Homework

Stochastic Models and Adaptive Algorithms

Réda Vince

Z697LX

January 6, 2021

Contents

1 Linear regression

1.1 Function approximation with least squares

1.1.1 Ordinary least-squares

The best linear approximation of a function can be calculated using least-squares.

At first, a noisy sample of (x, y) pairs is generated such that $y_i = c\,x_i\,\sin(c\,x_i) + \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, 1)$.

Let $[\mathbf{\Phi}]_{ij} = \mathrm{f}_j(x_i)$ be the transformed input vector and $\mathbf{y}$ the output, where f_j is a basis function. From this the $\mathbf{\Phi}(x)$ matrix can be generated after selecting a suitable f. A number of these were tried, and the best one for the problem seemed to be the polynomial one, that is $\mathrm{f}_i(x) = x^{i-1}$. As a parameter, $d = 10$ was used.

Now we have to find the optimal $\hat{\boldsymbol{\theta}}$ parameter vector, for which $\mathbf{\Phi}\,\boldsymbol{\theta} = \mathbf{y} = \begin{bmatrix} y_1 \dots y_n \end{bmatrix}^{\mathrm{T}}$. This is done like so: $\boldsymbol{\theta}^* \approx \hat{\boldsymbol{\theta}} = \mathbf{\Phi}^{+}\mathbf{y}$. The $\mathbf{\Phi}$ matrix of the sampled inputs and of the LS estimate are the same, so the function is evaluated at the same x values.

A computationally cheaper method for the pseudoinverse is QR decomposition. For this, the "economic" mode of scipy's `qr` function is used. Then the pseudoinverse is $\mathbf{\Phi}^{+} = \mathbf{R}^{-1}\mathbf{Q}^{\mathrm{T}}$. Because the pseudoinverse of a matrix is unique, this method gives the same result as the previous one.

The results are shown in figure 1a.

1.1.2 Recursive least-squares

Next, more of the above described (x, y) pairs is sampled and $\hat{\boldsymbol{\theta}}_n$ calculated periodically using recursive least-squares. In the code, all of the samples are measured beforehand for simplicity.

The equation for $\hat{\boldsymbol{\theta}}$ can be reformulated in the following way:

$$\hat{\boldsymbol{\theta}} = \underbrace{\left[\sum_{i=1}^{n} \boldsymbol{\varphi}\,\boldsymbol{\varphi}^{\mathrm{T}}\right]^{-1}}_{\mathbf{\Psi}_n} \underbrace{\sum_{i=1}^{n} y_i\,\boldsymbol{\varphi}_i}_{z_n}. \tag{1}$$

Now we have an update rule for both $\mathbf{\Psi}_{n+1} = \left(\mathbf{\Psi} + \boldsymbol{\varphi}_{n+1}{\boldsymbol{\varphi}_{n+1}}^{\mathrm{T}}\right)^{-1}$, and $z_{n+1} = z_n + \boldsymbol{\varphi}_{n+1}\mathbf{y}_{n+1}$. Both $\mathbf{\Psi}_0$ and z_0 are set to zero. Also, in the code $\hat{\boldsymbol{\theta}}_n$ is calculated only when plotted for speed.

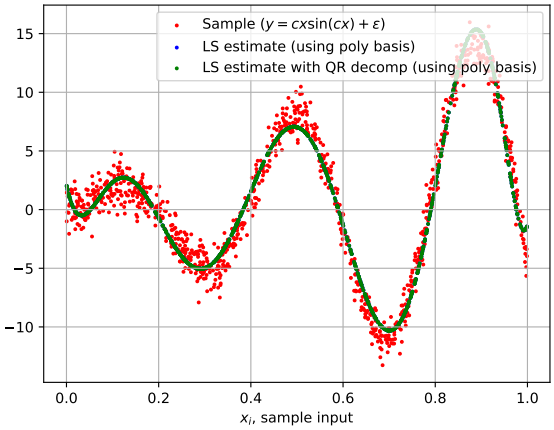
The resulting plots are shown in figure 1c., taken at $n \in [25, 50, 75, 100]$.

1.1.3 Least-norm problem

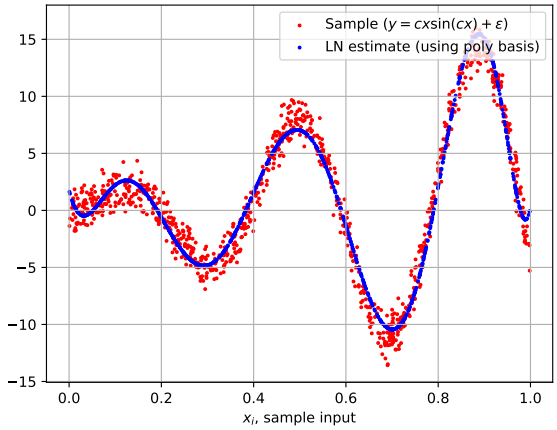
Now let's make $d > n$, specifically $n = 100$ and $d = 200$, which makes $\mathbf{\Phi}$ fat. We make the assumption that $\mathbf{\Phi}$ is still full-rank. The solution is the same, except we are going to use singular value decomposition (SVD) for the pseudoinverse of $\mathbf{\Phi}$.

The SVD is calculated like this: $\mathbf{\Phi}_{d\times n} = \mathbf{U}_{d\times d}\,\mathbf{\Sigma}_{d\times n}\,\mathbf{V}_{n\times n}{}^{\mathrm{T}}$. Let's denote the matrix of column vectors of the normalized eigen-vectors of matrix $\mathbf{\Phi}$ by $\mathrm{eig}(\mathbf{\Phi})$. Let's denote the eigenvalues by $\mathrm{eigval}(\mathbf{\Phi})$. Then, $\mathbf{U} = \mathrm{eig}(\mathbf{\Phi}\,\mathbf{\Phi}^{\mathrm{T}})$, $\mathbf{V} = \mathrm{eig}(\mathbf{\Phi}^{\mathrm{T}}\,\mathbf{\Phi})$ and $\mathbf{\Sigma} = \mathrm{diag}(\mathrm{eigval}(\mathbf{\Phi}\,\mathbf{\Phi}^{\mathrm{T}}))_{d\times n}$. Then, the pseudoinverse is $\mathbf{\Phi}^{+} = \mathbf{V}\,\mathbf{\Sigma}^{+}\,\mathbf{U}^{\mathrm{T}}$. For $\mathbf{\Sigma}^{+}$, we take the inverse of the non-zero elements of $\mathbf{\Sigma}$, and add zeros such that it has the shape of $d \times n$.

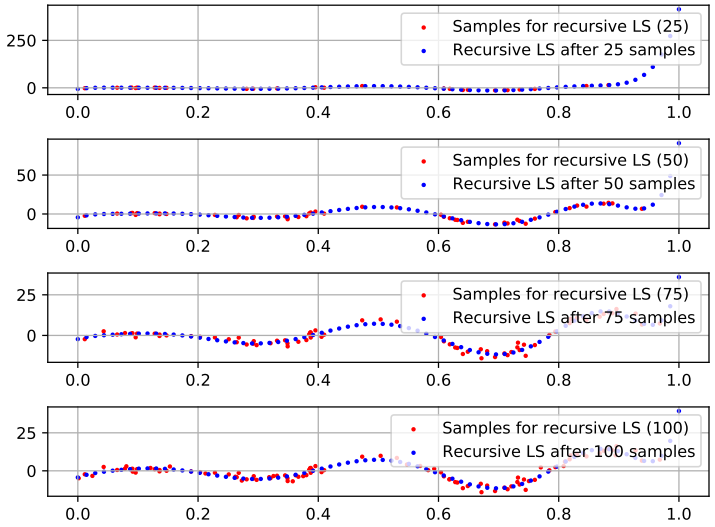
The resulting plots are shown in figure 1b.



(a) Least-squares estimate for thin $\mathbf{\Phi}$



(b) Least-norm estimate for fat $\mathbf{\Phi}$



(c) Recursive least-squares

Figure 1: Experiments with ordinary least-squares

1.2 Approximating auto-regressive series

A recursive time-series is generated from the give equation: $y_t = a\, y_{t-1} + b\, y_{t-2} + \epsilon_t$. Let's calculate the least-squares estimate using $\boldsymbol{\varphi}_t = \left[y_{t-1}, y_{t-2}\right]$, $\boldsymbol{\Phi} = \left[\boldsymbol{\varphi}_i \dots \boldsymbol{\varphi}_n\right]$. Then $\hat{\boldsymbol{\theta}} = \boldsymbol{\Phi}^+ \mathbf{y}$.

The time plots can be seen on figure 2a.

Let's calculate the inverse of the covariance matrix:s $\boldsymbol{\Gamma}_n = \frac{1}{n} \boldsymbol{\Phi}^{\mathrm{T}} \boldsymbol{\Phi}$. Now define $\Delta\boldsymbol{\theta} := (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}_n)$. The confidence ellipsoid is given by

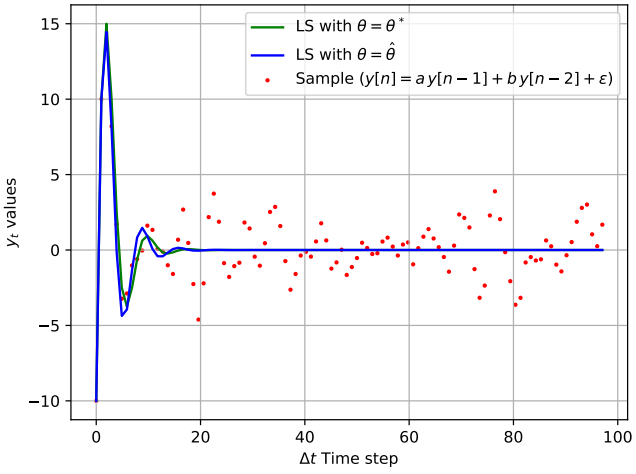
$$\Delta\boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\Gamma}_n \Delta\boldsymbol{\theta} \leq \frac{q\, \hat{\sigma}_n^2}{n}, \tag{2}$$

where q is calculated from the inverse of the cumulative χ^2 distribution function given the p probabilities ($q = \mathrm{F}(p)^{-1}_{\chi^2(d)}$). In this problem, $d = 2$. Eq. 2 means that with probability p, the optimal $\boldsymbol{\theta}^*$ is at most $\Delta\boldsymbol{\theta}$ distance from $\hat{\boldsymbol{\theta}}$.

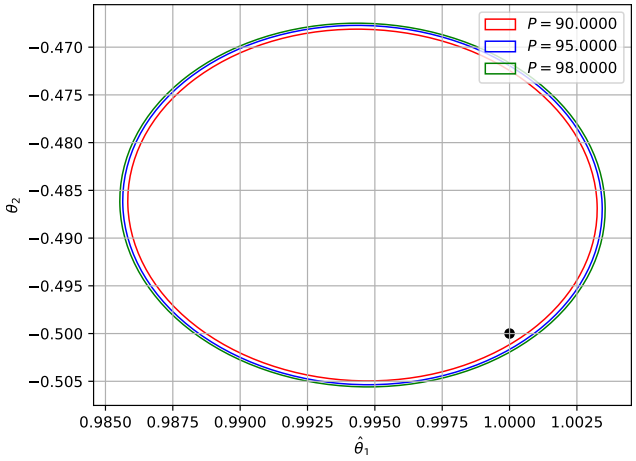
Now we assume that $\hat{\sigma}_n = 1$, and let $\Gamma_{ij}/n = [\boldsymbol{\Gamma}_n]_{ij}$. Then we have an equation that outputs an ellipse for a p probability. Written out:

$$\Delta\theta_1^2\, \Gamma_{11} + 2\, \Delta\theta_1\, \Delta\theta_2\, \Gamma_{12} + \Delta\theta_2^2\, \Gamma_{22} = q. \tag{3}$$

For the plotting, this equation is transformed so that the axis distances and the rotation angle is known with the function `ellipse_transform`[1].



(a) Time-series



(b) Confidence ellipsoids

Figure 2: Experiments with auto-regressive series

2 Kernel methods

3 Reinforcement learning

3.1 The environment

For the environment, the cliff walking with a size of 12×4 was chosen. Specifically, a modified version of caburu's `gym-cliffwalking`[2] is used. Two main modifications have been made to the original code:

- Originally, the state-space had a size of 48, though 10 of these are not real states. The cliff states were excluded.
- Rendering has been added.

So the environment has a state-space of size 38, and an action-space of size 4 (right, down, left, up).

3.2 Model based methods

3.2.1 Model generating

Firstly, the model has been generated, which is basically a database of transitions and rewards. $\mathrm{m} : \mathbb{X} \times \mathbb{A} \to \mathbb{R} \times \mathbb{X}$, where $\mathbb{X}$ is the state-space and $\mathbb{A}$ is the action-space. A random policy was used here. A figure of this can be seen running `3-Reinforcement_learning/show_scene.py` (taking the given action in the given state, on the left, the numbers mean the rewards, on the right, the next states).

3.2.2 Linear programming

The optimal solution is given by linear programming. Let V be an arbitrary value-function. The optimal V^* is obtained by minimizing $\sum \mathrm{V}_S$, given

$$\mathrm{V}_S \geq \mathrm{g}(S, A) + \beta \mathrm{V}_{S+1} \text{ and } \mathrm{V}_{\text{goal}} = 0, \tag{4}$$

where β is the discount factor, and $\mathrm{g}(S, A)$ is the immediate reward. The optimization is done with the python library `cvxpy`.

The resulting optimal value-function can be seen on figure 3.

3.2.3 Value iteration

An iterative solution is using value iteration. Start with and arbitrary value-function (all zeros in this case), and repeatedly sweep through the state-space. For all the states

$$\mathrm{V}_S^{n+1} = \max_{A \in \mathbb{A}} \left(R + \beta \mathrm{V}_{S+1}^n \right), \tag{5}$$

where $(S, A) \Rightarrow R$ and $(S, A) \Rightarrow S + 1$ can be queried from the model. Then, $\lim_{n\to\infty} \mathrm{V}^n \to \mathrm{V}^*$.

The value-functions at some iterations are shown on figure 4.

3.2.4 Policy iteration

An other iterative method is policy iteration. Here we start with an arbitrary policy, then we evaluate it by calculating its value-function. This is done similarly to 5, only A is not the one with the maximal reward, but the one given by the current policy.

After this, we make the policy greedy with respect to the calculated value-function: $\mathrm{p}(S) = \mathrm{argmax}_{A\in\mathbb{A}} (R + \beta \mathrm{V}_{S+1})$.

The value-functions at some iterations are shown on figure 5.

The (euclidean) distances of both V_{VI} and V_{PI} from V^* are shown on figure 9a.

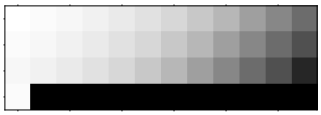
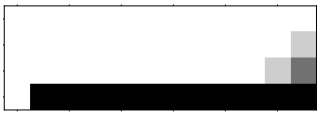
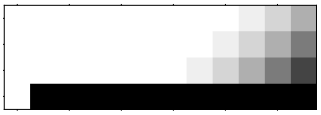


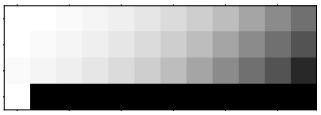
Figure 3: Linear programming



(a) Iteration 2

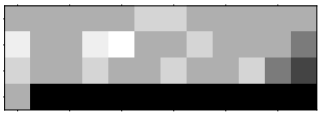


(b) Iteration 5

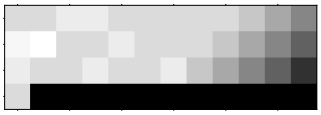


(c) Iteration 12

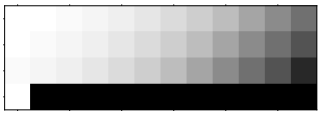
Figure 4: Value iteration



(a) Iteration 2



(b) Iteration 5



(c) Iteration 12

Figure 5: Policy iteration

3.3 Model-free methods

In this section, online Q-learning is going to be implemented. The update rule of Watkins' Q-learning is as follows:

$$\mathrm{Q}_{n+1}(S, A) = (1 - \gamma_n)\, Q_n(S, A) + \gamma_n \left(R + \beta \max_{B\in\mathbb{A}} \mathrm{Q}_n(\widetilde{S}, B)\right), \tag{6}$$

where $\gamma_n = \frac{1}{n+1}$ is the learning rate at step n, and $\widetilde{S}$ is the next state. At every step, $A = \mathrm{p}(S)$ is given by the policy.

Three policies are going to be put to test.

Firstly, the random policy, which just generates random actions for every state.

Second, the ϵ-greedy policy. This acts greedily (see paragraph 3.2.4), most of the time, with acts randomly with an ϵ probalility so as to encourage exploration.

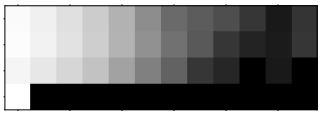
Lastly, the semi-greedy policy (called soft-max in the code) basically acts randomly if it doesn't have a much better choice. The exact probability of choosing action A is given by

$$\mathbb{P}\left(\pi_n(S) = A\right) = \frac{\exp\left(\mathrm{Q}_n(S, A)/\tau\right)}{\sum_{B \in \mathbb{A}} \exp\left(\mathrm{Q}_n(S, B)/\tau\right)}, \tag{7}$$

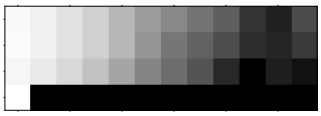
where τ is the so-called Boltzmann-temperature, which influences the randomness of the policy.

For all three policies, the value-functions at some iterations are shown on figures 6, 7 and 8.

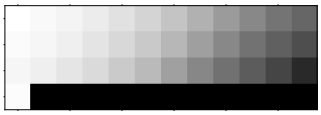
Also the distances of these policies' value-functions from the optimal one can be seen in figure 9b The sums of the rewards received are also shown in figure 9c.



(a) Iteration 2

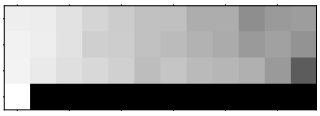


(b) Iteration 5

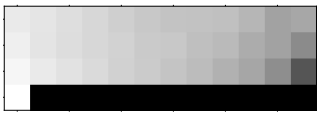


(c) Iteration 30

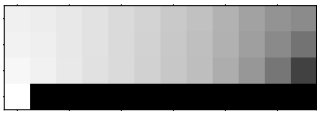
Figure 6: Random policy



(a) Iteration 2



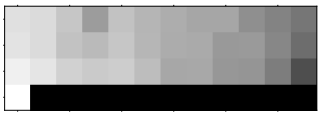
(b) Iteration 5



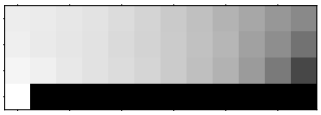
(c) Iteration 30

Figure 7: ϵ-greedy policy

(a) Iteration 2

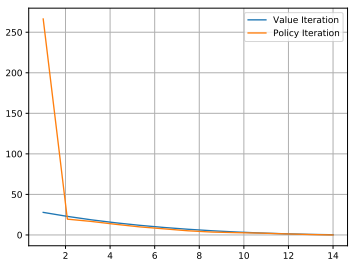


(b) Iteration 5

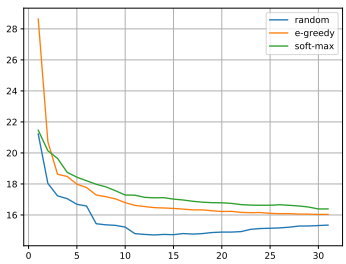


(c) Iteration 30

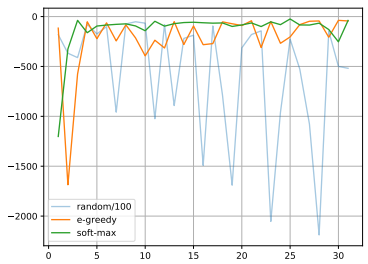
Figure 8: Semi-greedy policy



(a) Model-based



(b) Model-free



(c) Rewards (model-free)

Figure 9: Distance from the optimal value-function, and rewards of policies

References

[1] Ellipse equation:
https://www.maa.org/external_archive/joma/Volume8/Kalman/General.html,
2021. Jan 6., 12:22

[2] Cliffwalking environment:
https://github.com/caburu/gym-cliffwalking,
2021. Jan 6., 12:26