Machine Learning HW 1

Student ID: **2018280070** Name: **Peter Garamvoelgyi** Score:

1.1 Optimization using the Lagrange multiplier method

$$\min_{x_1, x_2} x_1^2 + x_2^2 - 1 \quad \text{s.t.} \quad x_1 + x_2 - 1 = 0, \quad x_1 - 2x_2 \geq 0$$

First, let us derive the Lagrangian by encoding the constraints into the objective function:

$$L(x_1, x_2, \alpha, \beta) = x_1^2 + x_2^2 - 1 - \alpha(x_1 + x_2 - 1) - \beta(x_1 - 2x_2) \quad \text{s.t.} \quad \beta \geq 0$$

… thus, our target becomes:

$$\min_{x_1, x_2} \max_{\alpha, \beta} L(x_1, x_2, \alpha, \beta)$$

Let us take the derivatives w.r.t. the parameters of L:

$$\frac{\partial L}{\partial x_1} = 2x_1 - \alpha - \beta = 0 \quad \rightarrow x_1^* = \frac{\alpha + \beta}{2} \quad (1)$$

$$\frac{\partial L}{\partial x_2} = 2x_2 - \alpha + 2\beta = 0 \quad \rightarrow x_2^* = \frac{\alpha - 2\beta}{2} \quad (2)$$

$$\frac{\partial L}{\partial \alpha} = 1 - x_1 - x_2 = 0 \quad (3)$$

$$\frac{\partial L}{\partial \beta} = 2x_2 - x_1 = 0 \quad (4)$$

By plugging x_1^* and x_2^* back into (3) and (4), we get:

$$\left.\begin{aligned} 2\alpha - \beta &= 2 \\ \alpha - 5\beta &= 0 \end{aligned}\right\} \rightarrow \begin{aligned} \alpha^* &= 10/9 \\ \beta^* &= 2/9 \end{aligned}$$

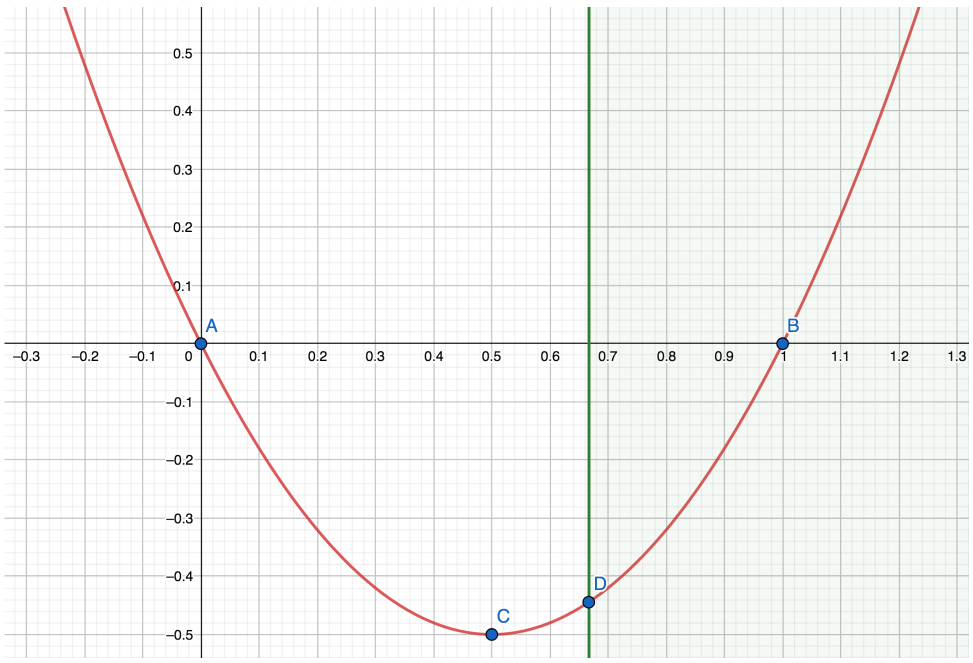
Plugging these back into (1) and (2), we get

$$x_1^* = \frac{2}{3} \quad x_2^* = \frac{1}{3}$$

Note, however, that such simple problems can simply be solved graphically.

$$\min_{x_1} 2x_1(x_1 - 1) \quad \text{s.t.} \quad x_1 \geq \frac{2}{3}$$

The term we are trying to minimize is a quadratic function with $x_1 \in \{0, 1\}$ being zero values (A and B).



When dealing with such quadratic functions, we can easily find the minimum (as shown above). The merit of the Lagrange multiplier method is that it can be uniformly applied to various kinds of complex optimization problems.

1.2. Coin toss

Let X_k be a random variable representing the number of tosses needed to encounter a continuous sequence of k tails. Using $e_k = E[X_k]$, we can construct the following recurrence relation:

$$e_k = \frac{1}{2}(e_{k-1} + 1) + \frac{1}{2}(e_{k-1} + 1 + e_k) \quad \rightarrow \quad e_k - 2e_{k-1} - 2 = 0$$

… with the initial conditions $e_0 = 0$ and $e_1 = 2$.

Intuitively, this means that given $k-1$ tails, our next toss is either T, in which case our result is $(e_{k-1} + 1)$, or H, in which case we have to start again, i.e. our result is $(e_{k-1} + 1 + e_k)$. These two outcomes have a probability of $\frac{1}{2}$ each.

Let us turn this into a linear homogeneous recurrence relation:

$$\left.\begin{array}{l} e_k - 2e_{k-1} - 2 = 0 \\ e_{k-1} - 2e_{k-2} - 2 = 0 \end{array}\right\} \rightarrow e_k - 3e_{k-1} + 2e_{k-2} = 0$$

The corresponding characteristic equation is:

$$q^2 - 3q + 2 = (q-1)(q-2) = 0$$

Our candidate becomes:

$$e_k = c_1 \cdot 1^k + c_2 \cdot 2^k$$

Using the initial conditions, we get

$$\left.\begin{array}{l} 0 = c_1 + c_2 \\ 2 = c_1 + 2c_2 \end{array}\right\} \rightarrow c_2 = 2 \quad c_1 = -2$$

… i.e. the closed-form solution for our recurrence relation is:

$$e_k = E[X_k] = -2 + 2 \cdot 2^k = 2 \cdot (2^k - 1)$$

It is easy to see that the expected value of any *specific* sequence of length k is described by the same formula (as long as we have a fair coin). Thus,

$$E\left[H\,\underbrace{TT \ldots T}_{k}\right] = E[X_{k+1}] = 2 \cdot (2^{k+1} - 1)$$

2. SVM

Given the training data

$$\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N} \; (\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R})$$

and a fixed small value $\epsilon > 0$, we can describe SVM regression as follows:

$$\min_{\boldsymbol{w},b,\boldsymbol{\xi},\hat{\boldsymbol{\xi}}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N}(\xi_i + \hat{\xi}_i)$$

$$\text{s.t. } \forall\, i = 1, \dots, N: \quad \begin{cases} y_i \leq \boldsymbol{w}^T\boldsymbol{x}_i + b + \epsilon + \xi_i \\ y_i \geq \boldsymbol{w}^T\boldsymbol{x}_i + b - \epsilon - \hat{\xi}_i \\ \xi_i \geq 0 \\ \hat{\xi}_i \geq 0 \end{cases}$$

The above formulation is the *primal soft SVM regression*. Let us apply the method of the Lagrange multipliers to arrive at the dual problem:

$$L(\boldsymbol{w}, b, \boldsymbol{\xi}, \hat{\boldsymbol{\xi}}, \boldsymbol{\alpha}, \hat{\boldsymbol{\alpha}}, \boldsymbol{\beta}, \hat{\boldsymbol{\beta}}) =$$

$$= \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{N}(\xi_i + \hat{\xi}_i) - \sum_{i=1}^{N}\alpha_i(\epsilon + \xi_i + \boldsymbol{w}^T\boldsymbol{x}_i + b - y_i)$$

$$- \sum_{i=1}^{N}\hat{\alpha}_i(\epsilon + \hat{\xi}_i - \boldsymbol{w}^T\boldsymbol{x}_i - b + y_i) - \sum_{i=1}^{N}\beta_i\xi_i - \sum_{i=1}^{N}\hat{\beta}_i\hat{\xi}_i$$

… i.e. we use $4N$ Lagrange multipliers ($\alpha_i \geq 0,\ \hat{\alpha}_i \geq 0,\ \beta_i \geq 0,\ \hat{\beta}_i \geq 0,\ i = 1, \dots, N$) to encode the $4N$ constraints. Let us set the derivatives to 0 so that we find the optimum:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \boldsymbol{w} + \sum_{i=1}^{N}(\hat{\alpha}_i - \alpha_i)\boldsymbol{x}_i = 0 \quad \rightarrow \quad \sum_{i=1}^{N}(\alpha_i - \hat{\alpha}_i)\boldsymbol{x}_i = \boldsymbol{w} \qquad (1)$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{N}(\hat{\alpha}_i - \alpha_i) = 0 \quad \rightarrow \quad \sum_{i=1}^{N}(\alpha_i - \hat{\alpha}_i) = 0 \qquad (2)$$

$$\frac{\partial L}{\partial \xi_i} = C - \alpha_i - \beta_i = 0 \quad \rightarrow \quad \alpha_i + \beta_i = C \qquad (3)$$

$$\frac{\partial L}{\partial \hat{\xi}_i} = C - \hat{\alpha}_i - \hat{\beta}_i = 0 \quad \rightarrow \quad \hat{\alpha}_i + \hat{\beta}_i = C \qquad (4)$$

Plugging these constraints back into the formula, many terms cancel out. Finally we get:

$$\tilde{L}(\boldsymbol{\alpha}, \widehat{\boldsymbol{\alpha}}) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\alpha_i - \hat{\alpha}_i)(\alpha_j - \hat{\alpha}_j)\boldsymbol{x}_j^T\boldsymbol{x}_i - \epsilon\sum_{i=1}^{N}(\alpha_i + \hat{\alpha}_i) + \sum_{i=1}^{N} y_i(\alpha_i - \hat{\alpha}_i)$$

Using the original Lagrange constraints as well as (3) and (4), we get:

$$\forall\, i = 1, \dots, N\colon \begin{cases} 0 \le \alpha_i \le C \\ 0 \le \hat{\alpha}_i \le C \end{cases}$$

… thus, the dual problem becomes

$$(\boldsymbol{\alpha}^*, \widehat{\boldsymbol{\alpha}}^*) = \underset{0\le\alpha_i\le C;\ 0\le\hat{\alpha}_i\le C}{\arg max}\ \tilde{L}(\boldsymbol{\alpha}, \widehat{\boldsymbol{\alpha}})$$

Using $\boldsymbol{\alpha}^*$ and $\widehat{\boldsymbol{\alpha}}^*$ we can derive the optimal weights:

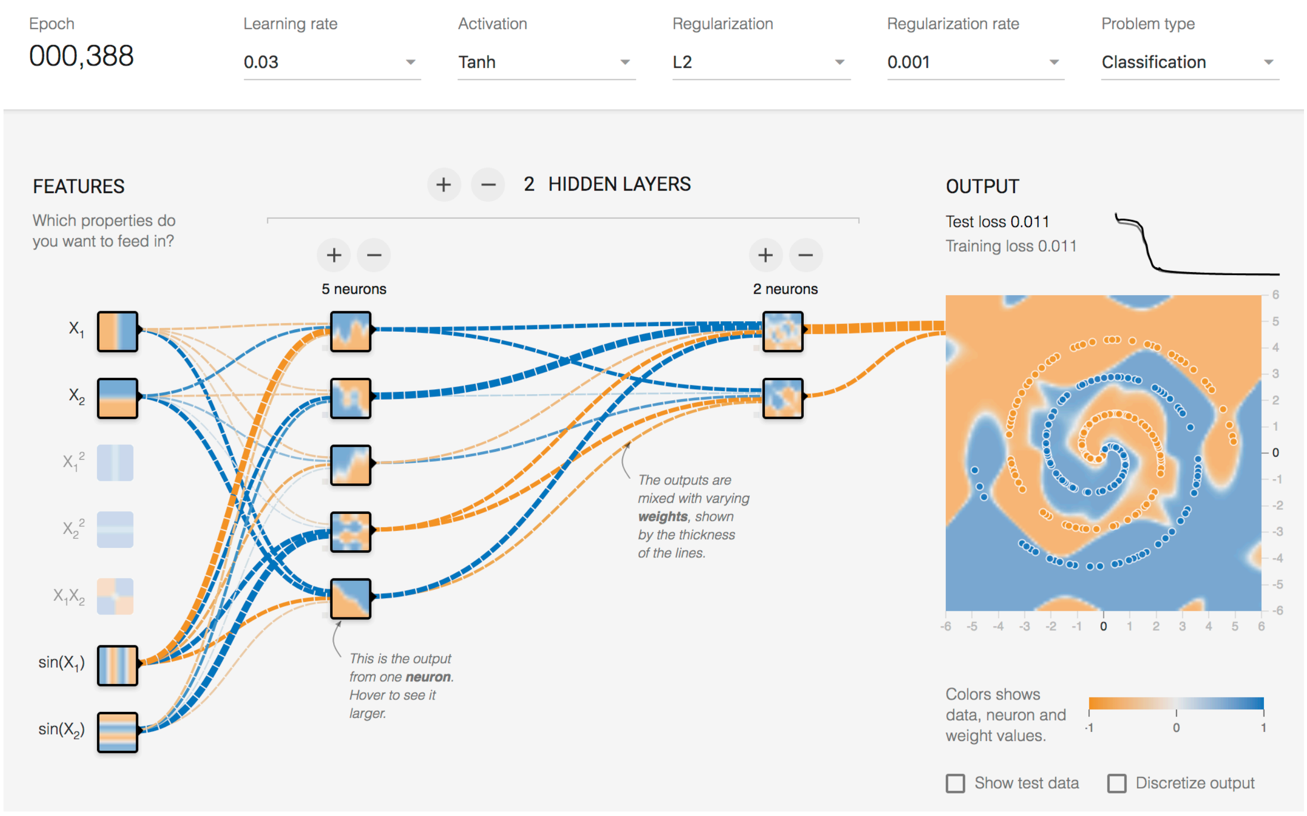
$$\boldsymbol{w}^* = \sum_{i=1}^{N}(\alpha_i^* - \hat{\alpha}_i^*)\boldsymbol{x}_i \qquad b^* = \cdots$$

Using this, prediction can be done using the following formula:

$$y(\boldsymbol{x}_{new}) = \boldsymbol{w}^{*T}\boldsymbol{x}_{new} + b = b + \sum_{i=1}^{N}(\alpha_i^* - \hat{\alpha}_i^*)\boldsymbol{x}_i^T\boldsymbol{x}_{new}$$

3. Deep Neural Networks

1. Best configuration for the spiral dataset



Epoch 000,388

Learning rate 0.03

Activation Tanh

Regularization L2

Regularization rate 0.001

Problem type Classification

Test loss 0.011

Training loss 0.011

*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

*This is the output from one **neuron**. Hover to see it larger.*

Colors shows data, neuron and weight values.

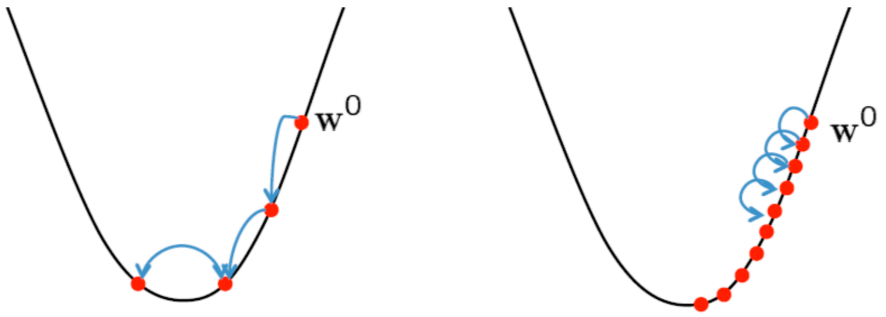
Many configurations perform very similarly on this problem. The one I ended up using is shown above. The configuration:

- 4 input features: linear and sin
- 2 hidden layers with 5 and 2 neurons respectively
- tanh activation function
- L2 regularization with very small regularization rate

2. Insights into how the parameters influence performance and convergence rate

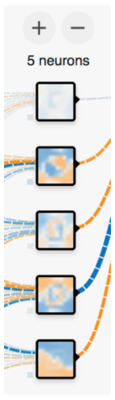
- **Learning rate**: The learning rate is basically the weight for change during optimization (e.g. during gradient descent). If the learning rate is large, we will potentially converge faster, but we might *jump over* optima, and some points even become unreachable. On the other hand, a small learning rate makes it possible to approximate local minima better, but convergence becomes slower.

  Given a convex error surface, the effect of the learning rate (i.e. step size) can be visualized as follows (image taken from course slides):



The error surface of DNNs is typically more complex and has multiple local minima, so the situation is more complex too. Often, the best approach is to use a dynamic learning rate: large at first so that we can quickly converge to a relatively good solution, then decrease it, so that we can better approximate the optimum.

- **Number of hidden layers**: In theory, increasing the number of hidden layers increases the number of complex features our network can capture. This can potentially result in a better approximation of the target function. However, the more layers and neurons we have, the more training data is needed.

  Sometimes, after adding neurons and training the network, we can notice that some neurons do not really contribute to the output (see the top neuron on the right). This shows that increasing the number of neurons does not necessarily mean better performance.

- **Activation function**: For the activation function, we can choose from the sigmoid function, the hyperbolic tangent function, and ReLu.

  For sigmoid, we have the problem of vanishing and saturating gradients. These might cause slower convergence or even the lack of further improvements from a give state. I also experienced that the network often did not converge if I used sigmoid.

  The hyperbolic tangent function is better in most cases but still has the vanishing gradients issue. ReLu is generally preferable for deep learning as it handles the vanishing gradients problem.

- **Regularization**: Regularization can help prevent overfitting by including a regularization term in the objective function, thus making sure that the resulting weights are not too large.

  My experience was that even regularization rates as small as 0.01 can sometimes prevent convergence of the network, that is why I decided to use a very small rate.

4. IRLS for Logistic Regression

Given a binary classification problem with the *logistic regression* decision rule

$$P_{\boldsymbol{w}}(y|\boldsymbol{x}) = \frac{\exp(y\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x})}{1+\exp(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x})} \quad \boldsymbol{x} \in \mathbb{R}^d \;\; y \in \{0,1\}$$

… giving the likelihood of a sample $\boldsymbol{x}$ being classified as y given our model $\boldsymbol{w}$, and the log-likelihood

$$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} (y_i \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}} - \log(1+\exp(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}})))$$

… the L2-norm regularized logistic regression is defined by the optimization problem

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \left(-\frac{\lambda}{2} \|\boldsymbol{w}\|_2^2 + \mathcal{L}(\boldsymbol{w}) \right) \quad \lambda \geq 0$$

To solve this problem, we can employ the method of *Iteratively Reweighted Least Squares (IRLS)*:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \mathbf{H}^{-1}\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}_t)$$

The calculation of the Hessian and the gradient can be done as follows[1]:

$$\mathbf{H}_{\boldsymbol{t}} = \mathbf{X}^{\boldsymbol{T}}\mathbf{R}_{\boldsymbol{t}}\,\mathbf{X} + \lambda\mathbf{I}_{\boldsymbol{d}}$$

$$\nabla_{\boldsymbol{w}}\mathcal{L}(\boldsymbol{w}_t) = \mathbf{X}^{\boldsymbol{T}}(\boldsymbol{\mu}_t - \boldsymbol{y}) + \lambda\boldsymbol{w}_t$$

… where $\mathbf{X}$ is a matrix containing all the traning examples, $\boldsymbol{y}$ is a vector containing all the traning labels, $\boldsymbol{w}_t$ is our current model, $\mathbf{I}_{\boldsymbol{d}}$ is the identity matrix, λ is the regularization parameter, and

$$\mu_{t,i} = \mathrm{sigm}(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_{\boldsymbol{i}})$$

… and $\mathbf{R}_{\boldsymbol{t}}$ is a diagonal matrix where

$$\mathbf{R}_{\boldsymbol{t},ii} = \mu_{t,i}(1-\mu_{t,i})$$

Having expressed the problem like this, it is fairly straightforward to implement it. I used Python, heavily relying on numpy for linear algebra. We can initialize the weights

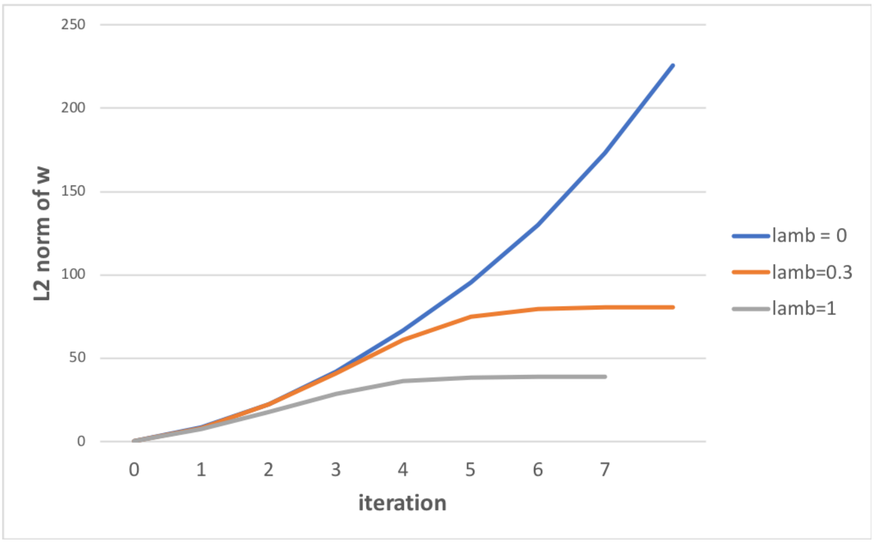
[1] Source: page 25 of https://www.cs.ubc.ca/~murphyk/Teaching/CS540-Fall08/L6.pdf

to 0s first, then iteratively adjust them using the formula above. The stopping condition is when the change in the gradient $\|\mathbf{H}^{-1}\nabla_{w}\mathcal{L}(\boldsymbol{w}_t)\|$ is below a certain threshold.

Below are some example outputs with- and without regularization;

$\boldsymbol{\lambda = 0}$	$\boldsymbol{\lambda = 0.3}$
--------------	--------------
>> iteration #0	>> iteration #0
L = -22569.565346212377	L = -22569.565346212377
acc_train = 24.0810%	acc_train = 24.0810%
acc_test = 23.6226%	acc_test = 23.6226%
diff = 0	diff = 0
L2 norm = 0.0	L2 norm = 0.0
--------------	--------------
--------------	--------------
>> iteration #1	>> iteration #1
L = -12414.373632533689	L = -12414.560529202518
acc_train = 84.5275%	acc_train = 84.5275%
acc_test = 84.5280%	acc_test = 84.5280%
diff = 2.868786807381289	diff = 2.8612209593094975
L2 norm = 8.229937746204929	L2 norm = 8.18658537799196
--------------	--------------
…	…
--------------	--------------
>> iteration #8	>> iteration #8
L = -10504.864652344353	L = -10506.395899310794
acc_train = 84.9145%	acc_train = 84.9145%
acc_test = 84.9948%	acc_test = 85.0009%
diff = 2.203547695633255	diff = 0.008601348059918785
L2 norm = 225.86124946358385	L2 norm = 80.59238452011788
--------------	--------------

While achieving similar accuracy, the regularized model has much smaller weights. This suggests that regularization decreases overfitting in this case. Another insight is that without regularization the algorithm did not converge, while regularized runs usually converged to some result within 10 iterations.


L2 norm of w
iteration
lamb = 0
lamb=0.3
lamb=1