Tutorial 1 Solutions: Learning the Ising Hamiltonian using linear regression

February 23, 2025

The objective of this tutorial is to use supervised learning, and specifically linear regression, to learn the Hamiltonian for a one-dimensional Ising model from randomly-generated data. You will reproduce results from Section VI.D of Reference [1].

Recall from Lecture 2 that the input to a linear regression algorithm is a dataset $\mathcal{D} = \{(\overrightarrow{x}^{(i)}, y^{(i)})\}_{i=1}^{M}$, where $\overrightarrow{x}^{(i)} \in \mathbb{R}^{d_x}$ and M is the number of samples. The goal is to fit a linear function $f(\overrightarrow{x})$ to the labels y, where

$$f(\overrightarrow{x}) = \sum_{j=1}^{d_x} w_j x_j, \tag{1}$$

and w_j are fitting parameters. One can solve for w_j by implementing the exact solution

$$\overrightarrow{w}_{\text{exact}} = (X^T X)^{-1} X^T \overrightarrow{y}, \tag{2}$$

or by using gradient descent. Note that in the equation above, the elements of X are given by $X_{ij} = x_j^{(i)}$, and $\overrightarrow{y} = (y^{(1)}, y^{(2)}, \ldots, y^{(M)})^T$ is the label vector.

In this tutorial, we will consider a dataset that is generated from the one-dimensional nearest-neighbour Ising model, for which the Hamiltonian is given by

$$H_{\texttt{data}} = -\sum_{j=1}^{N} s_j s_{j+1}, \tag{3}$$

where N is the total number of spins and s_j are the spin variables, with $s_j \in \{-1, 1\}$. Note that we have set the coupling energy to 1 and we assume periodic boundary conditions. Let us generate our dataset $\mathcal{D} = \{(\overrightarrow{s}^{(i)}, H^{(i)})\}_{i=1}^{M}$ such that each input data point is a spin configuration $\overrightarrow{s} = (s_1, s_2, \ldots, s_N)^T$ and each label is the corresponding energy $H_{\texttt{data}}(\overrightarrow{s})$.

Let us further impose that our supervised learning algorithm does not have knowledge of the Hamiltonian used to generate the labels, and instead assumes a more general energy function that accounts for the pairwise interactions from all possible pairs of spins such that

$$H_{\texttt{model}} = -\sum_{j=1}^{N} \sum_{k>j} J_{jk} s_j s_k. \tag{4}$$

The goal of our supervised learning algorithm is find the coupling parameters J_{jk} such that the model $f(\overrightarrow{s}) \equiv H_{\texttt{model}}(\overrightarrow{s})$ that can predict the labels (energies $H_{\texttt{data}}$) from the spin configurations.

a) Compute the second derivative of the loss function we have seen in lecture 2 and justify that setting the first derivatives of the loss to zero corresponds to a minimum and not a maximum.

Solution: We can show that the Hessian matrix

$$\frac{\partial L}{\partial w_j \partial w_i} = 2(\mathbf{X}^T\mathbf{X})_{ij}.$$

All we need to do is show that $\mathbf{X}^T\mathbf{X}$ is a matrix that has positive eigenvalues (positive semi-definite matrix in other terms).

b) If we succeed at finding a model that can correctly predict the labels for all spin configurations such that $H_{\texttt{model}}(\overrightarrow{s}) = H_{\texttt{data}}(\overrightarrow{s})$ for all $\overrightarrow{s}$, state the values you would expect for the coupling parameters J_{jk}.

Solution: If $H_{\texttt{model}}(\overrightarrow{s}) = H_{\texttt{data}}(\overrightarrow{s})$ for all $\overrightarrow{s}$, then we would have

$$J_{jk} = \begin{cases} 1 & \text{for } k = j+1, \\ 0 & \text{otherwise.} \end{cases}$$

c) Show that one can rewrite the model Hamiltonian in terms of a new data vector $\overrightarrow{x} \in \mathbb{R}^{d_x}$ such that

$$H_{\texttt{model}} = \sum_{p=1}^{d_x} w_p x_p, \tag{5}$$

where $\overrightarrow{x}$ stores the all possible pairwise interactions $\{s_j s_k\}$ for $1 \leq j \leq N$ and $k > j$. Note that this new vector $\overrightarrow{x}$ enters linearly in $H_{\texttt{model}}$ in the same form as in Equation (1). Give an expression for w_p in terms of J_{jk}, and for d_x in terms of N.

Solution: Let p be the index for the elements of the coordinate set $\{(j,k)\}$ for $1 \leq j \leq N$ and $k > j$. One can then write Equation (4) as

$$H_{\texttt{model}} = -\sum_{p=1}^{d_x} J_p x_p,$$

with $d_x = N(N-1)/2$. Defining $w_p = -J_p$ then gives Equation (5).

d) Open the notebook on this link, which generates a random one-dimensional Ising dataset as described above and attempts to apply linear regression. Start by examining the function `getEnergy_nnIsing1D(s)` within the section labelled "Generate the data set". Currently, this code sets the label (the energy) to zero for all spin configurations in the data set. Modify this function so that it calculates the energy as is Equation (3).

Hint: Using the functions `np.sum` and `np.roll`, you can calculate the energy using only one line of code.

Solution: Replace the replace the return statement inside the function `getEnergy_nnIsing1D(s)` with

```
return -J*np.sum(s*np.roll(s,-1,axis=1),axis=1)
```

e) After computing the labels for each spin configuration, this code constructs a matrix `x`. Currently, the code constructs a matrix of zeros of size $M \times N \times N$ and then takes its upper triangular elements to get a smaller matrix of size $M \times d_x$. Modify the code that sets `x` to zero such that it

instead stores the products $s_j s_k$ as discussed in part c). You do not need to modify the portion where the upper triangular elements are extracted.

Hint: Using `np.einsum`, you can complete this part using only one line of code.

Solution: Replace the line `x = np.zeros((M,N,N))` with
`x = np.einsum('...j,...k->...jk', s, s)`
or with
`x = np.einsum('ij,ik->ijk', s, s)`

f) After familiarizing yourself with the structure of the remainder of the notebook, examine the section labelled "Find the parameters J_{jk} using the exact solution". Currently, this code assigns each parameters w_p to zero. Modify the code so that it calculates `w_exact` by implementing the exact solution in Equation (2) using matrix methods from the package `numpy`. Check that your results agree with your prediction from part b), as well as the results from gradient descent in the section of code that follows.

Hint: You may wish to use the functions `np.linalg.inv` for matrix inversion, `np.dot` for matrix multiplication, and `np.ndarray.T` to transpose.

Solution: Replace the line defining `w_exact` with the following line of code:
`w_exact = np.linalg.inv(x.T.dot(x)).dot(x.T).dot(H)`

g) Set the number of spins to `N = 40`, and vary the number of samples from `M=1000` down to `M=10`. Explain the behaviour that you see when attempting to solve for `w_exact`.

Solution: When the number of samples M becomes too small, the matrix $X^T X$ is no longer invertible and Equation (2) can no longer be used to solve for `w_exact`.

h) Examine the section labelled "Find the parameters J_{jk} using gradient descent". You should find that the results of this section of code agree with the exact solution from the previous parts when `N=20`, `M=1000` and the learning rate is set to `eta0=0.001`. Explain what happens when you increase or decrease the learning rate.

Solution: When the learning rate is too high, the algorithm is not able to find a good solution and the magnitudes of the couplings J_{jk} become very large. When the learning rate is very small, the algorithm does not make significant changes to the couplings J_{jk} from their random initial values.

i) Using computational complexity algorithms, explain why using gradient descent can be computationally more efficient.

Solution: The complexity of computing $X^T X$ that has size $d_x \times d_x$ is $\mathcal{O}(d_x^2 M)$. Thus the cost of computing the gradients:

$$\frac{\partial L}{\partial w_i} = 2(\mathbf{X}^T \mathbf{X} \mathbf{w})_i - 2(\mathbf{X}^T \mathbf{y})_i$$

is also $\mathcal{O}(d_x^2 M)$. For the exact approach, we need to invert $X^T X$ which costs $\mathcal{O}(d_x^3)$ which has a worse scaling compared to gradient descent (assuming we can converge within a reasonable number of gradient descent steps). Gradient descent is also helpful because it handles the case of a non-invertible $X^T X$ matrix.

j) Modify your code again so that it now calculates the parameters `w_exact` using the function `linear_model.LinearRegression` from the package `sklearn`. Additionally, remove the

`penalty`, `learning_rate` and `eta0` arguments to the function `SGDRegressor()`. You will likely find that your code becomes more efficient since it will now use more sophisticated methods to solve for `w_exact` and `w_GD`.

Solution: Replace the line defining `w_exact` with:
`w_exact = linear_model.LinearRegression().fit(x, H).coef_`

Replace the line defining `w_GD` with:
`w_GD = linear_model.SGDRegressor().fit(x, H, coef_init=w_init).coef_`

k) Set the number of samples to `M = 10000`, and vary the number of spins from `N=10` to `N=70`. Explain how the efficiency of solving Equation (2) compares with the efficiency of the gradient descent algorithm for these various values of `N`.

Solution: As `N` increases, it becomes more efficient to use gradient descent rather than Equation (2).

You can find all the solutions notebook on the Github repository.

References

[1] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, `https://arxiv.org/abs/1803.08823`.