Contents

Chapter 1

Introduction

Some kinds of learning include;

1.1 Supervised learning

Given a dataset of pair

$$\mathcal{D}_n = \left\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\right\} \tag{1.1}$$

we wish to establish a relationship between $x^{(i)}$ and $y^{(i)}$. Typically, $x^{(i)} \in \mathbb{R}^d$ is a representation of input, called **feature representation**. Based on the format of the output we can have different types of supervised learning:

Classification when the set of possible values of $y^{(i)}$ is discrete (small finite set). If there two possible values then the classification problem is *binary* otherwise, it is called *multi-class*.

Regression when the set of possible values of $y^{(i)}$ is continuous (or a large finite set). That is, $y^{(i)} \in \mathbb{R}^k$.

1.2 Unsupervised learning

Given a dataset we wish to find some patterns or structures in it. There are several types of unsupervised learning:

Density estimation The data is i.i.d from some distribution $p_X(x)$. The goal is to predict the probability $p_X\left(x^{(n+1)}\right)$.

Clustering the goal is to find a partitioning of the sample data that groups together samples that are similiar. Clustering is sometimes used in density estimation.

Dimensionality reduction the goal is to re-represent the same data in $\mathbb{R}^l$ where $l < d$.

1.3 Reinforcement learning

The goal is to learn a mapping from input values to output values without a direct supervision signal. There is no training set specified *a priori*. Instead, the learning problem is framed as an agent interacting with an environment. Looking at input as our states and output as a transition between states, we can assign a reward value $r^{(i,j)}$ to each such transition. We aim to find a policy π that maximizes the long-term sum or average od rewards.

Chapter 2

Supervised learning

To predict, we come up with a **hypothesis**. A hypothesis is a parametrized function that maps input to output

$$y = h(x; \theta)\,, \qquad h \in \mathcal{H}, \theta \in \Theta$$

where $\mathcal{H}$ is our *hypothesis class*. We wish to find the parameters θ that matches our data well. One way to evaluate how well our hypothesis predicts is to introduce a **loss function** (or *cost function*), $L(a, a_h)$ where a, a_h are in the output set and the loss function assigns a value to how close our prediction a_h when the actual value is a. We wish that our hypothesis to have the least loss on new data.

$$\mathcal{E}(h) = \frac{1}{n'} \sum_{i=n+1}^{n+n'} L\big(h\big(x^{(i)}; \theta\big)\,, y^{(i)}\big)$$

One way to do this is minimize the training error

$$\hat{\mathcal{E}}(h) = \frac{1}{n} \sum_{i=1}^{n} L\big(h\big(x^{(i)}; \theta\big)\,, y^{(i)}\big)$$

There are several types of loss function

0-1 Loss

$$L(a, a_h) = \begin{cases} 0 & \text{if} a = a_h \\ 1 & \text{otherwise} \end{cases}$$

Squared loss

$$L(a, a_h) = (a - a_h)^2$$

Linear loss

$$L(a, a_h) = |a - a_h|$$

Asymmetric loss For example, maybe guessing negative wrong is costlier than guessing positive wrongly, in a binary classification problem.

The model we use, typically, selects h and we need to minimize the loss (or any other optimization) on the θ so that our prediction *fits* the data. To determine a good θ we need algorithms, *learning algorithms*. Given a classifier h we can easily evaluate its performance by testing it on new data. However, to evaluate a learning algorithm we have to first train it on some data and then evaluate the resulting classifier on the testing data. Doing this multiple gives us an estimate of how well the algorithm works. In most cases, we do not have access to a lot of data (we don't now the distribution). In these cases we can re-use data using cross validation

```
Algorithm 1: cross_validate $(\mathcal{D}, k)$
    divide $\mathcal{D}$ into $k$ equally sized chunks $\mathcal{D}_1, \ldots, \mathcal{D}_k$
    for $i = 1 \rightarrow k$ do
        train $h_i$ on $\mathcal{D} - \mathcal{D}_i$
        compute $\mathcal{E}_i(h_i)$ on the test data $\mathcal{D}_i$
    return $\frac{1}{k} \sum_{i=1}^{k} \mathcal{E}_i(h_i)$
```

2.1 Linear classifiers

A linear classifier has the following form

$$h(x; \theta, \theta_0) = \operatorname{sign}\Big(\theta^T x + \theta_0\Big) \qquad \theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$$

2.1.1 Random linear classifier

One way to select the parameters θ and θ_0 is to randomly select them and return the best one.

```
Algorithm 2: random_linear_classifier $(\mathcal{D}_n, k)$
    for $j = 1 \rightarrow k$ do
        $\theta^{(j)} = Random(\mathbb{R}^d)$
        $\theta_0^{(j)} = Random(\mathbb{R})$
    $j^* = \operatorname{argmin}_{1 \leq j \leq k} \hat{\mathcal{E}}\Big(h\Big(x, \theta^{(j)}, \theta_0^{(j)}\Big)\Big)$
    return $(\theta^{(j^*)}, \theta_0^{(j^*)})$
```

2.1.2 Perceptron

A more intelligent way of finding the parameters is to update when we encounter a mistake. The simplest update rule is the **perceptron**, which is as follows

$$\theta',\ \theta_0' \leftarrow \theta + y^{(i)} x^{(i)},\ \theta_0 + y^{(i)}$$

This update increase the magnitude of $y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0)$ as shown below, and hence in enough iterations, the sign will become positive.

$$\begin{aligned}
y^{(i)} \cdot \Big(\theta'^T x^{(i)} + \theta_0'\Big) &= y^{(i)} \left(\theta + y^{(i)} x^{(i)}\right)^T x^{(i)} + y^{(i)} \left(\theta_0 + y^{(i)}\right) \\
&= y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right) + (y^{(i)})^2 \Big(\left\|x^{(i)}\right\|^2 + 1\Big) \\
&= y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right) + \left\|x^{(i)}\right\|^2 + 1
\end{aligned}$$

however it is not clear how other mistakes will affect the current guess.

Algorithm 3: perceptron $(\mathcal{D}_n, T)$

```
$\theta = 0$
$\theta_0 = 0$
for $t = 1 \to T$ do
    for $i = 1 \to n$ do
        if $y^{(i)}(\theta^T x^{(i)}) + \theta_0 \leq 0$ then
            $\theta = \theta + y^{(i)} x^{(i)}$
            $\theta_0 = \theta_0 + y^{(i)}$
return $(\theta, \theta_0)$
```

By adding another dimension to our data set we can simplify our prediction to pass through the origin

$$x' = \begin{bmatrix} x_1 & \dots & x_n & 1 \end{bmatrix}, \qquad \theta' = \begin{bmatrix} \theta & \theta_0 \end{bmatrix}$$
$$\implies \theta'^T x' = \theta^T x + \theta_0$$

Therefore, one can also simplify the Line 3 to the following

Algorithm 4: perceptron $(\mathcal{D}_n, T)$

```
$\theta = 0$
for $t = 1 \to T$ do
    for $i = 1 \to n$ do
        if $y^{(i)}(\theta^T x^{(i)}) + \theta_0 \leq 0$ then
            $\theta = \theta + y^{(i)} x^{(i)}$
return $\theta$
```

To examine the convergence of the perceptron algorithm we need the following definitions. A dataset $\mathcal{D}_n$ is **linearly separable -through the origin** if there is some θ such that

$$y^{(i)} \theta^T x^{(i)} > 0 \quad \forall i$$

and similary it is linearly seperable if there are some θ, θ_0 such that

$$y^{(i)} \cdot \left(\theta^T x^{(i)} + \theta_0\right) > 0 \quad \forall i$$

The **margin** of a labeled data point (x, y) with respect to a seperator (hyperplane) θ, θ_0 is

$$y \cdot \frac{\theta^T x + \theta_0}{\|\theta\|}$$

which basically quantifies how we θ approximates the data point (x, y) in a data set $\mathcal{D}_n$. Also the margin of $\mathcal{D}_n$ w.r.t θ, θ_0 is the minimum of all margins:

$$\min_i y^{(i)} \cdot \frac{\theta^T x^{(i)} + \theta_0}{\|\theta\|}$$

Theorem 2.1 (Perceptron convergence theorem). *If there exists a vector θ^* such that the margin of database with respect to θ^* is greater than $\gamma > 0$ and then norm $\left\|x^{(i)}\right\| \leq R$ for some R then perceptron will make at most $\left(\frac{R}{\gamma}\right)^2$ updates/mistakes.*

Proof. The idea of the proof is to put an increasing lower bound on the cosine of the angle between the k_{th} update $\theta^{(k)}$ and θ^*. Note that

$$\cos\big(\angle\theta^*, \theta^{(k)}\big) = \frac{(\theta^*)^T\theta^{(k)}}{\|\theta^*\|\,\|\theta^{(k)}\|}$$

we know that for some i, $\theta^{(k)} = \theta^{(k-1)} + y^{(i)}x^{(i)}$, then

$$\begin{aligned}(\theta^*)^T\theta^{(k)} &= (\theta^*)^T\left(\theta^{(k-1)} + y^{(i)}x^{(i)}\right)\\ &= (\theta^*)^T\theta^{(k-1)} + y^{(i)}(\theta^*)^Tx^{(i)}\\ &\geq (\theta^*)^T\theta^{(k-1)} + \gamma\,\|\theta^*\|\\ &\geq k\gamma\,\|\theta^*\|\end{aligned}$$

also

$$\begin{aligned}\left\|\theta^{(k)}\right\|^2 &= \left(\theta^{(k-1)} + y^{(i)}x^{(i)}\right)^T\left(\theta^{(k-1)} + y^{(i)}x^{(i)}\right)\\ &= \left\|\theta^{(k-1)}\right\|^2 + 2y^{(i)}(\theta^{(k-1)})^Tx^{(i)} + \left(y^{(i)}\right)^2\left\|x^{(i)}\right\|^2\\ &\leq \left\|\theta^{(k-1)}\right\|^2 + \left\|x^{(i)}\right\|^2\\ &\leq \left\|\theta^{(k-1)}\right\|^2 + R^2\\ &= kR^2\end{aligned}$$

therefore

$$\cos\big(\angle\theta^*, \theta^{(k)}\big) \geq \sqrt{k}\,\frac{\gamma}{R}$$

Since the cosine can not exceed one therefore the we at most make

$$k \leq \left(\frac{R}{\gamma}\right)^2$$

mistakes. ■

2.2 Features

2.2.1 Transformation

As we saw we can transform linearly separable dataset to another linearly seperable dataset but without an offset. What happens if the original dataset is not linear seperable? For example, *xor dataset*:

$$\mathcal{D} = \{((-1,-1),-1),((-1,1),1),((1,-1),-1),((1,1),1)\}$$

is not linearly seperable in 2 dimensions. A transformation that might be applicable here is **polynomial basis**. A polynomial basis transformation of order k, transforms a feature $x \in \mathbb{R}^d$ to

$$\phi(x) = (x_1^{\alpha_1}\ldots x_d^{\alpha_d}),\quad \sum_{i=1}^{d}\alpha_i \leq k, \forall i, \alpha_i \geq 0$$

which has $\binom{k+d}{d}$ dimension.

2.2.2 Representation

we can represent a discrete feature as

1. numeric
2. thermometer code (a vector of m booleans where $1 \ldots j$ bits are on and the rest or off)
3. one-hot (a vector of m booleans where j_{th} bit is on adn the rest are off)
4. factoring (group information of a feature based its structure maybe)

For numeric feature we would like to standardized as follow

$$\tilde{x_j} = \frac{x_j - \bar{x_j}}{\sigma}$$

Chapter 3

Logistic Regression

In machine learning we wish to optimize a function like $J(\Theta)$. Usually a function in form

$$\begin{aligned} J(\Theta) &= \left(\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\big(h\big(x^{(i)};\theta\big), y^{(i)}\big)\right) + \lambda R(\theta) \\ &= \mathcal{E}_n + \lambda R(\theta) \end{aligned}$$

where R is the **regularization function** and λ is a hyperparameter.

3.1 Linear logistic classifier

The problem of minimizing 0-1 Loss problem is NP-hard. A problem with sign is that incremental change are hard find because of the discrete nature of the function hence, to smooth out the sign function we use sigmoid function

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Equivalently, we want to make classifier that predict $+1$ when $\sigma\big(\theta^T x + \theta_0\big) > 0.5$ and -1 otherwise.

Loss on all data is inversely related to the probability that θ, θ_0 assign to the data. Assuming that points in data set are independent.

$$\begin{aligned} g^{(i)} &= \sigma\big(\theta^T x + \theta_0\big) \\ p^{(i)} &= \begin{cases} g^{(i)} & \text{if} \quad y^{(i)} = 1 \\ 1 - g^{(i)} & \text{if} \quad y^{(i)} = 0 \end{cases} \end{aligned}$$

and we wish to maximize the probability

$$\prod_{i=1}^{n} p^{(i)} = \prod_{i=1}^{n} (g^{(i)})^{y^{(i)}} (1 - g^{(i)})^{(1-y^{(i)})}$$

Using the log-likelihood

$$\implies \mathcal{L}_{LL}(p) = \sum_{i=1}^{n} y^{(i)} \log\big(g^{(i)}\big) + (1 - y^{(i)}) \log\big(1 - g^{(i)}\big)$$

3.2 Gradient descent

Algorithm 5: gradient descent $(f, \nabla f, \theta_{\text{init}}, \eta, \epsilon)$

```
$\theta^{(0)} = \theta_{\text{init}}$
$t = 0$
**repeat**
    $\theta^{(t)} = \theta^{(t-1)} - \eta \nabla f\big(\theta^{(t-1)}\big)$
**until** $\big|f\big(\theta^{(t)}\big) - f\big(\theta^{(t-1)}\big)\big| < \epsilon$
**return** $\theta$
```

Theorem 3.1. *If f is convex, for any desired accuracy ϵ there is some η such that gradient descent will converge to θ within ϵ of the optimum.*