Homework

Stochastic Models and Adaptive Algorithms

Réda Vince

Z697LX

January 12, 2021

Contents

1 Linear regression **2**

1.1 Function approximation with least squares 2

1.2 Approximating auto-regressive series 4

2 Kernel methods **5**

2.1 Linear classification 5

2.2 Nonlinear classification 6

3 Reinforcement learning **8**

3.1 The environment 8

3.2 Model based methods 8

3.3 Model-free methods 9

4 Stochastic approximation **11**

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.

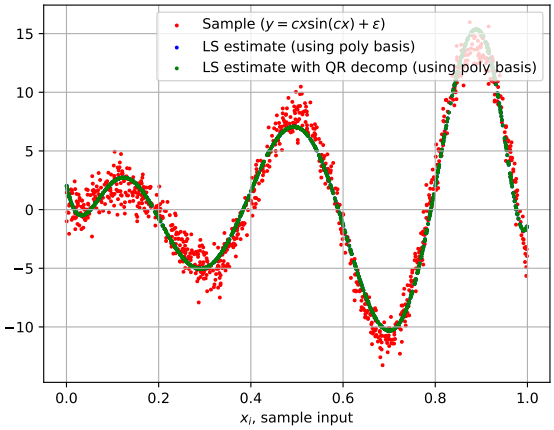
The resulting plots are shown in figure 1c., taken at $n \in [25, 50, 75, 100]$.

1.1.3 Least-norm problem

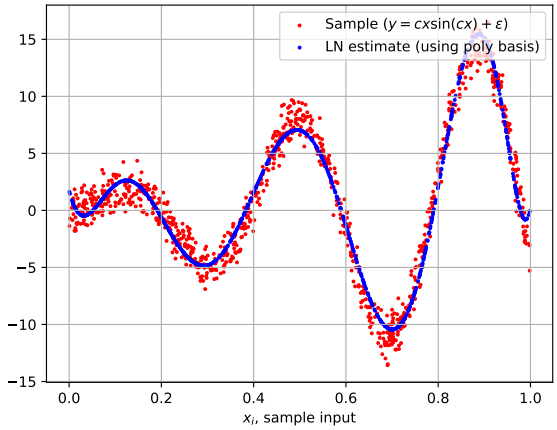
Now let's make $d > n$, specifically $n = 100$ and $d = 200$, which makes $\mathbf{\Phi}$ fat. We make the assumption that $\mathbf{\Phi}$ is still full-rank. The solution is the same, except we are going to use singular value decomposition (SVD) for the pseudoinverse of $\mathbf{\Phi}$.

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 $d \times n$.

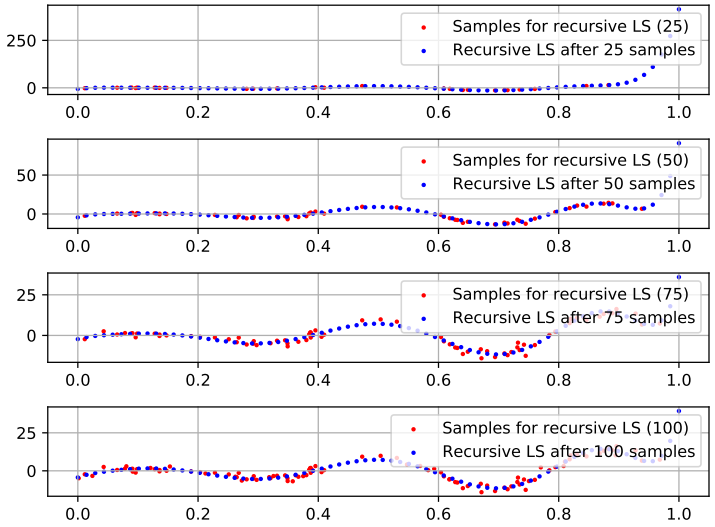
The resulting plots are shown in figure 1b.



(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 $\boldsymbol{\varphi}_t = \Big[y_{t-1}, y_{t-2}\Big]$, $\boldsymbol{\Phi} = \Big[\boldsymbol{\varphi}_i \dots \boldsymbol{\varphi}_n\Big]$. Then $\hat{\boldsymbol{\theta}} = \boldsymbol{\Phi}^+ \mathbf{y}$.

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

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

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:

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

The resulting confidence ellipsoids can be seen on figure 1a.

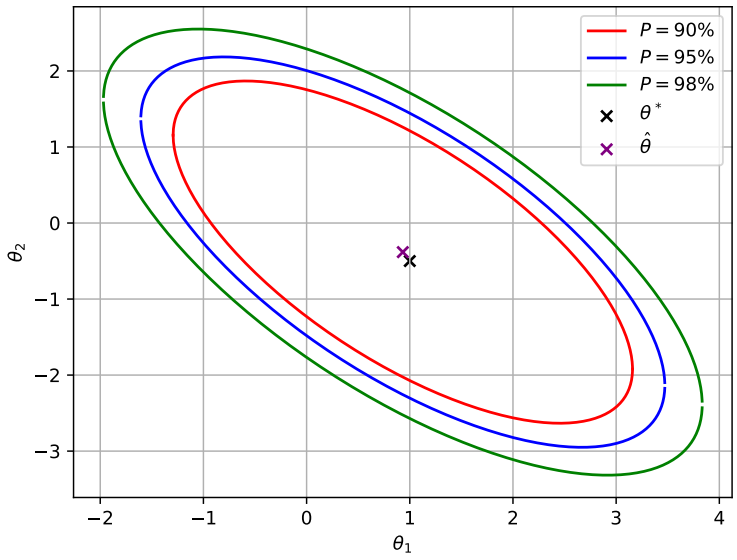


Figure 2: Confidence ellipsoids

2 Kernel methods

2.1 Linear classification

At first, we create three datasets, with different distances between the means of two classes, so that there is one with no overlap, one with heavy overlap and one semi-overlapping. Each dataset consists of the coordinates of the points $\mathbf{x}_i \in \mathbb{R}^2$, and the corresponding classification $y_i \in \{-1, 1\}$.

Three classifiers are implemented.

2.1.1 Soft Margin Support Vector Classification

The support vector classifier is a modification of Vapnik's SVC, such that badly classified data points are penalized.

The problem can be solved via convex optimization, using the cvxpy python library.

$$\begin{aligned} &\underset{\mathbf{w},\, \epsilon \in \mathbb{R}^d,\, b \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{w}\|^2 + \lambda \sum \epsilon \\ &\text{subject to} \quad y_k \left(\mathbf{w}^{\mathrm{T}} \mathbf{x}_k + b\right) \geqslant 1 - \epsilon_k \quad \text{and} \quad \epsilon_k \geqslant 0 \quad \text{for} \quad k = 1, \dots n \end{aligned} \tag{4}$$

2.1.2 Least Squares SVM

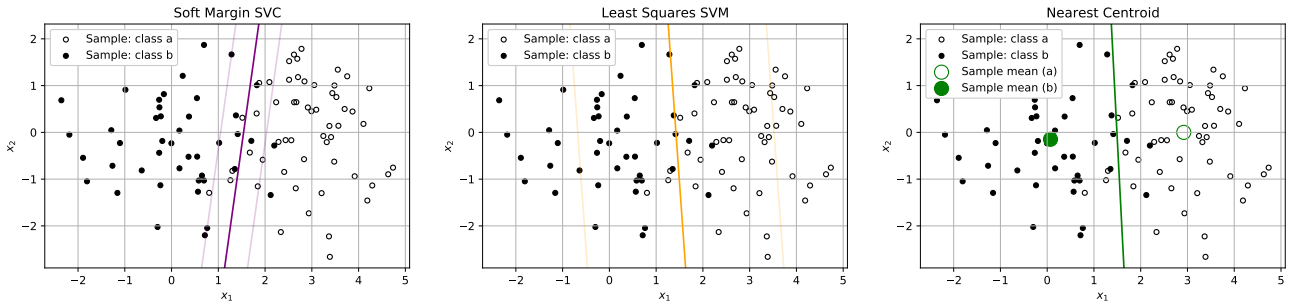
LS-SVM is the least squares reformulation of Vapnik's SVC. This too can be solved via convex optimization.

$$\begin{aligned} &\underset{w,\, \epsilon \in \mathbb{R}^d,\, b \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{2} \|\mathbf{w}\|^2 + \lambda \sum \epsilon^2 \\ &\text{subject to} \quad y_k \left(\mathbf{w}^{\mathrm{T}} \mathbf{x}_k + b\right) = 1 - \epsilon_k \quad \text{for} \quad k = 1, \dots n \end{aligned} \tag{5}$$

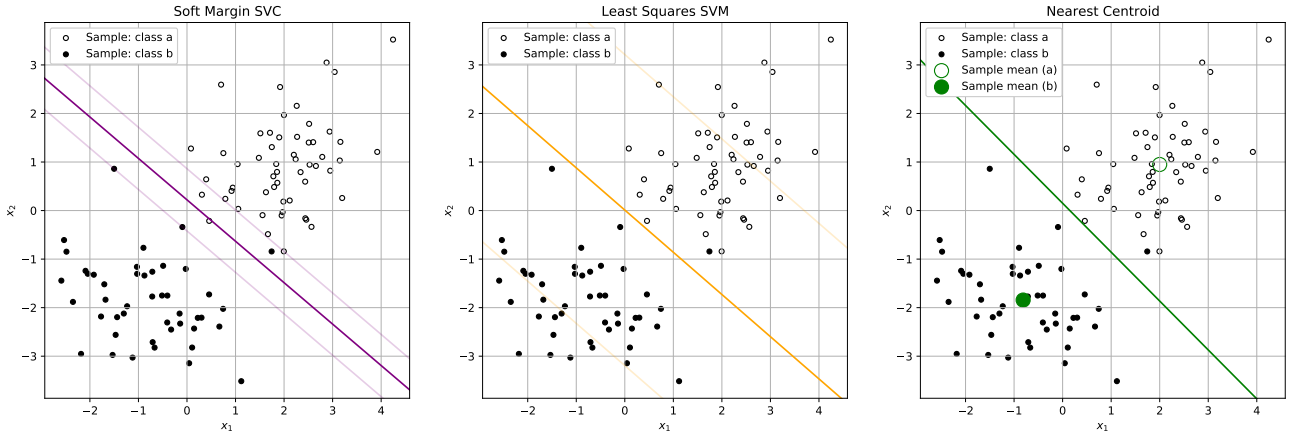
2.1.3 Nearest centroid classifier

With the NNC solution, the means of the classes are calculated. Now a new sample would be classified in the class whose mean it is closer to.

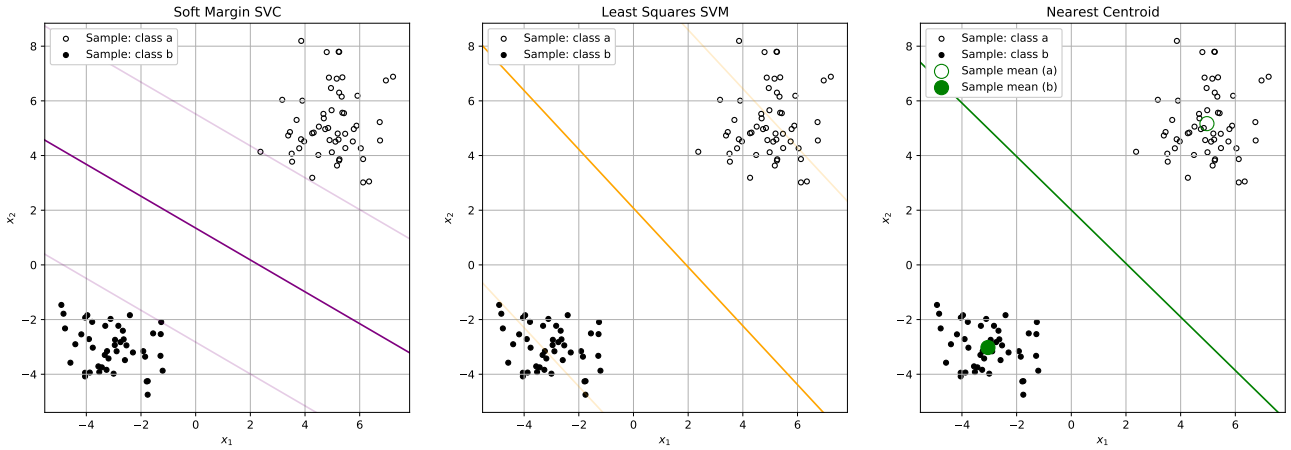
The plots of the classifiers applied to the different datasets can be seen on figure 3.



(a) Heavily overlapping distributions



(b) Semi-overlapping distributions



(c) Non-overlapping distributions

Figure 3: Linear classification experiments

2.2 Nonlinear classification

In the next exercise a linearly not separable problem is going to be solved via the kernelized soft margin SVC.

At first, a dataset is generated much like before, except that one of them concentrically surrounds the other. For this to be solved via an SVC, the points need to be mapped to a Reproducing Kernel Hilbert Space, called the feature space. This is

done with the Gaussian kernel:

$$\mathrm{k}(\mathbf{x}, \mathbf{y}) = \exp \frac{-\|\mathbf{x} - \mathbf{y}\|^2}{\sigma^2}. \tag{6}$$

Then we define matrix $\mathbf{K}$ as $\mathbf{K}_{ij} = \mathrm{k}(\mathbf{x}_i, \mathbf{x}_j)$.

This poses a convex optimization problem, the dual of which is going to be solved:

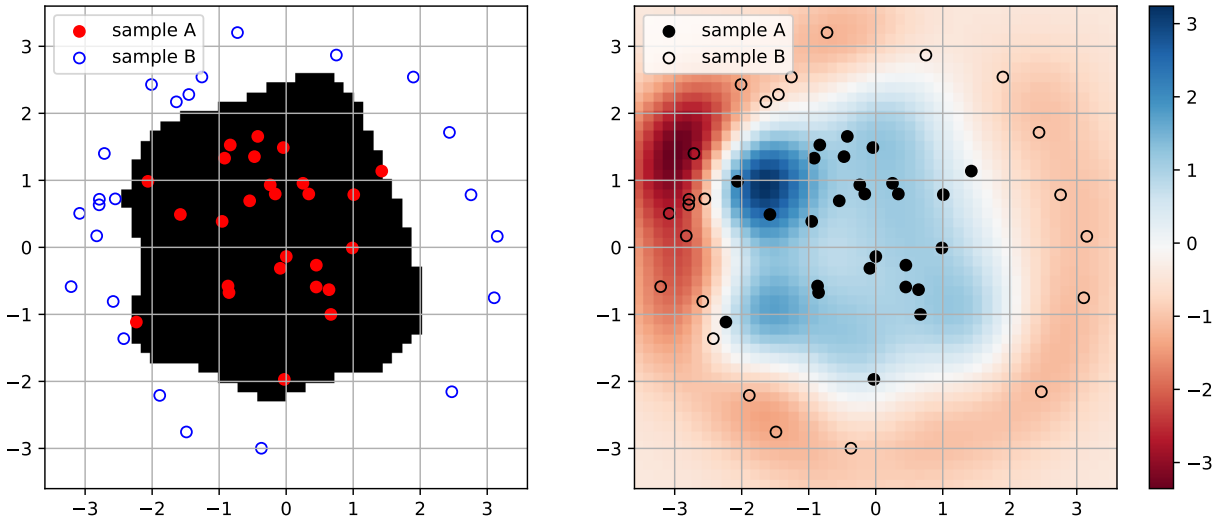
$$\begin{aligned} &\underset{\alpha \in \mathbb{R}^n}{\text{maximize}} \quad \sum_{k=1}^{n} \alpha_k - \frac{1}{2} (\boldsymbol{\alpha} \odot \mathbf{y})^{\mathrm{T}} \mathbf{K} (\boldsymbol{\alpha} \odot \mathbf{y}) \\ &\text{subject to} \quad \sum_{k=1}^{n} \alpha_k \, y_k = 0 \quad \text{and} \quad \lambda \geqslant \alpha_k \geqslant 0 \quad \text{for} \quad k = 1, \dots n, \end{aligned} \tag{7}$$

where $\odot$ indicates elementwise multiplication. This ensures sparsity, specifically, for every $\alpha_k \neq 0$, x_k is a support vector. b^* can be calculated from KKT: $b^* = y_{\text{supp}} - \sum_{k=1}^{n} \alpha_k^* y_k \mathrm{k}(\mathbf{x}_{\text{supp}}, \mathbf{x}_k)$.

From this, the classifier function is the following:

$$\mathrm{f}(\mathbf{x}) = \operatorname{sign} \left[\sum_{k=1}^{n} \alpha_k^* \, y_k \mathrm{k}(\mathbf{x}, \mathbf{x}_k) + b^* \right] \tag{8}$$

The dataset along with the classification boundaries is shown on figure 4. Both the classification boundaries and the underlying (smooth) function is shown.



(a) Classifier with taking the sign

(b) Classifier without taking the sign

Figure 4: Nonlinear classification

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`[1] is used. The main modification was the following. Originally, the state-space had a size of 48, though 10 of these are not real states. The cliff states were excluded. 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

$$\mathrm{V}_S \geqslant \mathrm{g}(S, A) + \delta\, \mathrm{V}_{S+1} \text{ and } \mathrm{V}_{\text{goal}} = 0, \tag{9}$$

where δ 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 5b.

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

$$\mathrm{V}_S^{n+1} = \max_{A \in \mathbb{A}} \left(R + \delta\, \mathrm{V}_{S+1}^n \right), \tag{10}$$

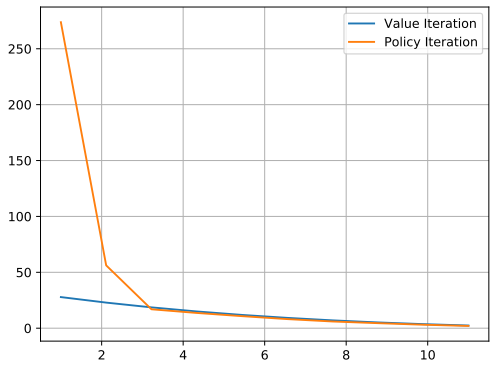
where $(S, A) \Rightarrow R$ and $(S, A) \Rightarrow S + 1$ can be queried from the model. Then, $\lim_{n \to \infty} \mathrm{V}^n \to \mathrm{V}^*$.

3.2.4 Policy iteration

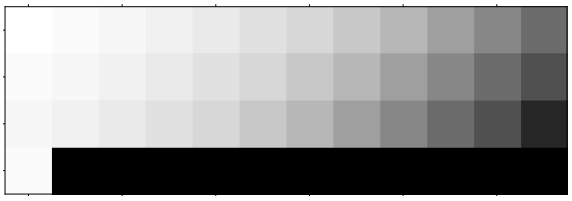
An other iterative method is policy iteration. Here we start with an arbitrary policy, then we evaluate it by calculating its value-function. This is done similarly to 10, only A is not the one with the maximal reward, but the one given by the current policy.

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

The (euclidean) distances of both V_{VI} and V_{PI} from V^* are shown on figure 5a.



(a) Distances of value-functions from the optimal one



(b) Optimal value-function

Figure 5: Model-based results

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:

$$\mathrm{Q}_{n+1}(S, A) = (1 - \gamma_n) \, Q_n(S, A) + \gamma_n \left(R + \delta \, \max_{B \in \mathbb{A}} \mathrm{Q}_n(\widetilde{S}, B)\right), \tag{11}$$

where $\gamma_n = \frac{1}{n+1}$ is the learning rate at step n, and $\widetilde{S}$ is the next state. The speed of decay can be adjusted by setting $\gamma_n = \frac{1}{r\,n+1}$ with $r > 0$. 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.

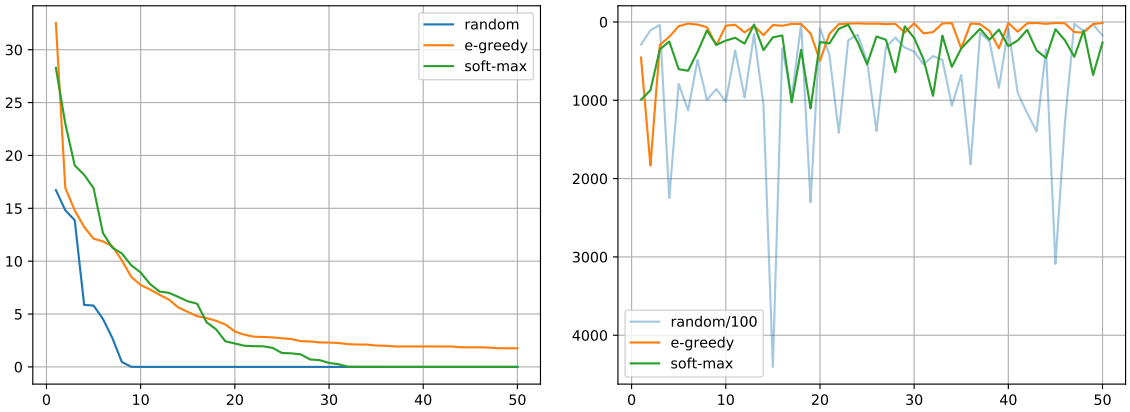
Second, the ϵ-greedy policy. This acts greedily (see paragraph 3.2.4), most of the time, with acts randomly with an ϵ probalility so as to encourage exploration.

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

$$\mathbb{P}\left(\pi_n(S) = A\right) = \frac{\exp\left(\mathrm{Q}_n(S, A)/\tau\right)}{\sum\limits_{B \in \mathbb{A}} \exp\left(\mathrm{Q}_n(S, B)/\tau\right)}, \tag{12}$$

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

Also the distances of these policies' value-functions from the optimal one can be seen in figure 6a The sums of the rewards received are also shown in figure 6b.



(a) Distances of value-functions from the optimal one

(b) Rewards per iteration

Figure 6: Model-free results

4 Stochastic approximation

Let's take the online Q-learning algorithm developed in subsection 3.3. The ϵ-greedy strategy is used for this experiment. Learning can be sped up using momentum acceleration, and Polyak averaging.

Momentum acceleration, per visit learning rate

Let's modify the update rule for Q to the following:

$$\mathrm{Q}_{n+1}(S,A) = (1-\gamma_n)\,\mathrm{Q}_n(S,A) + \gamma_n\left(R + \delta \max_{B\in\mathbb{A}} \mathrm{Q}(\widetilde{S},B)\right) + \beta_n\left[\mathrm{Q}_n(S,A) - \mathrm{Q}_{n-1}(S,A)\right]. \tag{13}$$

β_n is the coefficient of momentum acceleration. This method can increase convergence rate and helps bypass local minima.

Moreover, set $\gamma_n = \frac{c_1}{\mathrm{n}(S,A)}$ and $\beta_n = \frac{c_2}{\mathrm{n}(S,A)}$. $\mathrm{n}(S,A)$ denotes the number of visits to the state-action pair (S,A), and c_i are additional coefficients. This makes sure that rarely visited state-action pairs are updated with a sufficient learning rate.

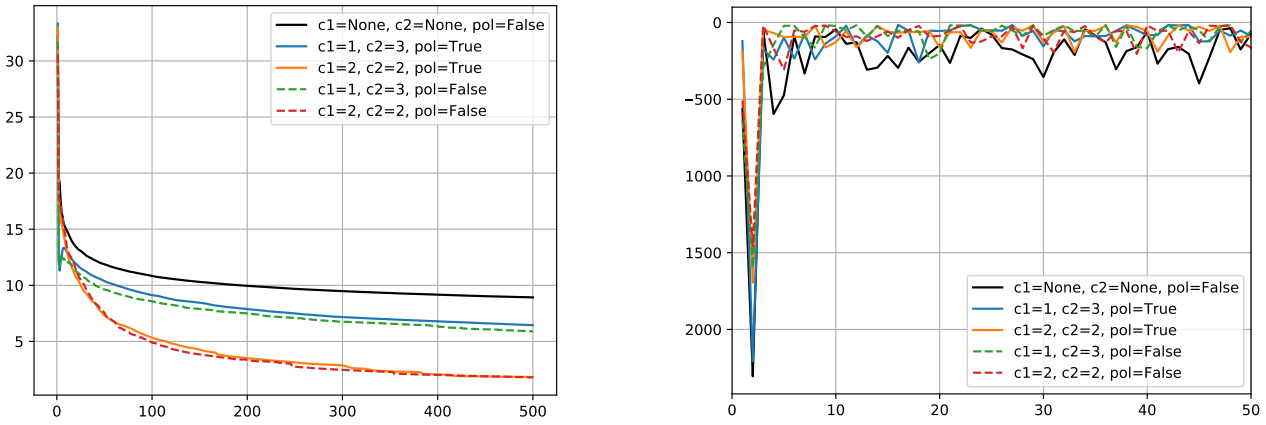
Polyak averaging

The purpose of Polyak averaging is to smooth out errors in Q_n. This is done by defining

$$\bar{\mathrm{Q}}_n = \frac{1}{\tau_n}\sum_{t=n-\tau_n+1}^{n} \mathrm{Q}_t. \tag{14}$$

Here, $\{\tau_n\}$ is the window width, such that $\tau_n \to \infty$ and $n - \tau_n \geqslant 0$. As an example, $\tau_n = \mathrm{ceil}(\log(n))$ was used.

The results are shown in figure 7. Figure 7b. has been cropped so that it is more understandable.



(a) Distances from optimal value-function

(b) Rewards at each iteration

Figure 7: Results of the extended Q-learning

References

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