Managing Uncertainty

Bayesian Linear Regression and Kalman Filter

November 14, 2016

Objectives

The goal of this lab is multiple:

1. First it is a reminder of some central elementary notions of Bayesian Machine Learning in the specific context of linear regression: Bayes inference, MLE and MAP estimators, conjugate prior, prior as a regularization factor, etc.
2. Second it shows how the classical Ordinary Least Square (OLS) and the ridge regression (with L2 regularization) can be interpreted as special cases of a more general Bayesian linear regression. In other words, it shows how the frequentist approach of Machine Learning and the Bayesian one can intersect.
3. Third it introduces Recursive Least Square, an original application of Kalman filter to fit parameters of a linear model in an online manner.

Notations

In the following, a matrix will be denoted $\boldsymbol{X}$, a column vector $\boldsymbol{x}$ and a scalar x. $\boldsymbol{I}$ is the identity matrix and $\boldsymbol{X}^T$ is the transpose of matrix $\boldsymbol{X}$. Given vector $\boldsymbol{c}$, $\boldsymbol{c}^{(j)}$ denotes the j$^{\text{th}}$ component of vector $\boldsymbol{c}$.

1 Ordinary Least Square

Given a dataset $\mathcal{D}$ of samples $(\boldsymbol{x}_i, y_i)_{i=1,\dots,n}$ (with $\boldsymbol{x}_i \in \mathbb{R}^m$ and $y_i \in \mathbb{R}$), *regression* consists in learning a model to predict some value for y given some new $\boldsymbol{x}$. Values of $\boldsymbol{x}$ are called *inputs* and values of y *outputs*. A *parametric approach* to solve regression problems consists in learning from the samples the parameters $\boldsymbol{\theta}$ of some parametrized predictive function $f_{\boldsymbol{\theta}}$, so that $y \approx f_\theta(\boldsymbol{x})$. A linear model is such that the output is assumed to be a linear combination of the input variables:

$$f_\theta(\boldsymbol{x}) = \sum_{j=1}^{m} \theta^{(j)}\, x^{(j)} = \boldsymbol{x}^T \boldsymbol{\theta} \tag{1}$$

Ordinary Least Square (OLS) consists in finding the parameters $\boldsymbol{\theta}^*_{OLS}$ minimizing the empirical risk $J_2(\boldsymbol{\theta})$ for a quadratic loss:

$$\boldsymbol{\theta}^*_{OLS} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J_2(\boldsymbol{\theta}) \text{ with } J_2(\boldsymbol{\theta}) = \frac{1}{n}\left(\sum_{i=1}^{n} \left(y_i - \boldsymbol{x}_i^T \boldsymbol{\theta}\right)^2\right) \tag{2}$$

This can be rewritten in a matrix form by introducing $\boldsymbol{y}$ as the n-vector containing outputs y_i and $\boldsymbol{X}$ as the $n \times m$-matrix whose i$^{\text{th}}$ line is the input $\boldsymbol{x_i}$:

$$\boldsymbol{\theta}^*_{OLS} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J_2(\boldsymbol{\theta}) \text{ with } J_2(\boldsymbol{\theta}) = \frac{1}{n}\|\boldsymbol{y} - \boldsymbol{X}\,\boldsymbol{\theta}\|^2 \tag{3}$$

Question 1.1 Compute the gradient of $J(\boldsymbol{\theta})$. Deduce that:

$$\boldsymbol{\theta}^*_{OLS} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{4}$$

$(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ is called the *pseudoinverse matrix* of $\boldsymbol{X}$.

Question 1.2 Implement the OLS estimator based on the following recommendations.

One will use *Python3* and the matrix operators of *Numpy*. Some useful functions are already provided in the script `linear.py`. Some useful numpy functions are recalled in the appendix section. To run this script, you need to pass arguments from the command line to tell which test function you want to call, with what values for the main arguments. For instance you must type in a terminal:

```
python3 linear.py --n 50 --sig 0.1 --test 1
```

This command will call `testSimpleOLS(50,0.1)`, i.e the test function number 1 that computes OLS using 50 samples with an error standard deviation of $\sigma_e = 0.1$.

In order to learn and evaluate linear regressors, one will generate n samples from a known model such that:

$$y = \theta_0\, x^{(1)} + \theta_1\, x^{(2)} + \theta_2 + \epsilon = \boldsymbol{x}^T\boldsymbol{\theta} \text{ with } \boldsymbol{x}^T = [x^{(1)}, x^{(2)}, 1] \tag{5}$$

Parameters $\boldsymbol{\theta}$ will be fixed to some arbitrary values, for instance $\boldsymbol{\theta}^T = [1, 2, 3]$. White noise ϵ is sampled from distribution $\mathcal{N}(0, \sigma_\epsilon^2)$. To do so, the input matrix $[x_i^{(1)}, x_i^{(2)}]$ of size $(n, 2)$ will be drawn from the unit square, thanks to the provided function `X = drawInputsInSquare(n)`. The function `addConstantInput` will add an extra column of ones. The output vector $\boldsymbol{y}$ will be drawn with the function `y = drawOutput(X, theta, sigmae)` where `theta` is the array $[\theta_0, \theta_1, \theta_2]$. In order to compare the learnt model with the real one, one will use the function `compareModels(realTheta, estTheta, X, Y)` where `realTheta` is the real model used to draw the data, `estTheta` is the estimated model $\boldsymbol{\theta}^*_{OLS}$, `X` and `Y` are the inputs and outputs of the data used to learn the model.

Question 1.3 Compare the empirical risk J_e obtained by OLS on the training dataset with an estimation of the real risk J_r. Represent graphically both risks as functions of σ_ϵ and n using the following recommandations.

To estimate the real risk, one will need to compute the empirical risk (i.e. the average quadratic loss) on a test set of, let's say, 1000 samples, drawn indepdently of the training dataset. For each pair of values for σ_ϵ and n, one will store in two distinct matrices the risks computed on the training and test samples so that these results comply with the function `drawTrainAndTestEmpiricalRisk`, used to display both matrices as colormaps. One can use the function `quadraticRisk` to compute empirical risks. Finally the estimation of risks for every pair (σ, n) can be smoothed by averaging multiple estimations.

Question 1.4 OLS is undefined when $\boldsymbol{X}^T\boldsymbol{X}$ is singular. When does it happen?

Question 1.5 Test the OLS estimator in this case using the following recommendations.

One will use the function `drawInputsAlmostAligned` instead of `drawInputsInSquare`. To choose a correct value for the alignment factor, one can observe graphically the alignement using `drawModel`.

2 Regularized Linear Regression

2.1 Maximum Likelihood Estimation and OLS

The OLS can be interpreted as a Bayesian estimation problem. To this end, a discriminative model must be defined, i.e. the distribution $P(Y|\boldsymbol{X})$ of Y given $\boldsymbol{X} = [X_1, \ldots, X_m]^T$.

Question 2.6 Choose the most obvious model for $P(Y|\boldsymbol{X})$ and precise its parameters $\boldsymbol{\theta}$.

One recalls that the MLE estimator chooses the parameter values that maximize the likelihood:

$$\boldsymbol{\theta}^*_{MLE} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left(P(\mathcal{D}|\boldsymbol{\theta})\right) \tag{6}$$

Let first assume the variance of the additive white noise is known.

Question 2.7 Give the expression of the loglikelihood. Deduce that $\boldsymbol{\theta}^*_{MLE} = \boldsymbol{\theta}^*_{OLS}$

2.2 MAP estimator and Ridge regression

MLE assumes a uniform prior on $\boldsymbol{\theta}$. We now relaxes this assumption and look for the MAP estimator $\boldsymbol{\theta}^*_{MAP}$ that maximizes the posterior distribution $P(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y})$:

$$\boldsymbol{\theta}^*_{MAP} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left(P(\boldsymbol{\theta}|\mathcal{D})\right) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \left(P(\mathcal{D}|\boldsymbol{\theta})\, P(\boldsymbol{\theta})\right) \tag{7}$$

One assumes the prior is a multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu_0}, \boldsymbol{\Sigma_0})$:

Question 2.8 Show that $\boldsymbol{\theta}^*_{MAP}$ minimizes a regularized quadratic risk $J_r(\boldsymbol{\theta})$. Give its expression.

Question 2.9 Find the expression of the gradient of $J_r(\boldsymbol{\theta})$ and give the expression of $\boldsymbol{\theta}^*_{MAP}$.

The standard ridge regression consists in an L2 regularized version of OLS:

$$\boldsymbol{\theta}^*_{Ridge} = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left(J_{Ridge}(\boldsymbol{\theta})\right) \text{ with } J_{Ridge}(\boldsymbol{\theta}) = J_2(\boldsymbol{\theta}) + \lambda \left\|\boldsymbol{\theta}\right\|^2 \tag{8}$$

Question 2.10 Show that Ridge regression can be interpreted as a special case of a MAP estimator for some values of $\boldsymbol{\mu_0}$ and $\boldsymbol{\Sigma_0}$. Interpret the expression of λ as a balance between confidence in prior and observations.

Question 2.11 Implement the Ridge estimator (or equivalently the MAP estimator). Compare its level of performance with the one of OLS estimator, especially on almost aligned inputs.

3 Strong Bayesian Approach and Recursive Least Square

We would like to regress the model online, i.e. on the fly, every time a new sample is received. This is possible with a "strong" Bayesian approach where the parameters are described by some distribution $P(\boldsymbol{\theta})$ that is updated every time a new observation o is received, i.e by replacing the prior $P(\boldsymbol{\theta})$ by the posterior distribution $P(\boldsymbol{\theta}|o)$.

Question 3.12 Give a possible conjugate prior $P(\boldsymbol{\theta})$ for the given likelihood function.

3.1 Recursive Least Square

It is possible to derive the posterior from a normal prior and a normal likelihood. However one can shortcut this calculation by reusing the resulting equations of Kalman filters. Indeed the output y can be interpreted as an observation that is linear with the parameters $\boldsymbol{\theta}$ to estimate.

Question 3.13 Specify the Kalman filter that estimates $\boldsymbol{\theta}$ from a stream of data (i.e. a temporal sequence of couple $(\boldsymbol{x}_t, y_t)$ of inputs/output). Is it a stationnary filter?

This online method is called *Recursive Least Square*.

Question 3.14 Implement it in Python by using the following recommandations.

One will complete the constructor `__init__` and the method `update` of the provided class `Filter`. At each iteration, one will update the filter with a new sample $(\boldsymbol{x}, y)$ and one will collect the current state of the filter and the diagonal coefficients of the covariance matrix $\boldsymbol{P}$ to store them in two distinct matrices whose number of rows correspond to the size of the state and whose number of columns is the number n of iterations. These matrices will then be passed as arguments to the function `plotFilterState` in order to analyse the state convergence.

Question 3.15 Observe how the initial covariance on the filter state has some influence on the convergence. Compare the final estimate of the filter with the OLS estimation computed on the set of samples used to update the filter. Explain.

One will draw for each parameter θ_0, θ_1 and θ_2 the expected value $\hat{\theta}_x$ within a confidence interval of $[\hat{\theta}_x - 2\,\sigma_{\theta_x}, \hat{\theta}_x + 2\,\sigma_{\theta_x}]$.

3.2 Kalman Filter

In the previous Kalman filter, the parameters $\boldsymbol{\theta}$ are supposed to be constant. Let's now assume every parameter has a slow random drift with time. This drift is modelled as a random walk whose standard deviation per unit of time is estimated to be equal to $\sigma_q = 0.1/s$. Suppose one wants to estimate the parameters of our model every second and that observations come with some frequency f_{obs}.

Question 3.16 Modify your filter to take into account the drift

Question 3.17 Observe the behaviour of your filter when the observation frequency is getting low (for instance one observation every 10 or 100 seconds).

3.3 Appendix

Here are some Python commands that can be useful:

```
import numpy as np
A = np.arange(0.1, 1, 0.2) # Create a numpy array [0.1, 0.3, ..., 0.9]
A /= 2 # Divide coefficients of A by 2
B = np.zeros((2,4)) # Create a null matrix of size (2,4)
print(B.T) # Print transpose of B
C = np.eye(4) # Create a 2D array equal to the identity matrix
  # of size 4 x 4
B * C # => Does not work with the array multiplication
    # component by component
D = np.asmatrix(C) # Interpret C as a matrix
B * D # Works as one uses the matrix multiplication

E = np.array([1,2,3,4]) # Create a 1D array
```

```
F = np.matrix(E) # Create a line matrix
G = np.matrix(F) # Create a column matrix
H = np.reshape(np.asarray(G),4) # Create a 1D array
   # from a matrix

print(C.I) # Print the inverse matrix of D
print(B.shape) # B.shape gives the size of B as a pair (5,2)
for k in range(10): print(k) # Loops for k from 0 to 9

for v in A: print(v) # Iterate over the elements of A
for i,v in enumerate(A): print(str(i) + " -> " + str(v))
  # Same thing but having with the element index i as well
```