988,  0.09495377],  #
```

```
may vary
          [ 0.01255155,  0.02510441,  0.04794055,  0.09502834,  0.18996269],
          [ 0.02396251,  0.04794055,  0.09631614,  0.19092151,  0.38165151],
          [ 0.04750988,  0.09502834,  0.19092151,  0.38341252,  0.7664427 ],
          [ 0.09495377,  0.18996269,  0.38165151,  0.7664427,   1.53713523]])


    Next, consider a minimization problem with several constraints (namely
    Example 16.4 from [5]_). The objective function is:

    >>> fun = lambda x: (x[0] - 1)**2 + (x[1] - 2.5)**2

    There are three constraints defined as:

    >>> cons = ({'type': 'ineq', 'fun': lambda x:  x[0] - 2 * x[1] + 2},
    ...         {'type': 'ineq', 'fun': lambda x: -x[0] - 2 * x[1] + 6},
    ...         {'type': 'ineq', 'fun': lambda x: -x[0] + 2 * x[1] + 2})

    And variables must be positive, hence the following bounds:

    >>> bnds = ((0, None), (0, None))

    The optimization problem is solved using the SLSQP method as:

    >>> res = minimize(fun, (2, 0), method='SLSQP', bounds=bnds,
    ...                constraints=cons)

    It should converge to the theoretical solution (1.4 ,1.7).
```

[10]:
```
statsmodels_ols_summary
```

[10]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.982
Model:                            OLS   Adj. R-squared:                  0.982
Method:                 Least Squares   F-statistic:                 1.080e+04
Date:                Tue, 06 Aug 2019   Prob (F-statistic):          1.13e-174
Time:                        11:13:41   Log-Likelihood:                -672.67
No. Observations:                 200   AIC:                             1349.
Df Residuals:                     198   BIC:                             1356.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
```

```
==============================================================================
constant      -1.4593      0.990     -1.474      0.142      -3.411       0.493
x             -8.8947      0.086   -103.903      0.000      -9.064      -8.726
==============================================================================
Omnibus:                        0.634   Durbin-Watson:                   1.803
Prob(Omnibus):                  0.728   Jarque-Bera (JB):                0.415
Skew:                          -0.099   Prob(JB):                        0.813
Kurtosis:                       3.101   Cond. No.                         23.2
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
"""
```

[26]:
```
np.round(resultsdf.head(2), 4)
```

[26]:
```
             coef
constant -0.5108
x        -9.1022
```

[ ]:
```

```