Lecture 2

Linear Regression, Perceptron, Stochastic Gradient Descent

Reading

1. Bishop 3.1, 4.1, 4.3
2. Goodfellow Chapter 5.1-5.4
3. "Random features for large-scale kernel machines" by Rahimi and Recht (2008).

2.1 Problem setup for machine learning

Nature gives us data X and targets Y for this data.

$$X \to Y.$$

Nature does not usually tell us what property of a datum $x \in X$ results in a particular prediction $y \in Y$. We would like to learn to imitate Nature, namely predict y given x.

What does such learning mean? It is simply a notion of being able to identify patterns in the input data without explicitly programming a computer for prediction. We are often happy with a learning process that identifies correlations: if we learn correlations on a few samples $(x^1, y^1), \ldots, (x^n, y^n)$, we may be able to predict the output for a new datum x^{n+1}. We may not need to know *why* the label of x^{n+1} was predicted to be so and so.

Let us say that Nature possesses a probability distribution P over (X, Y). We will formalize the problem of machine learning as Nature drawing n independent and identically distributed samples from this distribution. This is

denoted by

$$D_{\text{train}} = \left\{(x^i, y^i) \sim P\right\}_{i=1}^{n}$$

is called the "training set". We use this data to identify patterns that help make predictions on some future data.

What is the task in machine learning?

Suppose D_{train} consists of $n = 50$ RGB images of size 100×100 of two kinds, ones with an orange inside them and ones without. 10^4 is a large number of pixels, each pixel taking any of the possible 255^3 values. Suppose we discover that one particular pixel, say at location $(25, 45)$, takes distinct values in all images inside our training set. We can then construct a predictor based on this pixel. This predictor, it is a binary classifier, perfectly maps the training images to their labels (orange: +1 or no orange: -1). If x_{ij}^k is the $(ij)^{\text{th}}$ pixel for image x^k, then we use the function

❓ How many such binary classifiers are there at most?

$$f(x) = \begin{cases} y^k & \text{if } x_{ij}^k = x_{ij} \text{ for some } k = 1, \ldots, n \\ -1 & \text{otherwise.} \end{cases}$$

This predictor certainly solves the task. It correctly works for all images in the training set. Does it work for images outside the training set?

Our task in machine learning is to learn a predictor that works *outside* the training set. The training set is only a source of information that Nature gives us to find such a predictor.

> Designing a predictor that is accurate on D_{train} is trivial. A hash function that memorizes the data is sufficient. This is NOT our task in machine learning. We want predictors that generalize to new data outside D_{train}.

2.1.1 Generalization

If we never see data from outside D_{train} why should we hope to do well on it? The key is the distribution P. Machine learning is formalized as constructing a predictor that works well on new data that is also drawn independently from the distribution P. We will call this set of data the "test set".

$$D_{\text{test}}.$$

This assumption is important. It provides coherence between past and future samples: past samples that were used to train and future samples that we will wish to predict upon.

How to find such predictors that work well on new data? The central idea in machine learning is to restrict the set of possible binary functions that we consider.

> We are searching for a predictor that generalizes well but only have the training to ascertain which predictor works well.

The *right* class of functions f cannot be too large, otherwise we will find our binary classifier above as the solution and that is not too useful. The class of functions cannot be too small either, otherwise we won't be able to predict difficult images. If the predictor does not even work well on the training set, why should we expect it to work on the test set!

Finding this correct class of functions with the right balance is what machine learning is all about.

❓ Can you now think how is machine learning different from other fields you might know such as statistics or optimization?

2.2 Linear regression

Let us focus on a simpler problem. We fix the class of functions, our predictors, to only have linear classifiers. We will consider that our data $X \subset \mathbb{R}^d$ and labels $Y \subset \mathbb{R}$. If the labels/targets are real-valued, we call it is a regression problem. Our predictor for any $x \in X$ is

$$f(x; w, b) = w^\top x + b. \tag{2.1}$$

This is a linear function in the data x with parameters $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$. Different settings of w and b give access to different functions f. Picking a particular function f is therefore akin to picking particular values of the parameters. Parameters are also called weights. We can visualize what this predictor does in two ways, consider the case of $d = 2$.

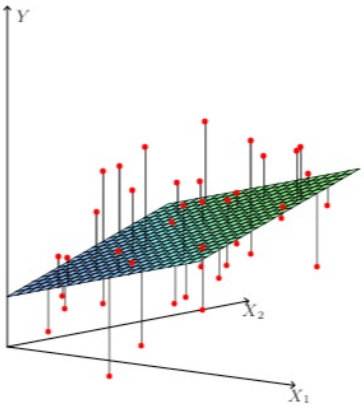


Figure 2.1: Linear least squares with $X \subset \mathbb{R}^2$.

Figure 2.1 shows the hyperplane corresponding to a particular (w, b) with the data x^i, y^i (in red). Each hyperplane is a particular predictor $f(x; w, b)$. You can also think of the function f as a point in three dimensional space $w \in \mathbb{R}^2$ and $b \in reals$.

Predicting the target accurately using this linear model would require us to find values (w, b) that minimize the average distance to the hyperplane of each sample in the training dataset. We write this as an *objective function*.

$$\begin{aligned} \ell(w, b) &:= \frac{1}{2n} \sum_{i=1}^{n} \left(y^i - \widehat{y^i} \right)^2 \\ &= \frac{1}{2n} \sum_{i=1}^{n} \left(y^i - x^\top x^i - b \right)^2 \end{aligned} \tag{2.2}$$

where we have written the prediction as

$$\widehat{y^i} = x^\top x^i + b.$$

The quadratic term for each datum $\frac{1}{2}\left(y^i - \widehat{y^i}\right)^2$ is known as the *loss function*. The objective above is thus an average of the loss for each datum. Finding the best weights w, b now boils down to solving the optimization problem

$$w^*, b^* = \underset{w\in\mathbb{R}^d, b\in\mathbb{R}}{\text{argmin}} \quad \ell(w, b) \tag{2.3}$$

How to solve the optimization problem? We will learn many techniques to solve problems of the form (2.3). We have a simple case here and therefore can use what you did did in HW0. The solution is given by

$$w^* = (\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}})^{-1} \tilde{\mathbf{X}}^\top \mathbf{Y} \tag{2.4}$$

where we have denoted by $\tilde{\mathbf{X}} \in \mathbb{R}^{n\times(d+1)}$ the matrix whose i^{th} row is the datum with a constant entry 1 appended at the end $[x^i, 1]$. Similarly $\mathbf{Y} \in \mathbb{R}^n$ is a vector whose i^{th} entry is the target y^i.

Why use the average, as opposed to say the maximum value?

When is our solution to least squares regression in (2.4) not defined?

What are we losing by fitting a linear predictor? Will this work if the true model from which Nature generates the data was different, say a polynomial?

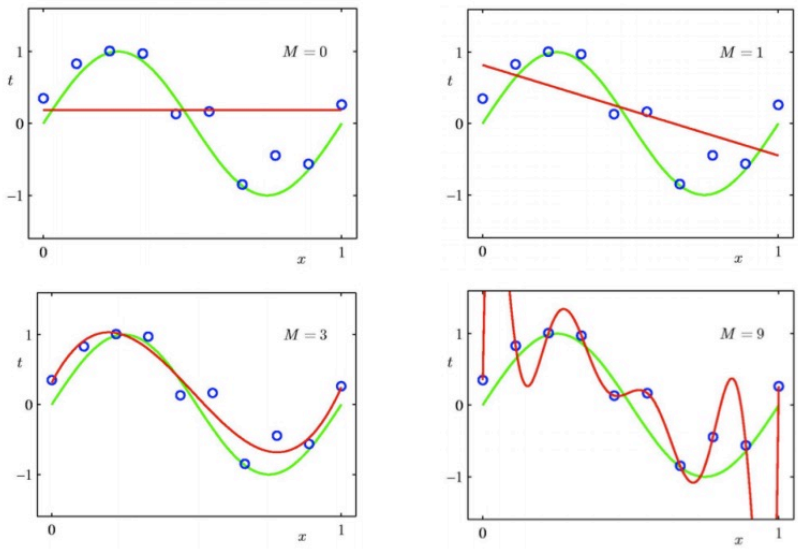


Figure 2.2: Least squares fitting using polynomials. As the degree of the polynomial M increases the predictor f fits the training data (in blue) better and better. But such a well-fitted predictor may be very different from the true model from which Nature generated the data (in green). The red curve in the fourth panel in these cases is said to have been *over-fitted*.

2.2.1 Maximum Likelihood Estimation

There is another perspective to fitting a machine learning model. We will *suppose* that our training data was created using a statistical model. We write this as

$$y = w^\top x + b + \epsilon \tag{2.5}$$

Of course we do not know whether Nature used this particular model $f(x; w, b) := w^\top + b$ to create the data, it might have created the data using some other model. This discrepancy between the models is *modeled* as noise ϵ. Noise in machine learning comes from the fact that we the user do not know Nature's model.

What model is appropriate for the noise ϵ? There can be many models depending upon your experiment (think of a model that predicts the arrival time of a bus at the bus stop, what noise would you use?). For our purpose we will use zero-mean Gaussian noise

$$\epsilon \sim N(0, \sigma_\epsilon^2)$$

that does not depend on the sample x. The probability that a sample (x^i, y^i) in our dataset D_{train} was created using our statistical model is then

$$p(y^i|x^i, w, b) = N(w^\top x^i + b,\ \sigma_\epsilon^2).$$

We have assumed that the data was drawn iid by Nature so the likelihood of our entire dataset is

$$p(D_{\text{train}}|w, b) = \prod_{i=1}^{n} p(y^i|x^i, w, b).$$

Finding good values of w, b can now be thought of as finding values that maximize the likelihood of our observed data

$$w^*, b^* = \underset{w,b}{\text{argmin}}\ -\log p(D_{\text{train}}|w, b). \tag{2.6}$$

Observe that our objective is written as the minimization of the *negative log-likelihood*. This is equivalent to maximizing the likelihood because logarithm is monotonic function. We can now rewrite the objective as

$$-\log p(D_{\text{train}}|w, b) = \frac{n}{2}\log(\sigma_\epsilon^2) + \frac{n}{2}\log(2\pi) + \frac{1}{2\sigma_\epsilon^2}\sum_{i=1}^{n}\left(y^i - w^\top x^i - b\right)^2.$$

Notice that only the third term depends on w, b. The first term is a function of our *chosen* value σ_ϵ^2, the second term is a constant. In other words, finding maximizing the likelihood boils down to solving the optimization problem

$$w^*, \beta^* = \underset{w,b}{\text{argmin}}\ \frac{1}{2\sigma_\epsilon^2}\sum_{i=1}^{n}\left(y^i - w^\top x^i - b\right)^2. \tag{2.7}$$

This objective is nothing other than our least squares regression objective with σ_ϵ^2 set to 1. This objective known as the maximum likelihood objective (MLE).

Maximum likelihood objective has an interesting offshoot. In the least squares case, given an input x all that our fitted model could predict was

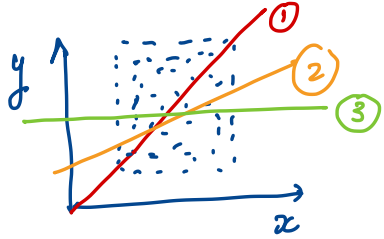
$$\widehat{y} = {w^*}^\top x + b^*.$$

MLE has helped us fit a statistical model to the data. So we can now predict the entire distribution

$$p(y|x, w^*, b^*) = N({w^*}^\top x + b^*, \sigma_\epsilon^2).$$

❓ Can you think any other sources of noise? For instance, if you scraped some images from the Internet, how will you label them?

❓ How does using a different value of σ_ϵ in (2.7) change the least squares solution in (2.4)?

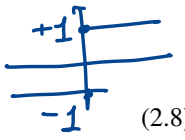


$\frac{1}{2}(y^i - \hat{y})^2$

The solution of least squares is the mean of the Gaussian random variable $y|x, w^*, b^*$, the variance of this random variance is σ_ϵ^2. So instead of just predicting $\widehat{y}$ the machine learning model can now give the probability distribution $p(y|x, w^*, b^*)$ as the output and the user is free to use it as they wish, e.g., compute the mean, the median, the 5% probability value of the right tail etc.

2.3 Perceptron

Let us now solve a classification problem. We will again go around the model selection problem and consider the class of linear classifiers. Assume binary labels $Y \in \{-1, 1\}$. To keep the notation clear, we will use the trick of appending a 1 to the data x and hide the bias term b in the linear classifier. The predictor is now given by



$$f(x; w) = \text{sign}(w^\top x) = \begin{cases} +1 & \text{if } w^\top x \geq 0 \\ -1 & \text{else.} \end{cases} \tag{2.8}$$

We have used the sign function denoted as sign to get binary $\{-1, +1\}$ outputs form our real-valued prediction $w^\top x$. This is the famous perceptron model of Frank Rosenblatt. We can visualize the perceptron the same way as we did for linear regression.

Let us now formulate an objective to fit/train the perceptron. As usual, we want the predictions of the model to match those in the training data.

$$\ell_{\text{zero-one}}(w) := \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{y^i \neq f(x^i; w)\}}. \tag{2.9}$$

The indicator function inside the summation measures the number of mistakes the perceptron makes on the training dataset. The objective here is designed to find weights w that minimizes the average number of mistakes, also known as the training error. Such a loss that measures the mistakes is called the zero-one loss, it incurs a penalty of 1 for a mistake and zero otherwise.

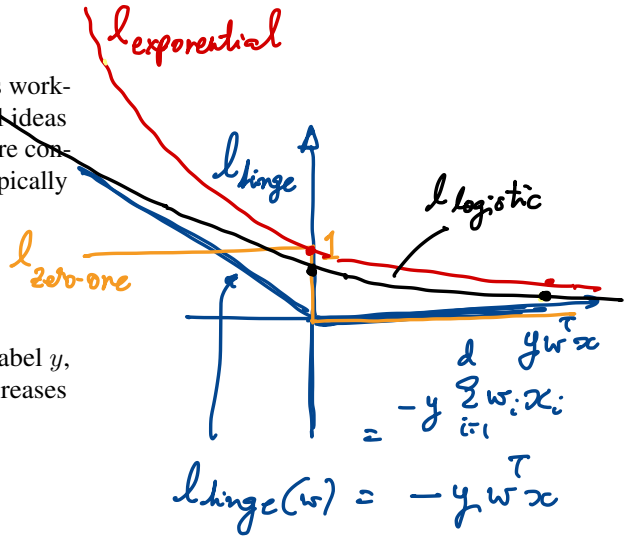
❓ Is a linear model appropriate if our data was natural images? What properties have we lost by restricting the classifier to be linear?

⚠ The linear classifier remains unchanged if we reorder the pixels of all images consistently in our entire training set and the weights w. The images will look nothing like real images to us. The perceptron does not care about which pixels in the input are close to which others.

❓ Can you think of some quantity other than the zero-one error that we may wish to optimize?

2.3.1 Surrogate Losses

The zero-one loss is the clearest indication of whether the perceptron is working well. It is however non-differentiable, so we cannot use powerful ideas from optimization theory to minimize it. This is why surrogate losses are constructed in machine learning. These are proxies for the loss function, typically for the classification problems and look as follows.



The hinge loss is one such surrogate loss. It is given by

$$\ell_{\text{hinge}}(w) = \max(0, -y\, w^\top x).$$

If the predicted label $\hat{y} = \text{sign}(w^\top x)$ have the same sign as the true label y, the hinge-loss is zero. If they have opposite signs, the hinge loss increases linearly. The exponential loss

$$\ell_{\text{exp}}(w) = e^{-y\,(w^\top x)}$$

$\frac{d}{dw}(\sum_{i=1} w_i x_i) = -y$

$\ell_{hinge}(w) = -y w^\top x$

$\left[\frac{\partial \ell}{\partial w_i} \cdots\right] = \frac{\partial}{\partial w}(\,) = -yx$

$$w^* = \min_w \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{hinge}}(x^i, y^i; w)$$

or the logistic loss

$$\ell_{\text{logistic}}(w) = \log\left(1 + e^{-yw^\top x}\right)$$

are some other popular losses for classification.

❓ Draw the three losses to observe their differences.

2.4 Stochastic Gradient Descent

We will now fit a perceptron using the hinge loss using a very simple optimization technique. At each iteration, this algorithm updates the weights w in the direction of the negative gradient. So first, let us compute the gradient of the hinge loss. It is easily seen to be

$$\frac{\mathrm{d}\ell_{\text{hinge}}(w)}{\mathrm{d}w} = \begin{cases} -y\,x & \text{for incorrect prediction} \\ 0 & \text{else.} \end{cases} \tag{2.10}$$

We will use a naive algorithm to update the weights. Here is how it goes.

The Perceptron algorithm

Perform the following steps for iterations $t = 1, 2, \ldots$.

1. At the t^{th} iteration, sample a data point from D_{train} uniformly randomly, call it (x^t, y^t).
2. Update the weights of the perceptron as

$$w_{t+1} = \begin{cases} w_t + y^t x^t & \text{if } \operatorname{sign}(w_t^\top x^t) \neq y^t \\ w_t & \text{else.} \end{cases} \tag{2.11}$$

$$w_{t+1} = w_t - \frac{d\ell_{\text{hinge}}(w)}{dw}$$

In other words, the perceptron weights is changed only if it makes a mistake on the sample (x^t, y^t). The updated perceptron improves its mistake on this sample. Observe that a mistake happens if the sign of $w^\top x^t$ and y^t are different, the product $y^t w^\top x^t$ is therefore negative. The updated weights of the perceptron now satisfy

$$\begin{aligned} y^t(w_t + y^t x^t)^\top x^t &= y^t w_t^\top x^t + y^{t2} x^{t\top} x^t \\ &= y^t w_t^\top x^t + \|x^t\|_2^2. \end{aligned}$$

In simple worlds, the value of $y^t w^\top x^t$ increases as a result of the update, it becomes more positive. If the perceptron makes mistakes on the same datum repeatedly, this value is eventually going to become positive. Of course, mistakes on other data in the training set may steer the perceptron towards other directions and it may continue to cycle ad infinitum, but it is easy to show that it ceases its updates when all data are correctly classified. More precisely, if the training data are such that they can be correctly classified using a linear predictor, then the perceptron will find this predictor after a finite number of iterations.

We have seen a powerful algorithm for machine learning. This algorithm is called stochastic gradient descent (SGD) and it is very general: so long as you can take the gradient of the objective you can execute SGD. The algorithm for fitting the perceptron above was given by Rosenblatt in 1957 and is popularly known as the "perceptron algorithm". It is interesting to note that it is simply the instantiation of SGD which was known before Robbins and Monro (1951) for the hinge loss.

❓ You may have seen the hinge loss written as $\ell_{\text{hinge}}(w) = \max(0, 1 - y\, w^\top x)$. Why the difference?

SGD is well-defined for any loss function (differtiable)

$$L(w) = \frac{1}{n} \sum_{i=1}^{n} \ell(w; x^i, y^i)$$

$$w_{t+1} = w_t - \left. \frac{dL(w)}{dw} \right|_{w_t}$$

Bibliography

Rahimi, A. and Recht, B. (2008). Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407.