Pract_regression_professor

October 3, 2023

1 Parametric ML and Bayesian regression

```
Notebook version: 1.2 (Sep 28, 2018)

Authors: Miguel Lázaro Gredilla
         Jerónimo Arenas García (jarenas@tsc.uc3m.es)
         Jesús Cid Sueiro (jesus.cid@uc3m.es)

Changes: v.1.0 - First version. Python version
         v.1.1 - Python 3 compatibility. ML section.
         v.1.2 - Revised content. 2D visualization removed.

Pending changes:
```

[1]:
```python
# Import some libraries that will be necessary for working with data and␣
 ↪displaying plots

# To visualize plots in the notebook
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm

import numpy as np
import scipy.io       # To read matlab files
from scipy import spatial
import pylab
pylab.rcParams['figure.figsize'] = 8, 5
```

1.1 1. Introduction

In this exercise the student will review several key concepts of Maximum Likelihood and Bayesian regression. To do so, we will assume the regression model

$$s = \mathbf{w}^\top \mathbf{z} + \varepsilon$$

where s is the output corresponding to input $\mathbf{x}$, $\mathbf{z} = T(\mathbf{x})$ is a possibly non-linear transformation of the input, and ε is white zero-mean Gaussian noise, i.e.,

$$\varepsilon \sim N(0, \sigma_\varepsilon^2).$$

Along this notebook, we will explore different types of transformations.

Also, we will assume an a priori distribution for $\mathbf{w}$ given by

$$\mathbf{w} \sim N(\mathbf{0}, \sigma_p^2 \, \mathbf{I})$$

1.1.1 Practical considerations

- Though sometimes unavoidable, it is recommended not to use explicit matrix inversion whenever possible. For instance, if an operation like $\mathbf{A}^{-1}\mathbf{b}$ must be performed, it is preferable to code it using python `numpy.linalg.lstsq` function (see http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html), which provides the LS solution to the overdetermined system $\mathbf{A}\mathbf{w} = \mathbf{b}$.
- [*Not needed*] Sometimes, the computation of $\log|\mathbf{A}|$ (where $\mathbf{A}$ is a positive definite matrix) can overflow available precision, producing incorrect results. A numerically more stable alternative, providing the same result is $2\sum_i \log([\mathbf{L}]_{ii})$, where $\mathbf{L}$ is the Cholesky decomposition of $\mathbf{A}$ (i.e., $\mathbf{A} = \mathbf{L}^\top\mathbf{L}$), and $[\mathbf{L}]_{ii}$ is the ith element of the diagonal of $\mathbf{L}$.
- [*Not needed*] Non-degenerate covariance matrices, such as the ones in this exercise, are always positive definite. It may happen, as a consequence of chained rounding errors, that a matrix which was mathematically expected to be positive definite, turns out not to be so. This implies its Cholesky decomposition will not be available. A quick way to palliate this problem is by adding a small number (such as 10^{-6}) to the diagonal of such matrix.

1.1.2 Reproducibility of computations

To guarantee the exact reproducibility of the experiments, it may be useful to start your code initializing the seed of the random numbers generator, so that you can compare your results with the ones given in this notebook.

[2]:
```
np.random.seed(3)
```

1.2 2. Data generation with a linear model

During this section, we will assume affine transformation

$$\mathbf{z} = T(\mathbf{x}) = \begin{pmatrix} 1 \\ \mathbf{x} \end{pmatrix}$$

.

The a priori distribution of $\mathbf{w}$ is assumed to be

$$\mathbf{w} \sim N(\mathbf{0}, \sigma_p^2 \, \mathbf{I})$$

1.2.1 2.1. Synthetic data generation

First, we are going to generate synthetic data (so that we have the ground-truth model) and use them to make sure everything works correctly and our estimations are sensible.

- [1] Set parameters $\sigma_p^2 = 2$ and $\sigma_\varepsilon^2 = 0.2$. To do so, define variables `sigma_p` and `sigma_eps` containing the respective standard deviations.

[3]:
```
# Parameter settings
# sigma_p = <FILL IN>
sigma_p = np.sqrt(2)
# sigma_eps = <FILL IN>
sigma_eps = np.sqrt(0.2)
```

- [2] Generate a weight vector `true_w` with two elements from the *a priori* distribution of the weights. This vector determines the regression line that we want to find (i.e., the optimum unknown solution).

[4]:
```
# Data dimension:
dim_x = 2

# Generate a parameter vector taking a random sample from the prior␣
 ↪distributions
# (the np.random module may be usefull for this purpose)
# true_w = <FILL IN>
true_w = np.random.normal(0, sigma_p, (2,1))
# --> Alternatively, you can use true_w = sigma_p * np.random.randn(dim_x, 1)
print('The true parameter vector is:')
print(true_w)
```

```
The true parameter vector is:
[[2.52950265]
 [0.61731815]]
```

- [3] Generate an input matrix **X** (in this case, a single column) containing 20 samples with equally spaced values between 0 and 2 (method `linspace` from numpy can be useful for this)

[5]:
```
# <SOL>
# Parameter settings
x_min = 0
x_max = 2
n_points = 20

# Training datapoints
X = np.linspace(x_min, x_max, n_points)[:,np.newaxis]
# </SOL>
```

- [4] Finally, generate the output vector **s** as the product ${**Z**} **$ true_w plus Gaussian noise of pdf $N(0,\sigma_\varepsilon^2)$ at each element.**

[41]:
```
# Expand input matrix with an all-ones column
col_1 = np.ones((n_points, 1))
# Z = <FILL IN>
Z = np.hstack((col_1,X))

# Generate values of the target variable
# s = <FILL IN>
s = Z @ true_w + sigma_eps * np.random.randn(n_points, 1)
```

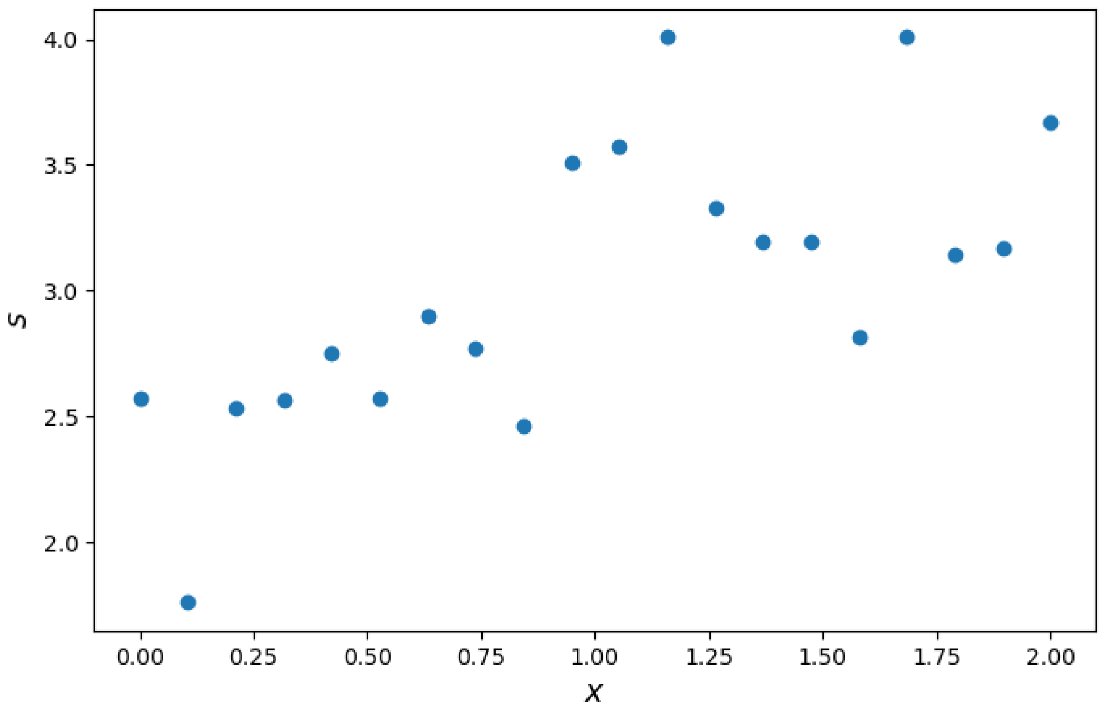
[7]:
```
print(s)
```

```
[[2.57265762]
 [1.76110423]
 [2.53541259]
 [2.56579218]
 [2.75242296]
 [2.57400371]
 [2.89979171]
 [2.77095026]
 [2.46177133]
 [3.50994552]
 [3.57344864]
 [4.0088364 ]
 [3.33164867]
 [3.19327656]
 [3.19534227]
 [2.81260983]
 [4.00852445]
 [3.14176482]
 [3.16918917]
 [3.67216952]]
```

1.2.2 2.2. Data visualization

- Plot the generated data. You will notice a linear behavior, but the presence of noise makes it hard to estimate precisely the original straight line that generated them (which is stored in `true_w`).

[8]:
```
# <SOL>
# Plot training points
plt.scatter(X, s);
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);
# </SOL>
```



1.3 3. Maximum Likelihood (ML) regression

1.3.1 3.1. Likelihood function

- [1] Define a function `predict(w, Z)` that computes the linear predictions for all inputs in data matrix `Z` (a 2-D numpy arry), for a given parameter vector `w` (a 1-D numpy array). The output should be a 1-D array. Test your function with the given dataset and `w = [0.4, 0.7]`

[9]:
```
# <SOL>
# Prediction function
def predict(Z, w):
    return Z @ w

w = np.array([0.4, 0.7])
p = predict(Z, w)
# </SOL>

# Print predictions
print(p)
```

```
[0.4        0.47368421 0.54736842 0.62105263 0.69473684 0.76842105
 0.84210526 0.91578947 0.98947368 1.06315789 1.13684211 1.21052632
 1.28421053 1.35789474 1.43157895 1.50526316 1.57894737 1.65263158
 1.72631579 1.8       ]
```

- [2] Define a function `sse(w, Z, s)` that computes the sum of squared errors (SSE) for the linear prediction with parameters `w` (1D numpy array), inputs `Z` (2D numpy array) and targets `s` (1D numpy array). Using this function, compute the SSE of the true parameter vector in `true_w`.

[10]:
```
# <SOL>
# Sum of Squared Errors
def sse(Z, s, w):
    return np.sum((s - predict(Z, w))**2)

SSE = sse(Z, s, true_w)
# </SOL>

print(f" The SSE is: {SSE:.5f}")
```

```
 The SSE is: 3.40036
```

- [3] Define a function `likelihood(w, Z, s, sigma_eps)` that computes the likelihood of parameter vector `w` for a given dataset in matrix `Z` and vector `s`, assuming Gaussian noise with varianze σ_ϵ^2. Note that this function can use the `sse` function defined above. Using this function, compute the likelihood of the true parameter vector in `true_w`.

[11]:
```
# <SOL>
# Likelihood function
def likelihood(w, Z, s, sigma_eps):
    K = len(s)
    lw = 1.0 / (np.sqrt(2*np.pi)*sigma_eps)**K * np.exp(- sse(Z, s, w)/
 ↪(2*sigma_eps**2))
    return lw

L_w_true = likelihood(true_w, Z, s, sigma_eps)
# </SOL>

print(f"The likelihood of the true parameter vector is {L_w_true:.5E}")
```

```
The likelihood of the true parameter vector is 2.07017E-05
```

- [4] Define a function `LL(w, Z, s, sigma_eps)` that computes the log-likelihood of parameter vector `w` for a given dataset in matrix `Z` and vector `s`, assuming Gaussian noise with varianze σ_ϵ^2. Note that this function can use the `likelihood` function defined above. However, for a higher numerical precission, implemening a direct expression for the log-likelihood is recommended.

  Using this function, compute the likelihood of the true parameter vector in `true_w`.

[12]:
```
# <SOL>
# The plot: LHS is the data, RHS will be the cost function.
def LL(w, Z, s, sigma_eps):
    K = len(s)
    Lw = - 0.5 * K * np.log(2*np.pi*sigma_eps**2) - sse(Z, s, w)/
↪(2*sigma_eps**2)
    return Lw

LL_w_true = LL(true_w, Z, s, sigma_eps)
# </SOL>

print(f"The log-likelihood of the true parameter vector is {LL_w_true:.5f}")
```

```
The log-likelihood of the true parameter vector is -10.78529
```

1.3.2 3.2. ML estimate

- [1] Compute the ML estimate of **w** given the data. Remind that using `np.linalg.lstsq` ia a better option than a direct implementation of the formula of the ML estimate, that would involve a matrix inversion.

[13]:
```
# <SOL>
w_ML, _, _, _ = np.linalg.lstsq(Z, s, rcond=None)

# </SOL>

print(w_ML)
```

```
[[2.39342127]
 [0.63211186]]
```

- [2] Compute the maximum likelihood, and the maximum log-likelihood.

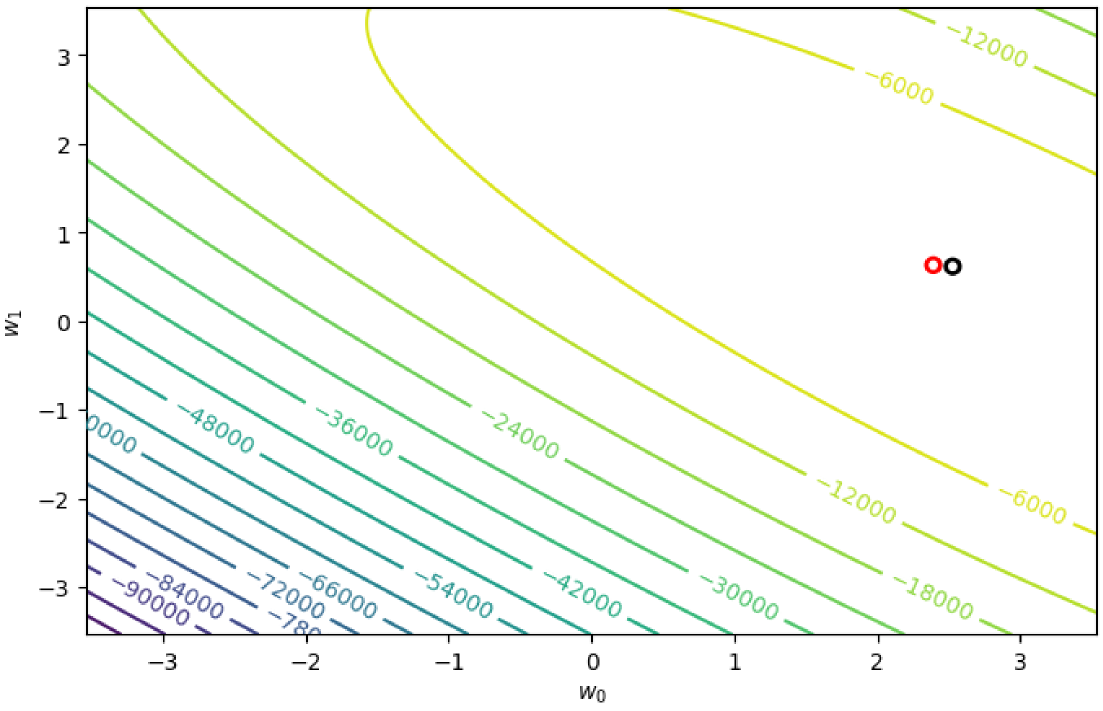
[14]:
```
# <SOL>
L_w_ML = likelihood(w_ML, Z, s, sigma_eps)
LL_w_ML = LL(w_ML, Z, s, sigma_eps)
# </SOL>

print(f'Maximum likelihood: {L_w_ML:.5E}')
print(f'Maximum log-likelihood: {LL_w_ML:.5f}')
```

```
Maximum likelihood: 4.33706E-05
Maximum log-likelihood: -10.04573
```

Just as an illustration, the code below generates a set of points in a two dimensional grid going from $(-\sigma_p, -\sigma_p)$ to (σ_p, σ_p), computes the log-likelihood for all these points and visualize them using a 2-dimensional plot. You can see the difference between the true value of the parameter **w** (black) and the ML estimate (red). If they are not quite close to each other, maybe you have made some mistake in the above exercises:

[15]:
```
# First construct a grid of (theta0, theta1) parameter pairs and their
# corresponding cost function values.
N = 200     # Number of points along each dimension.
```

```
w0_grid = np.linspace(-2.5*sigma_p, 2.5*sigma_p, N)
w1_grid = np.linspace(-2.5*sigma_p, 2.5*sigma_p, N)

Lw = np.zeros((N,N))
# Fill Lw with the likelihood values
for i, w0i in enumerate(w0_grid):
    for j, w1j in enumerate(w1_grid):
        we = np.array((w0i, w1j))
        Lw[i, j] = LL(we, Z, s, sigma_eps)

WW0, WW1 = np.meshgrid(w0_grid, w1_grid, indexing='ij')
contours = plt.contour(WW0, WW1, Lw, 20)

plt.figure
plt.clabel(contours)
plt.scatter([true_w[0]]*2, [true_w[1]]*2, s=[50,10], color=['k','w'])
plt.scatter([w_ML[0]]*2, [w_ML[1]]*2, s=[50,10], color=['r','w'])
plt.xlabel('$w_0$')
plt.ylabel('$w_1$')
plt.show()
```



1.3.3 3.3. [OPTIONAL]: Convergence of the ML estimate for the true model

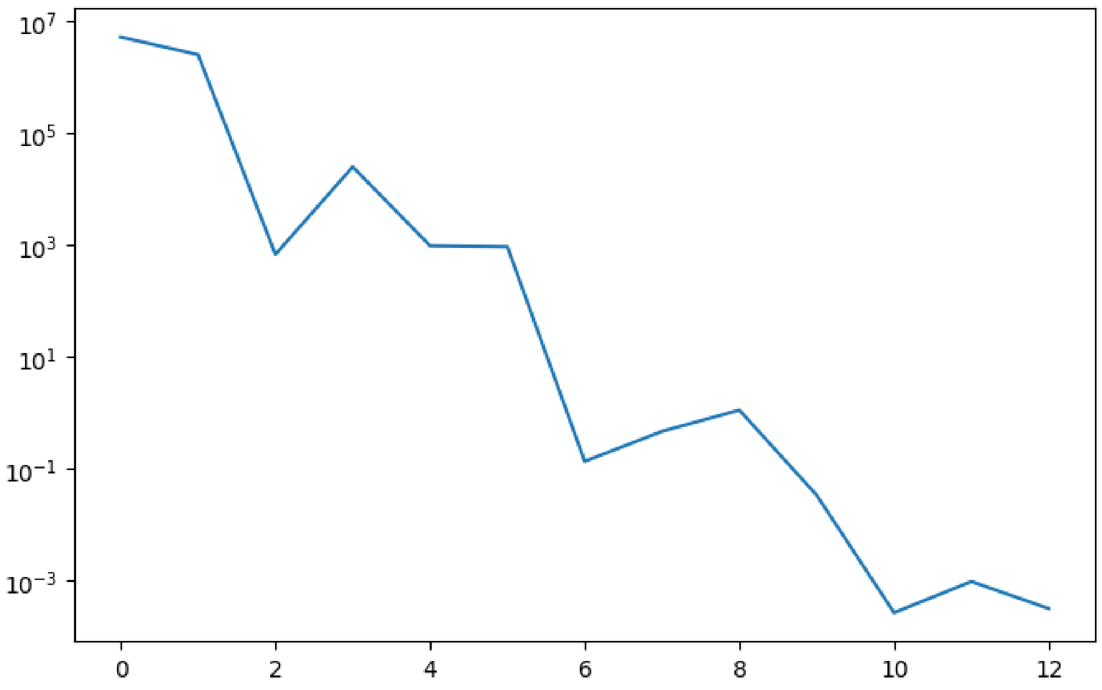
Note that the likelihood of the true parameter vector is, in general, smaller than that of the ML estimate. However, as the sample size increasis, both should converge to the same value.

- [1] Generate a longer dataset, with $K_{\text{max}} = 2^{16}$ samples, uniformly spaced between 0 and 2. Store it in the 2D-array `X2` and the 1D-array `s2`

[16]:
```
# Parameter settings
x_min = 0
x_max = 2
n_points = 2**16

# <SOL>
# Training datapoints
X2 = np.linspace(x_min, x_max, n_points)

# Expand input matrix with an all-ones column
col_1 = np.ones((n_points,))
Z2 = np.vstack((col_1, X2)).T
s2 = Z2 @ true_w + sigma_eps * np.random.randn(n_points, 1)
# </SOL>
```

- [2] Compute the ML estimate based on the first 2^k samples, for $k = 2, 3, \ldots, 15$. For each value of k compute the squared euclidean distance between the true parameter vector and the ML estimate. Represent it graphically (using a logarithmic scale in the y-axis).

[17]:
```
# <SOL>
e2 = []

for k in range(3, 16):
    Zk = Z2[0:2**k, :]
    sk = s2[0:2**k]

    w_MLk, _, _, _ = np.linalg.lstsq(Zk, sk, rcond=None)
    e2.append(np.sum((true_w - w_MLk)**2))

plt.semilogy(e2)
plt.show()
# </SOL>
```



1.4 4. ML estimation with real data. The stocks dataset.

Once our code has been tested on synthetic data, we will use it with real data.

1.4.1 4.1. Dataset

- [1] Load the dataset file provided with this notebook, corresponding to the evolution of the stocks of 10 airline companies. (The dataset is an adaptation of the Stock dataset, which in turn was taken from the StatLib Repository)

[18]:
```
# <SOL>
matvar = scipy.io.loadmat('DatosLabReg.mat')
Xtrain = matvar['Xtrain']
Xtest = matvar['Xtest']
strain = matvar['Ytrain']
stest = matvar['Ytest']
# </SOL>
```

- [2] Normalize the data so all training sample components have zero mean and unit standard deviation. Store the normalized training and test samples in 2D numpy arrays `Xtrain` and `Xtest`, respectively.

[19]:
```
# <SOL>
# Data normalization
mean_x = np.mean(Xtrain, axis=0)
```

```
std_x = np.std(Xtrain, axis=0)
Xtrain = (Xtrain - mean_x) / std_x
Xtest = (Xtest - mean_x) / std_x
# </SOL>
```

1.4.2 4.2. Polynomial ML regression with a single variable

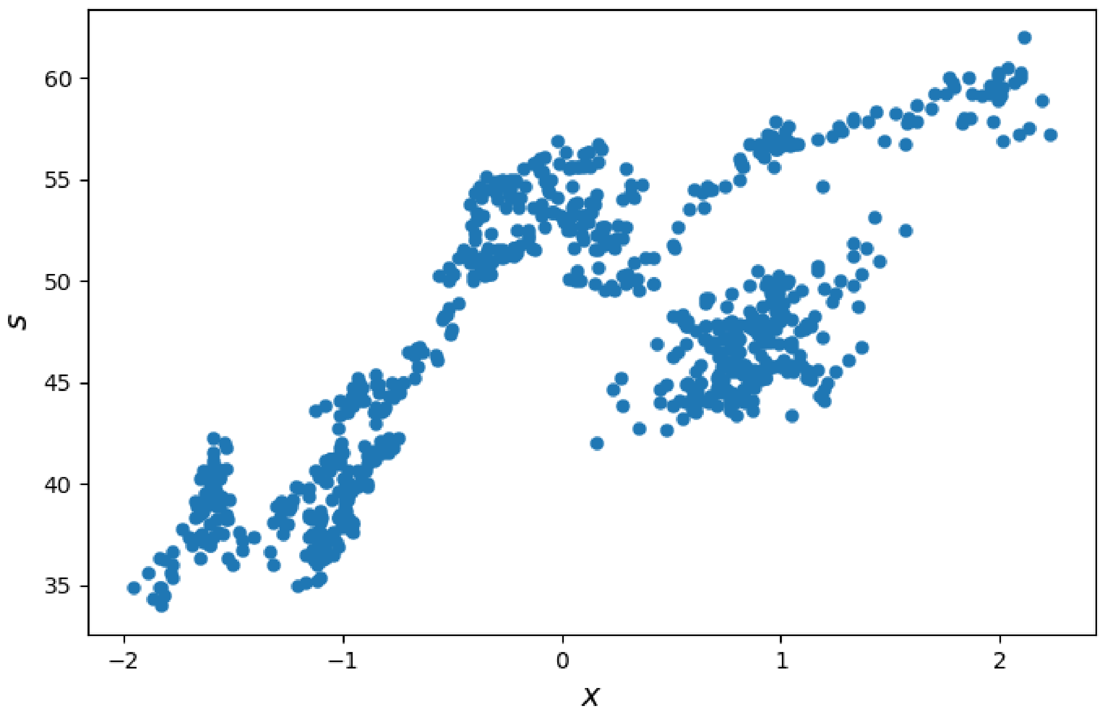
In this first part, we will work with the first component of the input only.

- [1] Take the first column of `Xtrain` and `Xtest` into arrays `X0train` and `X0test`, respectively.

[20]:
```
# <SOL>
X0train = Xtrain[:, [0]]
X0test = Xtest[:, [0]]

# Uncomment this to reduce the dataset size
#ntr = 55
#X0train = Xtrain[0:ntr, [0]]
#strain = strain[0:ntr]
# </SOL>
```

- [2] Visualize, in a single scatter plot, the target variable (in the vertical axes) versus the input variable, using the training data

[21]:
```
# <SOL>
# Plot training points
plt.scatter(X0train, strain, linewidths=0.1);
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);
# </SOL>
```



- [3] Since the data have been taken from a real scenario, we do not have any *true* mathematical model of the process that generated the data. Thus, we will explore different models trying to take the one that fits better the training data.

  Assume a polinomial model given by

$$\mathbf{z} = T(\mathbf{x}) = (1, x_0, x_0^2, \ldots, x_0^{g-1})^\top.$$

  Compute matrices `Ztrain` and `Ztest` that result from applying the polynomial transformation to the inputs in `X0train` and `X0test` for a model with degree `g_max = 50`. The `np.vander()` method may be useful for this.

  Note that, despite `X0train` and `X0test` where normalized, you will need to re-normalize the transformed variables.

  Note, also, that the first component of the transformed variables, which must be equal to 1, should not be normalized. To simplify the job, the code below defines a normalizer class that performs normalization to all components unless for the first one.

[22]:
```
# The following normalizer will be helpful: it normalizes all components of the
# input matrix, unless for the first one (the "all-one's" column) that
# should not be normalized
class Normalizer():
    """
    A data normalizer. Usage:
        nm = Normalizer()
        Z = nm.fit_transform(X)  # to estimate the normalization mean and␣
 ↪variance an normalize
                                  # all columns of X unles the first one
        Z2 = nm.transform(X)      # to normalize X without recomputing mean and␣
 ↪variance parameters
    """

    def fit_transform(self, Z):

        self.mean_z = np.mean(Z, axis=0)
        self.mean_z[0] = 0
        self.std_z = np.std(Z, axis=0)
        self.std_z[0] = 1
        Zout = (Z - self.mean_z) / self.std_z

        # sc = StandardScaler()
        # Ztrain = sc.fit_transform(Ztrain)
        return Zout

    def transform(self, Z):
        return (Z - self.mean_z) / self.std_z
        # Ztest = sc.transform(Ztest)

# Set the maximum degree of the polynomial model
g_max = 50

# Compute polynomial transformation for train and test data
# <SOL>
Ztrain = np.vander(X0train.flatten(), g_max + 1, increasing=True)
Ztest = np.vander(X0test.flatten(), g_max + 1, increasing=True)
# </SOL>

# Normalize training and test data
# <SOL>
nm = Normalizer()
Ztrain = nm.fit_transform(Ztrain)
Ztest = nm.transform(Ztest)
# </SOL>
```

- [4] Fit a polynomial model with degree g for g ranging from 0 to `g_max`. Store the weights of all models in a list of weight vectors, named `models`, such that `models[g]` returns the parameters estimated for the polynomial model with degree g.

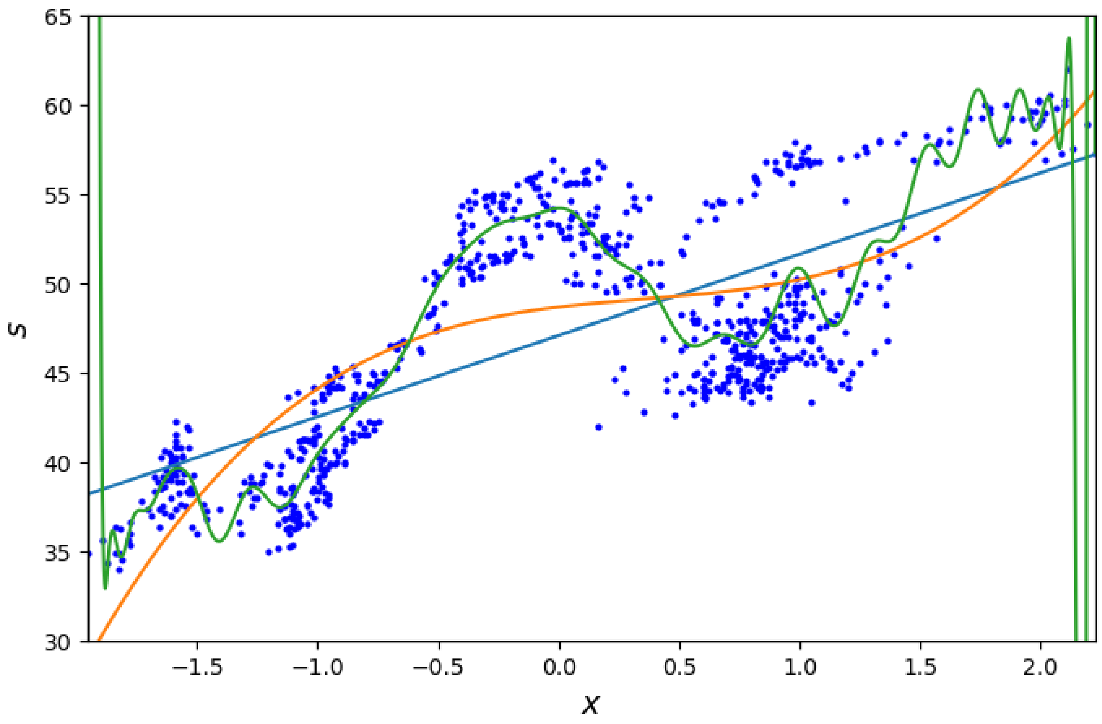
  We will use these models in the following sections.

[23]:
```
# IMPORTANT NOTE: Use np.linalg.lstsq() with option rcond=-1 for better␣
 ↪precission.
```

```
# HINT: Take into account that the data matrix required to fit a polynomial␣
 ↪model
#       with degree g consists of the first g+1 columns of Ztrain.

# <SOL>
models = []
for g in range(g_max + 1):
    w_MLg, _, _, _ = np.linalg.lstsq(Ztrain[:, 0:g+1], strain, rcond=-1)
    models.append(w_MLg)
# </SOL>
```

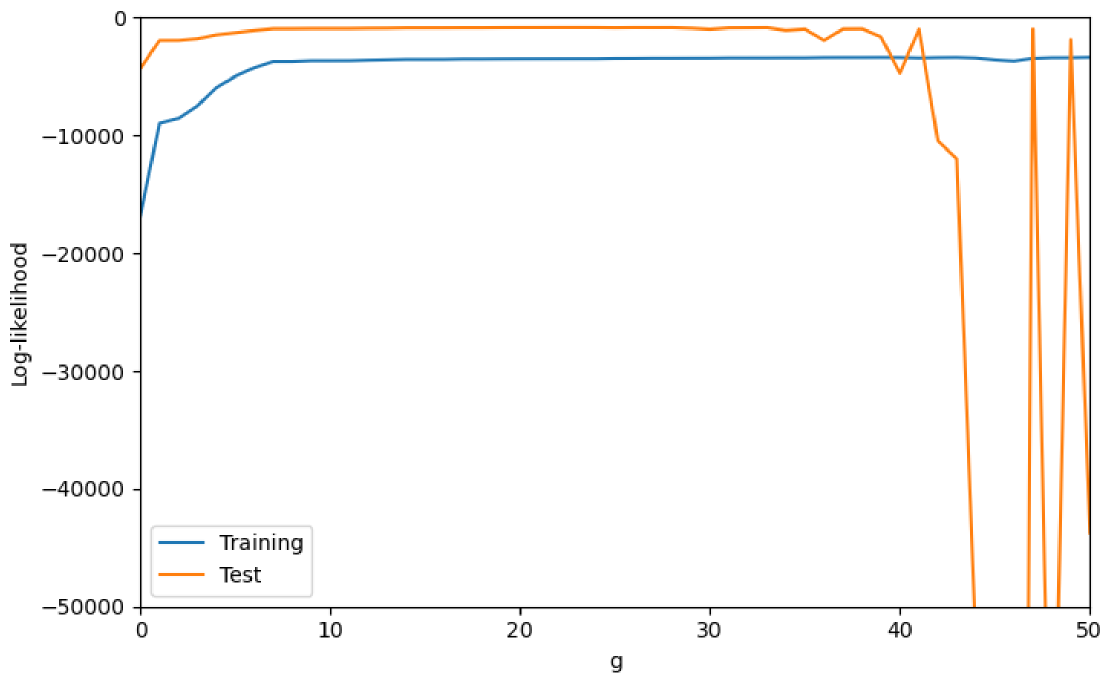
- [5] Plot the polynomial models with degrees 1, 3 and `g_max`, superimposed over a scatter plot of the training data.

[24]:
```
# Create a grid of samples along the x-axis.
n_points = 10000
xmin = min(X0train)
xmax = max(X0train)
X = np.linspace(xmin, xmax, n_points)

# Apply the polynomial transformation to the inputs with degree g_max.
# <SOL>
Z = np.vander(X.flatten(), g_max+1, increasing=True)
Z = nm.transform(Z)
# </SOL>

# Plot training points
plt.plot(X0train, strain, 'b.', markersize=4);
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);
plt.xlim(xmin, xmax)
plt.ylim(30, 65)

# Plot the regresion function for the required degrees
# <SOL>
for g in [1, 3, g_max]:
    s_ML = predict(Z[:,0:g+1], models[g])
    plt.plot(X, s_ML)
# </SOL>

plt.show()
```



- [6] Taking `sigma_eps = 1`, show, in the same plot:
  - The log-likelihood function corresponding to each model, as a function of g, computed over the training set.
  - The log-likelihood function corresponding to each model, as a function of g, computed over the test set.

[25]:
```
LLtrain = []
LLtest = []
sigma_eps = 1

# Fill LLtrain and LLtest with the log-likelihood values for all values of
# g ranging from 0 to g_max (included).
# <SOL>
for g in range(g_max + 1):
    LLtrain.append(LL(models[g], Ztrain[:, :g+1], strain, sigma_eps))
    LLtest.append(LL(models[g], Ztest[:, :g+1], stest, sigma_eps))
# </SOL>

plt.figure()
plt.plot(range(g_max + 1), LLtrain, label='Training')
plt.plot(range(g_max + 1), LLtest, label='Test')
plt.xlabel('g')
plt.ylabel('Log-likelihood')
plt.xlim(0, g_max)
plt.ylim(-5e4,100)
plt.legend()
plt.show()
```



- [7] You may have seen the likelihood function over the training data grows with the degree of the polynomial. However, large values of g produce a strong data overfitting. For this reasong, g cannot be selected with the same data used to fit the model.

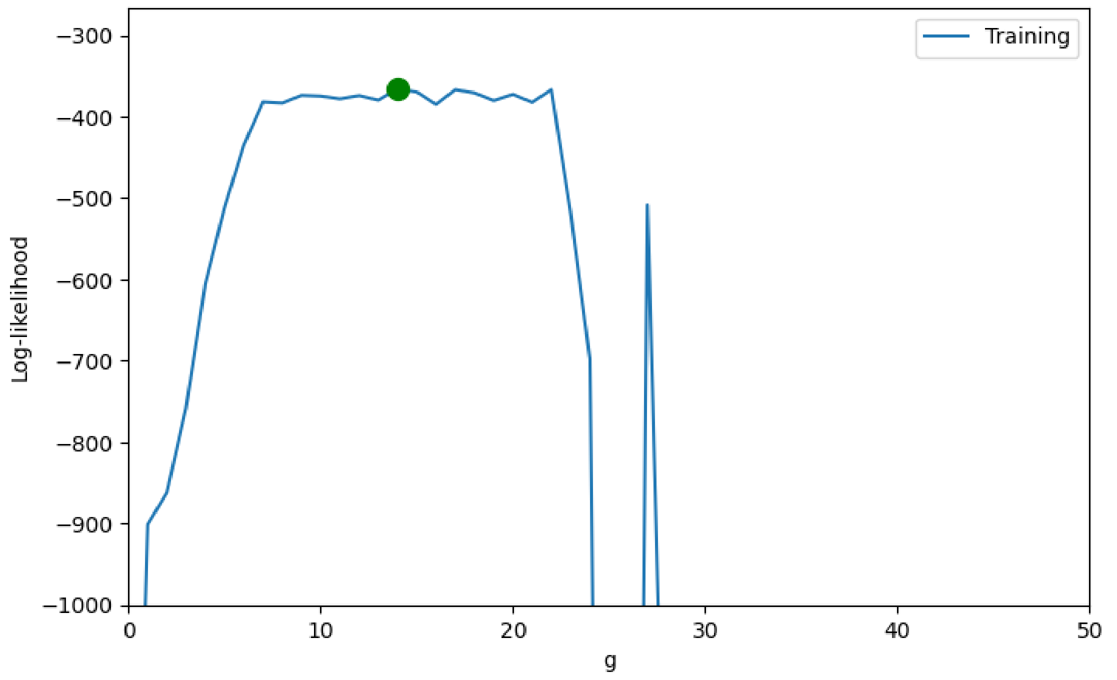
  This kind of parameters, like g are usually called *hyperparameters* and need to be selected by cross validation.

  Select the optimal value of g by 10-fold cross-validation. To do so, the cross validation methods provided by sklearn will simplify this task.

[26]:
```
from sklearn.model_selection import KFold

# Select the number of splits
n_sp = 10

# Create a cross-validator object
kf = KFold(n_splits=n_sp)
# Split data from Ztrain
kf.get_n_splits(Ztrain)

LLmean = []
```

```
for g in range(g_max + 1):
    # Compute the cross-validation Likelihood
    LLg = 0
    for tr_index, val_index in kf.split(Ztrain):
        # Take the data matrices for the current split
        Z_tr, Z_val = Ztrain[tr_index, 0:g+1], Ztrain[val_index, 0:g+1]
        s_tr, s_val = strain[tr_index], strain[val_index]

        # Train with the current training splits.
        # w_MLk, _, _, _ = np.linalg.lstsq(<FILL IN>)
        w_MLk, _, _, _ = np.linalg.lstsq(Z_tr[:, 0:g+1], s_tr, rcond=-1)

        # Compute the validation likelihood for this split
        # LLg += LL(<FILL IN>)
        LLg += LL(w_MLk, Z_val[:, :g+1], s_val, sigma_eps)

    LLmean.append(LLg / n_sp)

# Take the optimal value of g and its correpsponding likelihood
# g_opt = <FILL IN>
g_opt = np.argmax(LLmean)
# LLmax = <FILL IN>
LLmax = np.max(LLmean)

print(f"The optimal degree is: {g_opt}")
print(f"The maximum cross-validation likehood is {LLmax:.5f}")

plt.figure()
plt.plot(range(g_max + 1), LLmean, label='Training')
plt.plot([g_opt], [LLmax], 'g.', markersize = 20)
plt.xlabel('g')
plt.ylabel('Log-likelihood')
plt.xlim(0, g_max)
plt.ylim(-1e3, LLmax + 100)
plt.legend()
plt.show()
```

```
The optimal degree is: 14
The maximum cross-validation likehood is -365.90322
```



- [8] You may have observed the overfitting effect for large values of g. The best degree of the polynomial may depend on the size of the training set. Take a smaller dataset by running, after the code in section 4.2[1]:
- `X0train = Xtrain[0:55, [0]]`
- `X0test = Xtest[0:100, [0]]`

  Then, re-run the whole code after that. What is the optimal value of g in that case?

[27]:
```
# You do not need to code here. Just copy the value of g_opt obtained after␣
↪re-running the code
# g_opt_new = <FILL IN>
g_opt_new = 7

print(f"The optimal value of  g for the 100-sample training set is {g_opt_new}")
```

```
The optimal value of  g for the 100-sample training set is 7
```

- [9] [OPTIONAL]

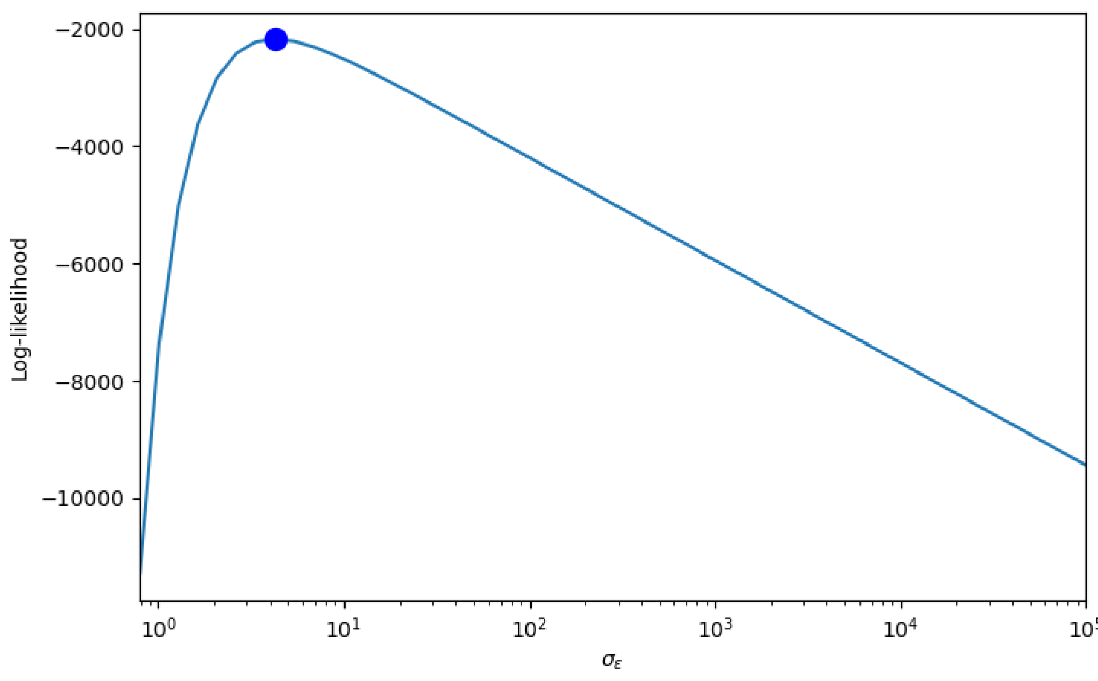
  Note that the model coefficients do not depend on σ_ϵ^2. Therefore, we do not need to care about its values for polynomial ML regression.

  However, the log-likelihood function do depends on σ_ϵ^2. Actually, we can estimate its value by cross-validation. By simple differentiation, it is not difficult to see that the optimal ML estimate of σ_ϵ is

$$\hat{\sigma}_\epsilon^2 = \sqrt{\frac{1}{K}\|\mathbf{s} - \mathbf{Z}\mathbf{w}\|^2}$$

Plot the log-likelihood function corresponding to the polynomial model with degree 3 for different values of σ_ϵ^2, for the training set, and verify that the value computed with the above formula is actually optimal.

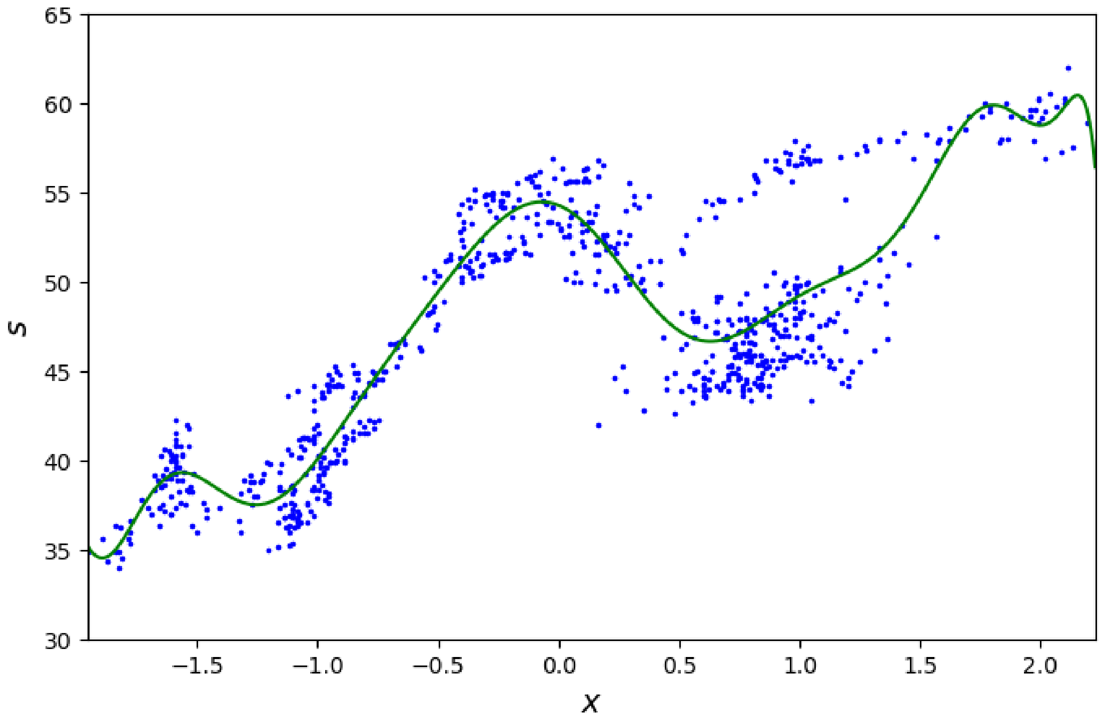
[28]:
```
# Explore the values of sigma logarithmically spaced according to the following␣
↪array
sigma_eps = np.logspace(-0.1, 5, num=50)

g = 3
K = len(strain)

# <SOL>
LL_eps = []
for sig in sigma_eps:
    # LL_eps.append(<FILL IN>)
    LL_eps.append(LL(models[g], Ztrain[:, :g+1], strain, sig))

sig_opt = np.sqrt(sse(Ztrain[:, :g+1], strain, models[g]) / K)
LL_opt = LL(models[g], Ztrain[:, :g+1], strain, sig_opt)

plt.figure()
plt.semilogx(sigma_eps, LL_eps)
plt.plot([sig_opt], [LL_opt], 'b.', markersize=20)
plt.xlabel('$\sigma_\epsilon$')
plt.ylabel('Log-likelihood')
plt.xlim(min(sigma_eps), max(sigma_eps))
plt.show()
# </SOL>
```



- [10] [OPTIONAL] For the selected model:
  - Plot the regresion function over the scater plot of the data.
  - Compute the log-likelihood and the SSE over the test set.

[29]:
```
# Note that you can easily adapt your code in 4.2[5]

# <SOL>

# Create a grid of samples along the x-axis.
n_points = 100000
xmin = min(X0train)
xmax = max(X0train)
X = np.linspace(xmin, xmax, n_points)

# Apply the polynomial transformation to the inputs with degree g_max.
Z = np.vander(X.flatten(), g_max+1, increasing=True)
Z = nm.transform(Z)

# Plot training points
plt.plot(X0train, strain, 'b.', markersize=3);
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);

# Plot the regresion function for the required degrees
s_ML = predict(Z[:,0:g_opt+1], models[g_opt])
```

```
plt.plot(X, s_ML, 'g-')

plt.xlim(xmin, xmax)
plt.ylim(30, 65)
plt.show()
# </SOL>
```



1.5 5. Bayesian regression. The stock dataset.

In this section we will keep using the first component of the data from the stock dataset, assuming the same kind of plolynomial model. We will explore the potential advantages of using a Bayesian model. To do so, we will asume that the a priori distribution of $\mathbf{w}$ is

$$\mathbf{w} \sim N(\mathbf{0}, \sigma_p^2 \, \mathbf{I})$$

1.5.1 5.1. Hyperparameter selection

Since the values σ_p and σ_ε are no longer known, a first rough estimation is needed (we will soon see how to estimate these values in a principled way).

To this end, we will adjust them using the ML solution to the regression problem with g=10:

- σ_p^2 will be taken as the average of the square values of $\hat{\mathbf{w}}_{ML}$
- σ_ε^2 will be taken as two times the average of the square of the residuals when using $\hat{\mathbf{w}}_{ML}$

[30]:
```
# Degree for bayesian regression
gb = 10

# w_LS, residuals, rank, s = <FILL IN>
w_ML, residuals, rank, s = np.linalg.lstsq(Ztrain[:, :gb+1], strain, rcond=-1)

# sigma_p = <FILL IN>
sigma_p = np.sqrt(np.mean(w_ML**2))
# sigma_eps = <FILL IN>
sigma_eps = np.sqrt(2*np.mean((strain - Ztrain[:, :gb+1] @ w_ML)**2))

print(f"{sigma_p:.5f}")
print(f"{sigma_eps:.5f}")
```

```
57.21889
3.93863
```

1.5.2 5.2. Posterior pdf of the weight vector

In this section we will visualize prior and the posterior distribution functions. First, we will restore the dataset at the begining of this notebook:

- [1] Define a function `posterior_stats(Z, s, sigma_eps, sigma_p)` that computes the parameters of the posterior coefficient distribution given the dataset in matrix `Z` and vector `s`, for given values of the hyperparameters. This function should return the posterior mean, the covariance matrix and the precision matrix (the inverse of the covariance matrix). Test the function to the given dataset, for $g = 3$.

[31]:
```
# <SOL>
def posterior_stats(Z, s, sigma_eps, sigma_p):

    dim_w = Z.shape[1]
    iCov_w = (Z.T @ Z)/(sigma_eps**2) + np.eye(dim_w, dim_w)/(sigma_p**2)
    Cov_w = np.linalg.inv(iCov_w)
    mean_w = (Cov_w @ Z.T @ s)/(sigma_eps**2)

    return mean_w, Cov_w, iCov_w
# </SOL>

mean_w, Cov_w, iCov_w = posterior_stats(Ztrain[:, :gb+1], strain, sigma_eps,␣
 ↪sigma_p)

print(f'mean_w = {mean_w}')
# print('Cov_w = {Cov_w}')
# print('iCov_w = {iCov_w}')
```

```
mean_w = [[ 47.06072634]
 [ -5.43972905]
 [-19.6947545 ]
 [ 40.06018631]
 [ 49.95913747]
 [-75.35809116]
 [-44.86743888]
 [ 67.27934244]
 [ 10.79541196]
 [-21.80928632]
 [  1.5718318 ]]
```

- [2] Define a function `gauss_pdf(w, mean_w, iCov_w)` that computes the Gaussian pdf with mean `mean_w` and precision matrix `iCov_w`. Use this function to compute and compare the ML estimate and the MSE estimate, given the dataset.

[32]:
```
# <SOL>
def gauss_pdf(w, mean_w, iCov_w):

    d = w - mean_w
    w_dim = len(mean_w)
    pw = np.sqrt(np.linalg.det(iCov_w)) / (2*np.pi)**(w_dim/2) * np.exp(- d.T.
 ↪dot(iCov_w.dot(d))/2)

    return pw[0][0]
# </SOL>

print(f'p(w_ML | s)  = {gauss_pdf(w_ML, mean_w, iCov_w):.5E}')
print(f'p(w_MSE | s)  = {gauss_pdf(mean_w, mean_w, iCov_w):.5E}')
```

```
p(w_ML | s)  = 4.80554E-06
p(w_MSE | s)  = 2.17834E-05
```

- [3] [OPTIONAL] Define a function `log_gauss_pdf(w, mean_w, iCov_w)` that computes the log of the Gaussian pdf with mean `mean_w` and precision matrix `iCov_w`. Use this function to compute and compare the log of the posterior pdf value of the true coefficients, the ML estimate and the MSE estimate, given the dataset.

[33]:
```
# <SOL>
def log_gauss_pdf(w, mean_w, iCov_w):

    d = w - mean_w
    w_dim = len(mean_w)
    pw = 0.5 * np.log(np.linalg.det(iCov_w)) - 0.5 * w_dim * np.log(2*np.pi) -␣
 ↪0.5 * d.T.dot(iCov_w.dot(d))

    return pw[0][0]
# </SOL>

print(f'log(p(w_ML | s))  = {log_gauss_pdf(w_ML, mean_w, iCov_w):.5f}')
print(f'log(p(w_MSE | s))  = {log_gauss_pdf(mean_w, mean_w, iCov_w):.5f}')
```

```
log(p(w_ML | s))  = -12.24574
log(p(w_MSE | s))  = -10.73436
```

1.5.3 5.3 Sampling regression curves from the posterior

In this section we will plot the functions corresponding to different samples drawn from the posterior distribution of the weight vector.

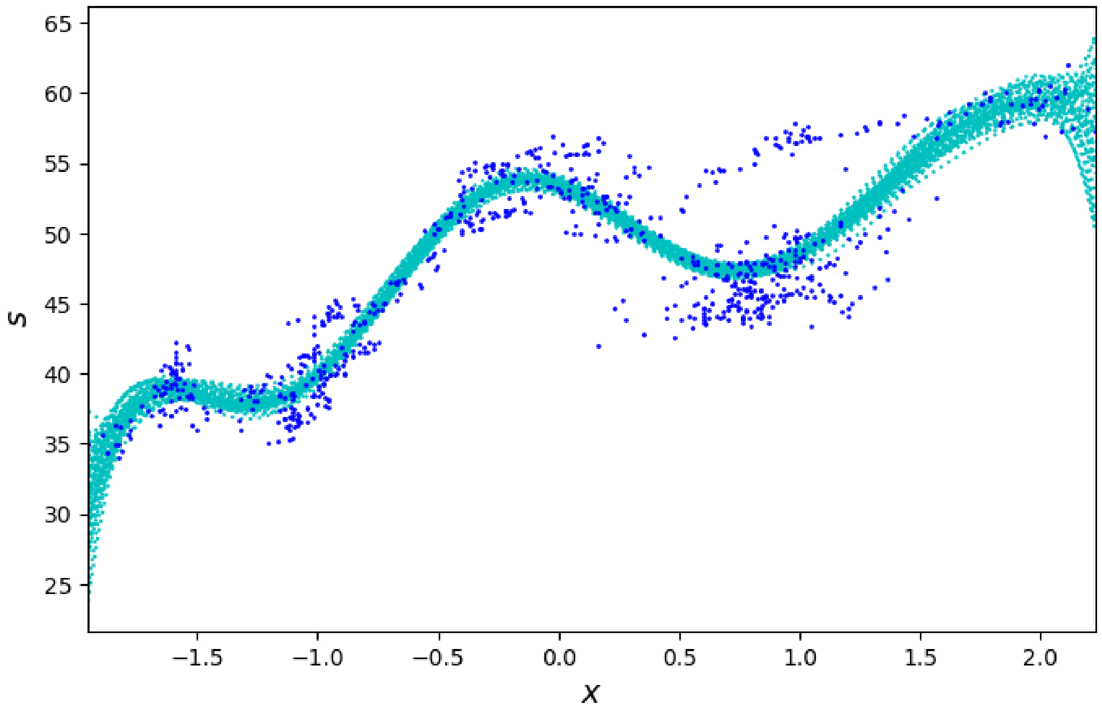
To this end, we will first generate an input dataset of equally spaced samples. We will compute the functions at these points

[34]:
```python
# Definition of the interval for representation purposes
xmin = min(X0train)
xmax = max(X0train)
n_points = 100   # Only two points are needed to plot a straigh line

# Build the input data matrix:
# Input values for representation of the regression curves
X = np.linspace(xmin, xmax, n_points)
Z = np.vander(X.flatten(), g_max+1, increasing=True)
Z = nm.transform(Z)[:, :gb+1]
```

Generate random vectors $\mathbf{w}_l$ with $l = 1, \ldots, 50$, from the posterior density of the weights, $p(\mathbf{w} \mid \mathbf{s})$, and use them to generate 50 polinomial regression functions, $f(\mathbf{x}^*) = {\mathbf{z}^*}^\top \mathbf{w}_l$, with $\mathbf{x}^*$ between -1.2 and 1.2, with step 0.1.

Plot the line corresponding to the model with the posterior mean parameters, along with the 50 generated straight lines and the original samples, all in the same plot. As you can check, the Bayesian model is not providing a single answer, but instead a density over them, from which we have extracted 50 options.

[35]:
```python
# Drawing weights from the posterior
for l in range(50):
    # Generate a random sample from the posterior distribution (you can use np.
 ↪random.multivariate_normal())
    # w_l = <FILL IN>
    w_l = np.random.multivariate_normal(mean_w.flatten(), Cov_w)

    # Compute predictions for the inputs in the data matrix
    # p_l = <FILL IN>
    p_l = Z.dot(w_l)

    # Plot prediction function
    # plt.plot(<FILL IN>, 'c:');
    plt.plot(X, p_l, 'c:');

# Plot the training points
plt.plot(X0train, strain,'b.',markersize=2);
plt.xlim((xmin, xmax));
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);
```



1.5.4 5.4. Plotting the confidence intervals

On top of the previous figure (copy here your code from the previous section), plot functions

$$\mathbb{E}\left\{f(\mathbf{x}^*) \mid \mathbf{s}\right\}$$

and

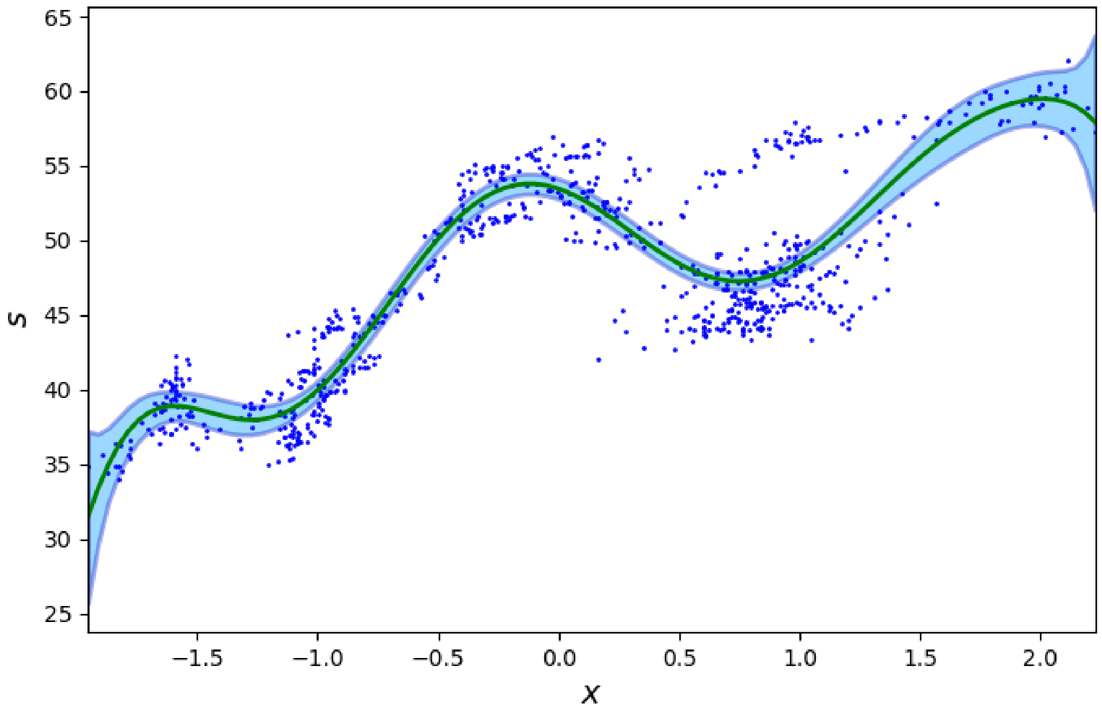
$$\mathbb{E}\left\{f(\mathbf{x}^*) \mid \mathbf{s}\right\} \pm 2\sqrt{\mathbb{V}\left\{f(\mathbf{x}^*) \mid \mathbf{s}\right\}}$$

(i.e., the posterior mean of $f(\mathbf{x}^*)$, as well as two standard deviations above and below).

It is possible to show analytically that this region comprises 95.45% probability of the posterior probability $p(f(\mathbf{x}^*) \mid \mathbf{s})$ at each $\mathbf{x}^*$.

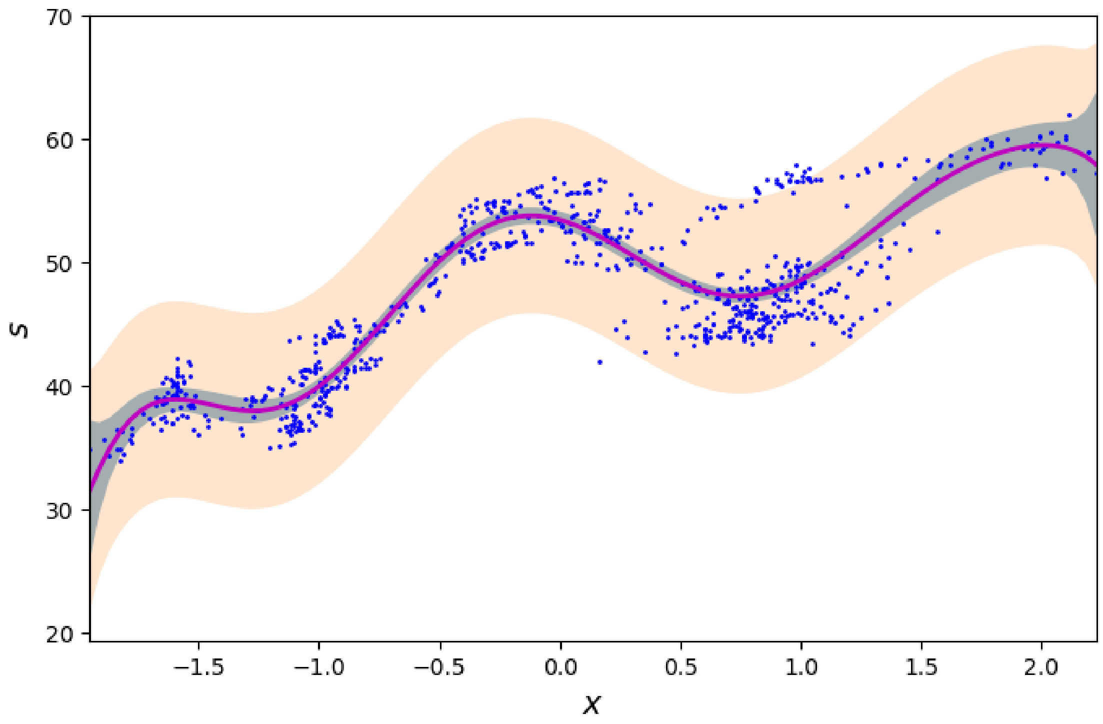
[36]:
```
# Note that you can re-use code from sect. 4.2 to solve this exercise

# Plot the training points
# plt.plot(X, Z.dot(true_w), 'b', label='True model', linewidth=2);
plt.plot(X0train, strain,'b.',markersize=2);
plt.xlim(xmin, xmax);
```

```
# </SOL>

# Plot the posterior mean.
# mean_s = <FILL IN>
mean_s = Z @ mean_w
plt.plot(X, mean_s, 'g', label='Predictive mean', linewidth=2);

# Plot the posterior mean +- two standard deviations
# std_f = <FILL IN>
std_f = np.sqrt(np.diagonal(Z @ Cov_w @ Z.T))[:, np.newaxis]

# Plot the confidence intervals.
# To do so, you can use the fill_between method
plt.fill_between(X.flatten(), (mean_s - 2*std_f).flatten(), (mean_s + 2*std_f).
 ↪flatten(),
                 alpha=0.4, edgecolor='#1B2ACC', facecolor='#089FFF',␣
 ↪linewidth=2)

# plt.legend(loc='best')
plt.xlabel('$x$',fontsize=14);
plt.ylabel('$s$',fontsize=14);
plt.show()
```



Plot now $\mathbb{E}\left\{s(\mathbf{x}^*) \mid \mathbf{s}\right\} \pm 2\sqrt{\mathbb{V}\left\{s(\mathbf{x}^*) \mid \mathbf{s}\right\}}$ (note that the posterior means of $f(\mathbf{x}^*)$ and $s(\mathbf{x}^*)$ are the same, so there is no need to plot it again). Notice that 95.45% of observed data lie now within the newly designated region. These new limits establish a confidence range for our predictions. See how the uncertainty grows as we move away from the interpolation region to the extrapolation areas.

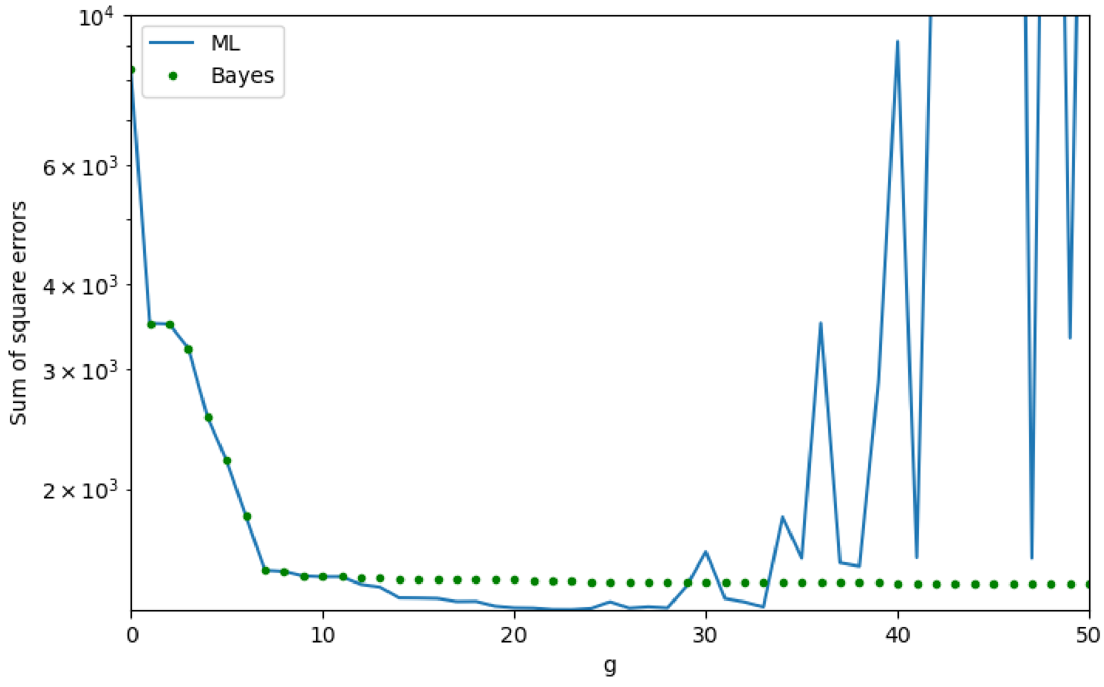
```
[37]: # Plot sample functions confidence intervals and sampling points
      # Note that you can simply copy and paste most of the code used in the cell␣
        ↪above.
      # <SOL>

      # Plot the training points
      plt.figure()
      plt.plot(X0train, strain,'b.',markersize=2);
      plt.xlim(xmin, xmax);
      plt.xlabel('$x$',fontsize=14);
      plt.ylabel('$s$',fontsize=14);

      # Plot the posterior mean.
      plt.plot(X, mean_s, 'm', label='Predictive mean', linewidth=2);

      # Plot the posterior mean +- two standard deviations (as in the previous cell)
      # plt.fill_between(# <FILL IN>)
      plt.fill_between(X.flatten(), (mean_s - 2*std_f).flatten(), (mean_s + 2*std_f).
        ↪flatten(), alpha=0.5)

      # Compute the standard deviations for s and plot the confidence intervals
      # <SOL>
      std_s = np.sqrt(np.diagonal(Z @ Cov_w @ Z.T) + sigma_eps**2)[:, np.newaxis]

      # Plot now the posterior mean and posterior mean \pm 2 std for s (i.e., adding␣
        ↪the noise variance)
      # plt.fill_between(# <FILL IN>)
      plt.fill_between(X.flatten(), (mean_s- 2*std_s).flatten(), (mean_s + 2*std_s).
        ↪flatten(), alpha=0.2)
      # </SOL>

      plt.show()
```



1.5.5 5.5. Test square error

- [1] To test the regularization effect of the Bayesian prior. To do so, compute and plot the sum of square errors of both the ML and Bayesian estimates as a function of the polynomial degree.

[38]:
```python
SSE_ML = []
SSE_Bayes = []

# Compute the SSE for the ML and the bayes estimates
for g in range(g_max + 1):
    # <SOL>
    SSE_ML.append(sse(Ztest[:, :g+1], stest, models[g]))

    mean_w, Cov_w, iCov_w = posterior_stats(Ztrain[:, :g+1], strain, sigma_eps,␣
 ↪sigma_p)
    SSE_Bayes.append(sse(Ztest[:, :g+1], stest, mean_w))
    # </SOL>

plt.figure()
plt.semilogy(range(g_max + 1), SSE_ML, label='ML')
plt.semilogy(range(g_max + 1), SSE_Bayes, 'g.', label='Bayes')
plt.xlabel('g')
plt.ylabel('Sum of square errors')
plt.xlim(0, g_max)
plt.ylim(min(min(SSE_Bayes), min(SSE_ML)),10000)
plt.legend()
plt.show()
```



1.5.6 5.6. [Optional] Model assessment

In order to verify the performance of the resulting model, compute the posterior mean and variance of each of the test outputs from the posterior over $\mathbf{w}$. I.e, compute $\mathbb{E}\{s(\mathbf{x}^*) \mid \mathbf{s}\}$ and $\sqrt{\mathbb{V}\{s(\mathbf{x}^*) \mid \mathbf{s}\}}$ for each test sample $\mathbf{x}^*$ contained in each row of `Xtest`.

Store the predictive mean and variance of all test samples in two column vectors called `m_s` and `v_s`, respectively.

[39]:
```
# <SOL>
m_s = Ztest[:, :g+1].dot(mean_w)
# v_s = np.diagonal(Ztest[:, :g+1].dot(Cov_w).dot(Ztest[:, :g+1].T)) +␣
 ↪sigma_eps**2
v_s = np.diagonal(Ztest[:, :g+1] @ Cov_w @ Ztest[:, :g+1].T) + sigma_eps**2
v_s = np.matrix(v_s).T
# </SOL>
```

Compute now the mean square error (MSE) and the negative log-predictive density (NLPD) with the following code:

[40]:
```python
# <SOL>
MSE = np.mean((m_s - stest)**2)
NLPD = 0.5 * np.mean(((stest - m_s)**2)/v_s) + 0.5*np.log(2*np.pi*np.prod(v_s))
# </SOL>

print(f'MSE = {MSE:.5f}')
print(f'NLPD = {NLPD:.5f}')
```

```
MSE = 7.64624
NLPD = 263.86713
```

These two measures reveal the quality of our predictor (with lower values revealing higher quality). The first measure (MSE) only compares the predictive mean with the actual value and always has a positive value (if zero was reached, it would mean a perfect prediction). It does not take into account predictive variance. The second measure (NLPD) takes into account both the deviation and the predictive variance (uncertainty) to measure the quality of the probabilistic prediction (a high error in a prediction that was already known to have high variance has a smaller penalty, but also, announcing a high variance when the prediction error is small won't award such a good score).

1.6 6. [Optional] Regression with all variables from the stocks dataset.

Try to improve the test SSE of the best model used so far. To do so:

- Explore the use of all the input variables from the dataset.
- Explore other regression algorithms from the `sklearn` library.

[ ]:
```python

```