# Maximum Likelihood Estimation and discrete choice models

*Monday 23, September*

## Content ¶



## 1. Short intro to MLE

Maximum likelihood is an intuitive and popular method of statistical inference. It involves choosing a set of parameters such that, when fed to a Data Generating Process (DGP), they match the observed moments of the data. The main advantages of maximum likelihood estimation are:

- its efficiency: It achieves the **Cramer-Rao** lower bound when $n \rightarrow \infty$
- its consistency: The ML estimator converges in probability to the true parameter $\hat{\theta}_{MLE} \xrightarrow{p} \theta$, and even almost surely under stricter regularity conditions
- its versatility: Any DGP can be estimated by MLE. It is more versatile than linear regression as it allows for more intricate relationships between variables.

Its main drawback is that it requires us to assume a specific parametric distribution of the data.

## 2. Coding the MLE estimator

### 2.1. The linear case

Let's create a random sample of size $N = 1,000$, generated by the following DGP:

$$\mathbf{y} = \mathbf{X}\beta + \varepsilon$$

$$\varepsilon \sim \mathcal{N}(0, \sigma^2 I)$$

The normality of errors and their 0-correlation ensure they are independent, a property we will use when implementing the MLE method.

In [1]:

```
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mp
%matplotlib nbagg
import matplotlib.pyplot as plt

import statsmodels as sm
```

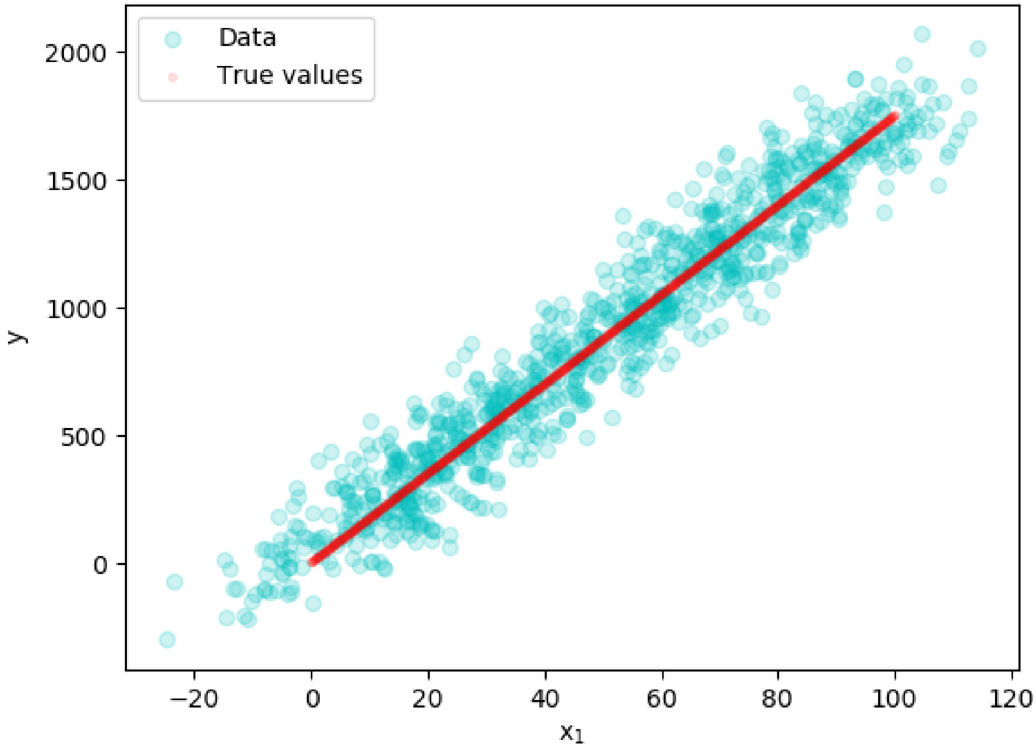
In [2]:

```
# True values of X's will be linear functions measured without noise
X1_true = np.linspace(0, 100, 1000)
X2_true = np.linspace(0, -500, 1000)
X3_true = np.linspace(0, 2000, 1000)

# We also add a vector of ones to the matrix of observables, this will allow us to incl
ude an intercept in the model
X0 = np.ones(1000)

X_true = np.array([X0,
              X1_true,
              X2_true,
              X3_true]).T

# We define observed covariates as the sum of the true X's + normal noise (can be inter
preted as measurement error or simply as a way to avoid perfect multicolinearity)
np.random.seed(123)
X1 = X1_true + np.random.normal(loc=0.0, scale = 10, size=1000)
X2 = X2_true + np.random.normal(loc=0.0, scale = 50, size=1000)
X3 = X3_true + np.random.normal(loc=0.0, scale = 200, size=1000)

X = np.array([X0,
              X1,
              X2,
              X3]).T

# True parameters
β = np.array([5, 10, -.5, .25]).T
sigma = 100
epsilon = np.random.normal(loc=0.0, scale = sigma, size=1000)

# Measured and true values
y = X @ β + epsilon
y_true = X_true @ β
```

We get a simple linear, homoskedastic relastionship between the covariates and the observed $y_i$. See for instance $\mathbf{y}$ with respect to $\mathbf{x_1}$ below

In [3]:

```
plt.scatter(X1, y, c="c", alpha=0.2, label="Data")
plt.scatter(X1_true, y_true, c="r", alpha=0.1, marker='.', label = "True values")
plt.xlabel("$\mathrm{x_1}$")
plt.ylabel("$\mathrm{y}$")
plt.legend()
```




Out[3]:

```
<matplotlib.legend.Legend at 0x1b5be229e48>
```

OLS is BLUE in this case. Now if we estimate the coefficients by OLS we get $\hat{\beta}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$

In [4]:

```
β_ols = np.linalg.inv((X.T @ X)) @ (X.T @ y)
β_ols
```

Out[4]:

```
array([11.83699744,  9.65608284, -0.57041114,  0.24929886])
```

Now under normality of errors, OLS and MLE estimates of the coefficients are asymptotically equal. Let's define an iterative procedure to estimate $\boldsymbol{\beta}$ through MLE and verify that the results are in line with OLS. When errors are normally distributed, we do not need to go through such lengths as the MLE estimator has a closed-form solution. But we simply show how the convergence algorithm works in this simple case.

Our true parameter values are

$$\boldsymbol{\beta} = \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 5 \\ 0.5 \\ 0.25 \end{bmatrix}, \sigma = 100$$

We define a likelihood function, based on $N$ draws of the data

$$f\left(\mathbf{y}|\mathbf{X};\boldsymbol{\beta},\sigma^2\right) = \prod_{i=1}^{N} f\left(y_i|x_i,\boldsymbol{\beta},\sigma^2\right)$$

where we assume that errors are normally i.i.d, so

$$f\left(y_i|x_i,\boldsymbol{\beta},\sigma^2\right) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\frac{1}{2\sigma^2}\left(y_i - x_i'\beta\right)^2}$$

We further define the likelihood function $\mathcal{L}(\boldsymbol{\beta}|\mathbf{y},\mathbf{X})$ which treats the parameters $\boldsymbol{\beta}$ and $\sigma$ as random and the values $\mathbf{y},\mathbf{X}$ as given. MLE consists in maximising the value of $\mathcal{L}(\boldsymbol{\beta},\sigma|\mathbf{y},\mathbf{X})$ by chosing $\boldsymbol{\beta}$ and $\sigma$ optimally. It is easier to work with the log of the likelihood function and this does not affect the maximiser as any monotonically increasing transformation of a function has the same maximiser.

We thus maximise:

$$\begin{aligned} \ln \mathcal{L}\left(\boldsymbol{\beta},\sigma|\mathbf{y},\mathbf{X}\right) =& \ell\left(\boldsymbol{\beta},\sigma|\mathbf{y},\mathbf{X}\right) \\ =& -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln\left(\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y_i - x_i'\beta\right)^2 \end{aligned}$$

And our solution is:

$$\left(\hat{\beta}_{MLE}, \hat{\sigma}_{MLE}\right) = \underset{\theta,\beta}{\operatorname{argmax}} \left\{ -\frac{n}{2}\ln(2\pi) - \frac{n}{2}\ln\left(\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}\left(y_i - x_i'\beta\right)^2 \right\}$$

### 2.2. Manual coding of the MLE problem

We now turn to coding the problem stated above. We aim to obtain MLE estimates based on the dataset created at the beginning of the Notebook.

We transform the problem into a minimisation one. Minimisation problems are more numerically stable. We will call a Scipy function for our proble: `scipy.optimize.minimize` .

The Scipy `minimize` function provides a vast collection of constrained and unconstrained minimizations algorithms for multivariate scalar functions. The default algorithm `minimize` resorts to when solving an unconstrained optimization problem (like ours) is the [Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm (https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algori](https://en.wikipedia.org/wiki/Broyden%E2%80%93Fletcher%E2%80%93Goldfarb%E2%80%93Shanno_algori) which is also used by MATLAB and R's optimisation routines. Another popular option is the [Nelder-Mead method (https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method)](https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method), whose robustness to many types of objective functions makes it a very versatile -but slow- algorithm. You can pass the algorithm you want use as an argument of `minimize` , based on your application.

We will simply use three arguments in our application of `minimize` :

- The function to be minimised, called the criterion (here $-\ell\left(\boldsymbol{\beta},\sigma|\mathbf{y},\mathbf{X}\right)$)

- An initial guess for the value of the $\beta$ and $\sigma$ parameters
- The data needed to be fed to the criterion

We first encode the criterion below. We need to be careful about the type of arguments taken by this function. Whether the parameters need to be entered as tuples, arrays, or lists is dictated by the how the `minimize` function works.

In [5]:

```
def negLogLNorm(params, *args):
    '''
    --------------------------------------------------------------------
    Calculate -log(likelihood) using data points "data" and parameters
    "params".
    The log(likelihood) is calculated based on the assumption that
    errors are normally distributed.
    --------------------------------------------------------------------

    INPUTS:
    params  =   numpy array with 2 elements: one k*1 vector of beta
                coefficients (including intercept), and one scalar for
                sigma
    data    =   numpy array with two elements: one N*1 vector of
                observations of the dependent variable, and one N*k
                matrix of observations of the dependent variables

    RETURNS:
    neg_log_lik_val =   The negative of the log likelihood, using the
                        assumption that errors are normally distributed
    --------------------------------------------------------------------
    '''

    # Fetching parameters and data (note: args and param are tuples)
    b0, b1, b2, b3, sigma = params
    β = np.array([b0, b1, b2, b3])    #Transforming the tuple of coefficients into an N
umpy array to be used in matrix multiplication
    y, X = args[0], args[1]

    # Calculating bits of the log-likelihood
    E = y - X @ β                     # vector of errors
    SSE = np.square(E).sum()          # sum of squared errors
    N = len(y)                        #sample size

    # Define the log likelihood function
    log_likelihood = - (N/2)*np.log(2 * np.pi) - (N/2)*np.log(sigma**2) - (1/(2 * sigma
**2))*SSE

    neg_log_likelihood = - log_likelihood

    return neg_log_likelihood
```

We can now use the `minimize` function. We import the `optimize` library under a convenient name, define some initial parameter values (our guesses), and define the data to be used.

In [6]:

```
import scipy.optimize as opt

β_0 = np.array([0, 0, 0, 0, 1])
data = (y, X)
MLE_results = opt.minimize(negLogLNorm, β_0, args=(data))
MLE_results
```

Out[6]:

```
      fun: 6011.9441161417335
 hess_inv: array([[ 3.79945576e-02,  2.96758209e-03, -1.54961040e-04,
        -2.06139340e-04, -1.43280697e-02],
       [ 2.96758209e-03,  1.96742993e-02, -1.10619039e-03,
        -1.23665279e-03, -5.10596635e-03],
       [-1.54961040e-04, -1.10619039e-03,  6.31791633e-05,
         6.93641915e-05,  2.82749051e-04],
       [-2.06139340e-04, -1.23665279e-03,  6.93641915e-05,
         8.43728092e-05,  3.34738415e-04],
       [-1.43280697e-02, -5.10596635e-03,  2.82749051e-04,
         3.34738415e-04,  1.63755838e-02]])
      jac: array([0.        , 0.        , 0.        , 0.00067139, 0.
])
  message: 'Desired error not necessarily achieved due to precision loss.'
     nfev: 999
      nit: 88
     njev: 141
   status: 2
  success: False
        x: array([11.83737456,  9.65608084, -0.57041128,  0.24929865, 98.7
9078009])
```

`minimize` returns our estimated parameters ( `x` ), and much more, as an `OptimizeResult` object. We get the value of the negative log likelihood, Jacobian and Hessian matrices at the solution, whether the optimiser has converged (in our case it hasn't).

As anticipated, the estimated parameters are the same as those found via OLS. Our encoding of MLE has worked.

If you are interested in coding your own optimisation algorithm, the [Quant-Econ MLE lecture (https://lectures.quantecon.org/py/mle.html)](https://lectures.quantecon.org/py/mle.html) has a nice explanation and implementation of the [Newton-Raphson algorithm (https://en.wikipedia.org/wiki/Newton%27s_method)](https://en.wikipedia.org/wiki/Newton%27s_method).

### 2.3. Variance-covariance of $\hat{\theta}_{MLE}$

Note that we can easily obtain the variance-covariance matrix of our MLE estimator, as the inverse of the estimated Hessian is reported in the output above. Formally, the variance-covariance matrix of $\hat{\theta}_{MLE}$ is:

$$\text{var}(\theta) = \left(-E\left[\frac{\partial^2 \ln \mathcal{L}(\theta)}{\partial\theta\partial\theta'}\right]\right)^{-1}$$

We get it as follows:

In [7]:

```
var = MLE_results.hess_inv
se_alpha = np.sqrt(var[0, 0])
se_beta1 = np.sqrt(var[1, 1])
se_beta2 = np.sqrt(var[2, 2])
se_beta3 = np.sqrt(var[3, 3])
se_sigma = np.sqrt(var[4, 4])

print('SE(alpha) = ', se_alpha,
      '\nSE(beta_1) =', se_beta1,
      '\nSE(beta_2) =', se_beta2,
      '\nSE(beta_3) =', se_beta3,
      '\nSE(sigma) =', se_sigma)
```

```
SE(alpha) =  0.1949219269573525
SE(beta_1) = 0.14026510380019627
SE(beta_2) = 0.007948532145159698
SE(beta_3) = 0.009185467285021886
SE(sigma) = 0.12796711991784762
```

#### 2.4. Constrained optimisation with `minimize`

`minimize` is also equipped with optimisation algorithms designed to deal with constrains. Let's say we believe that the intercept $\alpha$ must be between 5 and 6. The `trust-constr` (Trust-Region Constrained algorithm), `L-BFGS-B` (Limited memory BFGS with Box constraints), `TNC` (Truncated Newton, implemented in C), and `SLSQP` (Sequential Least SQuares Programming) methods can handle these constraints. See more information on these methods [here (https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html)](https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html). They are implemented by calling `method='chosenMethod'` as an argument of `minimize`.

To define our constraint $5 < \alpha < 6$, we actually need to define a system of inequalities over all parameters. For instance

$$\begin{bmatrix} 5 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \sigma \end{bmatrix} \leq \begin{bmatrix} 6 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Where only the first inequality could be binding.

This is how we implement it:

In [8]:

```
#Defining the constraint with LinearConstraint
from scipy.optimize import LinearConstraint
cons = LinearConstraint([[1, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0],
                         [0, 0, 0, 0, 0]],
                        [5, 1, 1, 1, 1],
                        [6, 1, 1, 1, 1])
```

In [9]:

```
results_cons = opt.minimize(negLogLNorm, β_0, args=(data), method='trust-constr', const
raints=[cons])
results_cons
```

```
C:\Users\dyevre\AppData\Local\Continuum\anaconda3\lib\site-packages\scipy
\optimize\_trustregion_constr\projections.py:182: UserWarning: Singular Ja
cobian matrix. Using SVD decomposition to perform the factorizations.
  warn('Singular Jacobian matrix. Using SVD decomposition to ' +
```

Out[9]:

```
 barrier_parameter: 0.1
 barrier_tolerance: 0.1
          cg_niter: 1000
      cg_stop_cond: 1
            constr: [array([5.06085929, 0.        , 0.        , 0.        
, 0.        ])]
       constr_nfev: [0]
       constr_nhev: [0]
       constr_njev: [0]
    constr_penalty: 387488835.11958194
  constr_violation: 1.0
    execution_time: 1.086634635925293
               fun: 57665.39881679692
              grad: array([  -45.31455124, -6556.96986293,  1107.88867302,
-7204.29541016,
       -9233.67633018])
               jac: [array([[1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]])]
   lagrangian_grad: array([-1.72169286e-01, -6.55696986e+03,  1.10788867e+
03, -7.20429541e+03,
       -9.23367633e+03])
           message: 'The maximum number of function evaluations is exceede
d.'
            method: 'tr_interior_point'
              nfev: 6006
              nhev: 0
               nit: 1001
             niter: 1001
              njev: 0
        optimality: 9233.676330184455
            status: 0
           success: False
         tr_radius: 22.74523204167703
                 v: [array([ 4.51423820e+01, -0.00000000e+00,  5.54982333e
-15, -0.00000000e+00,
       -0.00000000e+00])]
                 x: array([ 5.06085929,  3.85986813, -1.21915233,  0.37526
718, 11.65106079])
```

## 3. MLE with 'statsmodels'

It can be painful to code every single aspect of the Maximum Likelihood Estimation, and to fetch every interesting result manually. Fortunately, the `statsmodels` (https://www.statsmodels.org/stable/index.html) package has an extensive catalogue of tools for statistical inference, including MLE. This package also allows us to generate a much richer set of outputs very easily: confidence intervals, p-statistics, pseudo- R-squared are all generated by default.

STATA and R users will feel at home with this package as estimations results are presented in a similar way. However, using the package proficiently may sometimes require a good grasp of object-oriented programming. The time invested in getting familiar with object-oriented programming is definitely worth it, as it will make

#### 3.1. Hacking the `GenericLikelihoodModel` class from `statsmodels`

Let's first reproduce our results above with `statsmodels` , this will give us a feel of what output is generated. For this, we will need to define a new model class that inherits from `statsmodels` ' `GenericLikelihood Model` 's attributes. This class uses a log-likelihood defined ex-ante, and will be estimated by `statsmodels` algorithms. When defining a custom model relying on maximum likelihood, we need to respect the standard architecture of the `GenericLikelihoodModel` cannon (see here (https://www.statsmodels.org/dev/examples/notebooks/generated/generic_mle.html) for an example).

In [10]:

```
from scipy.stats import norm
import statsmodels.api as sm
from statsmodels.base.model import GenericLikelihoodModel

# We first define a log-likelihood, which will be fed to our custom class `MLE_for_OLS`
def logL(y, X, β, σ):
    y_hat = X @ β
    return norm(y_hat, σ).logpdf(y).sum()   # This is a shorter encoding of the log lik
elihood than the one above

# Now we build an MLE solver, using the 'GenericLikelihoodModel' class
class MLE_for_OLS(GenericLikelihoodModel):
    def __init__(self, endog, exog, **kwargs):
        super(MLE_for_OLS, self).__init__(endog, exog, **kwargs) # When we don't know h
ow many variable arguments can be passed on to the function, we use *args
                                                           # We use **kwargs instead wh
en we want a named list of arguments (a dictionary)
    def nloglikeobs(self, params):
        σ = params[-1]
        β = params[:-1]
        ll = logL(self.endog, self.exog, β, σ)
        return -ll

    def fit(self, start_params = None, maxiter = 10000, maxfun = 10000, **kwargs):
        # We need to add the σ to the list of parameters to be estimated
        self.exog_names.append('σ')
        if start_params == None:                                 # Default starting values,
 if none specified
            start_params = np.append(np.zeros(self.exog.shape[1]), 1)
        return super(MLE_for_OLS, self).fit(start_params=start_params,
                             maxiter=maxiter, maxfun=maxfun,
                             **kwargs)
```

And we now print the results.

In [11]:

```
sm_ols_manual = MLE_for_OLS( y, X).fit()
print(sm_ols_manual.summary())
```

```
Optimization terminated successfully.
         Current function value: 6.014743
         Iterations: 654
         Function evaluations: 1072
                             MLE_for_OLS Results
==========================================================================
====
Dep. Variable:                      y   Log-Likelihood:                -60
14.7
Model:                    MLE_for_OLS   AIC:                         1.204
e+04
Method:            Maximum Likelihood   BIC:                         1.206
e+04
Date:                Thu, 12 Sep 2019
Time:                        22:44:28
No. Observations:                1000
Df Residuals:                     996
Df Model:                           3
==========================================================================
====
                 coef    std err          z      P>|z|      [0.025      0.
975]
--------------------------------------------------------------------------
----
const         -2.8253      6.242     -0.453      0.651     -15.059
9.408
x1             9.7120      0.260     37.282      0.000       9.201      1
0.223
x2            -0.5824      0.054    -10.760      0.000      -0.689      -
0.476
x3             0.2544      0.013     19.408      0.000       0.229
0.280
σ             99.0605      2.227     44.476      0.000      94.695     10
3.426
==========================================================================
====
```

These results are very close to the ones we have found via manual encoding of MLE, and with OLS.

### 3.2. Exporting results to $\LaTeX$

The `as_latex()` command allows you to export your results in a neatly formatted TeX table. See the results of the last output when we write `print(sm_ols_manual.summary().as_latex())` . You will need the `booktabs` package for the table to compile in TeX.

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **Log-Likelihood:** | -6014.7 |
| **Model:** | MLE_for_OLS | **AIC:** | 1.204e+04 |
| **Method:** | Maximum Likelihood | **BIC:** | 1.206e+04 |
| **Date:** | Thu, 05 Sep 2019 | | |
| **Time:** | 15:31:09 | | |
| **No. Observations:** | 1000 | | |
| **Df Residuals:** | 996 | | |

| | **coef** | **std err** | **z** | **P> \|z\|** | **[0.025** | **0.975]** |
|---|---|---|---|---|---|---|
| **const** | -2.8255 | 6.242 | -0.453 | 0.651 | -15.060 | 9.409 |
| **x1** | 9.7119 | 0.261 | 37.282 | 0.000 | 9.201 | 10.223 |
| **x2** | -0.5823 | 0.054 | -10.757 | 0.000 | -0.688 | -0.476 |
| **x3** | 0.2545 | 0.013 | 19.410 | 0.000 | 0.229 | 0.280 |
| $\sigma$ | 99.0619 | 2.227 | 44.475 | 0.000 | 94.696 | 103.427 |

We can also print several estimation results next to each other, as is standard in academic papers. To this end, we use the `summary_col` command.

In [12]:

```
from statsmodels.iolib.summary2 import summary_col

# We define an empty list, it will contain the regression outputs generated by our MLE
 estimator
results = []

# We add one regressor at a time, and store the outputa in 'results'
for i in range(1, 5):
    col = MLE_for_OLS(y, X.T[0:i].T).fit()
    results.append(col)
```

```
Optimization terminated successfully.
         Current function value: 7.676231
         Iterations: 204
         Function evaluations: 398
Optimization terminated successfully.
         Current function value: 6.332248
         Iterations: 312
         Function evaluations: 554
Optimization terminated successfully.
         Current function value: 6.167039
         Iterations: 571
         Function evaluations: 968
Optimization terminated successfully.
         Current function value: 6.014743
         Iterations: 654
         Function evaluations: 1072
```

In [13]:

```
# We now use the functionalities of 'summary_col' to get a nicely formated table
summary = summary_col(results = results,
                      stars = True,
                      model_names = ['intercept', '2 variables', '3 variables', 'full'
],
                      info_dict = {'Observations': lambda x: f"{int(x.nobs):d}"},
                      float_format='%0.3f')
summary.add_title('OLS by MLE')
print(summary)
```

```
                        OLS by MLE
=========================================================
             intercept  2 variables 3 variables   full
---------------------------------------------------------
const        882.999*** 58.053***   31.344***   -2.825
             (16.501)   (8.258)     (7.129)     (6.242)
x1                      16.631***   11.724***   9.712***
                        (0.142)     (0.276)     (0.260)
x2                                  -1.082***   -0.582***
                                    (0.055)     (0.054)
x3                                              0.254***
                                                (0.013)
σ            521.804*** 136.089***  115.365***  99.061***
             (11.668)   (3.043)     (2.580)     (2.227)
Observations 1000       1000        1000        1000
=========================================================
Standard errors in parentheses.
* p<.1, ** p<.05, ***p<.01
```

And the TeX version, given by `print(summary.as_latex())`

Table 1: OLS by MLE

| | intercept | 2 variables | 3 variables | full |
|---|---|---|---|---|
| const | 882.999*** | 58.053*** | 31.344*** | -2.825 |
| | (16.501) | (8.258) | (7.129) | (6.241) |
| x1 | | 16.631*** | 11.724*** | 9.712*** |
| | | (0.142) | (0.276) | (0.260) |
| x2 | | | -1.082*** | -0.582*** |
| | | | (0.055) | (0.054) |
| x3 | | | | 0.254*** |
| | | | | (0.013) |
| $\sigma$ | 521.804*** | 136.089*** | 115.365*** | 99.061*** |
| | (11.668) | (3.043) | (2.580) | (2.227) |
| Observations | 1000 | 1000 | 1000 | 1000 |

## 4. Discrete choice models with 'statsmodels'

The strength of `statmodels` lies in the breadth of its model library. All necessary tools for performing discrete choice analysis such as logit, probit, multinomial logit, Poisson or negative binomial come in canned commands.

We just show an application of the negative binomial model, as an example. All other models mentioned above are similarly implemented and we redirect you the [`statsmodels` documentation (https://www.statsmodels.org/dev/examples/notebooks/generated/discrete_choice_overview.html)](https://www.statsmodels.org/dev/examples/notebooks/generated/discrete_choice_overview.html) on discrete choice models for more information.

Negative binomial regression is used to model count variables when the dependent variable is overdispersed. It is ideal when the dependent variable has a an excess count of zeros for instance. The probability mass function (PMF) of the negative binomial distribution gives us the probability that in a sequence of Bernoulli trials, $k$ successes have occured when $r$ failures have occured. In other words, if a Bernoulli trial with probability $p$ of success has been repeated until $r$ failures have occured, the number of successes $X$ will be negative Binomial:

$$X \sim \mathrm{NB}(r, p)$$

and the PMF is

$$f(k|r,p) \equiv \Pr(X=k) = \binom{k+r-1}{k}(1-p)^r p^k$$

We first import a native 'statsmodels' dataset: "RAND". It was collected by the RAND corporation as part of a US country-wide health insurance study (1971-1986). The dependent variable is the number of visits to a doctor in a year, for a given individual. We will try to explain the number of visits by using variables such as insurance coverage, self-rated health, and number of chronic diseases.

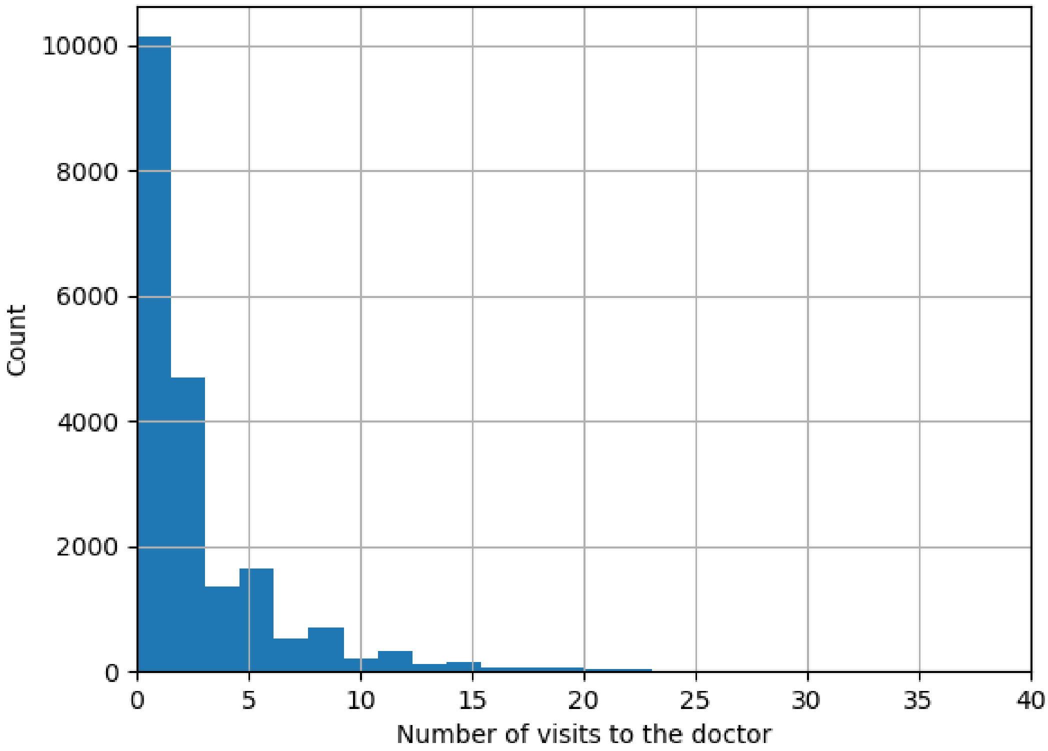
In [14]:

```
# Whole data
data = sm.datasets.randhie.load(as_pandas = False)

# Defining the set of exogenous variables, and adding an intercept
exog = data.exog.view(float).reshape(len(data.exog), -1)
exog = sm.add_constant(exog, prepend = False)
```

Our dependent variable is densely distributed around 0, which makes it a good candidate for the negative binomial model.

In [15]:

```
plt.figure(2)
plt.hist(data.endog, bins=50)
plt.xlim(xmin=0, xmax=40)
plt.grid()
plt.xlabel('Number of visits to the doctor')
plt.ylabel('Count')
plt.show()
```




We now fit the model:

In [16]:

```
results_NBin = sm.NegativeBinomial(data.endog, exog).fit()
results_NBin.summary()
```

```
Warning: Maximum number of iterations has been exceeded.
         Current function value: 2.148770
         Iterations: 35
         Function evaluations: 39
         Gradient evaluations: 39

C:\Users\dyevre\AppData\Local\Continuum\anaconda3\lib\site-packages\statsm
odels\base\model.py:512: ConvergenceWarning: Maximum Likelihood optimizati
on failed to converge. Check mle_retvals
  "Check mle_retvals", ConvergenceWarning)
```

Out[16]:

NegativeBinomial Regression Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **No. Observations:** | 20190 |
| **Model:** | NegativeBinomial | **Df Residuals:** | 20180 |
| **Method:** | MLE | **Df Model:** | 9 |
| **Date:** | Thu, 12 Sep 2019 | **Pseudo R-squ.:** | 0.01845 |
| **Time:** | 22:44:54 | **Log-Likelihood:** | -43384. |
| **converged:** | False | **LL-Null:** | -44199. |
| **Covariance Type:** | nonrobust | **LLR p-value:** | 0.000 |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **x1** | -0.0579 | 0.006 | -9.515 | 0.000 | -0.070 | -0.046 |
| **x2** | -0.2678 | 0.023 | -11.802 | 0.000 | -0.312 | -0.223 |
| **x3** | 0.0412 | 0.004 | 9.938 | 0.000 | 0.033 | 0.049 |
| **x4** | -0.0381 | 0.003 | -11.216 | 0.000 | -0.045 | -0.031 |
| **x5** | 0.2691 | 0.030 | 8.985 | 0.000 | 0.210 | 0.328 |
| **x6** | 0.0382 | 0.001 | 26.080 | 0.000 | 0.035 | 0.041 |
| **x7** | -0.0441 | 0.020 | -2.201 | 0.028 | -0.083 | -0.005 |
| **x8** | 0.0173 | 0.036 | 0.478 | 0.632 | -0.054 | 0.088 |
| **x9** | 0.1782 | 0.074 | 2.399 | 0.016 | 0.033 | 0.324 |
| **const** | 0.6635 | 0.025 | 26.786 | 0.000 | 0.615 | 0.712 |
| **alpha** | 1.2930 | 0.019 | 69.477 | 0.000 | 1.256 | 1.329 |