Foundations of Data Science 2020

Boqi Chen

December 12, 2020

Contents

1

Linear Regression

1.1 Linear Models

1.1.1 Linear Models

Suppose the input is a vector $\boldsymbol{x} \in \mathbb{R}^D$ and the output is $\boldsymbol{y} \in \mathbb{R}$. We have the data $< (\boldsymbol{x}_i, y_i) >_{i=1}^N$. Where D denotes the dimension of the data and N is the size of the dataset. The linear model is given in equation 1.1.

$$y = w_0 + x_1 w_1 + \cdots + x_D w_D + \epsilon \tag{1.1}$$

Where w_0 is the bias or the intercept and ϵ is the noise.

If we have the input vector augmented to 1(i.e. $\boldsymbol{x} = [1, x_0, \cdots, x_D]$), we can simplify the linear model to the dot product of input vector $\boldsymbol{x}$ and weight $\boldsymbol{w}$ plus the noise given in equation 1.2.

$$y = \boldsymbol{x} \cdot \boldsymbol{w} + \epsilon \tag{1.2}$$

Let's look at a toy example here. Say if we want to predict the commute time into the city center and we have two variables: the distance to the city center and days of the week. We have the data shown in Table 1.1.

Distance (km)	Days	Commute Time (min)
2.7	Fri.	25
4.1	Mon.	33
1.0	Sun.	15
5.2	Tue.	45
2.8	Sat.	22

Table 1.1: Commute Time Data

In order to implement the linear regression, we first have to convert the non-numeric variables $\{Mon, \cdots, Sun\}$ to numbers. Because in linear regression model variables have arithmetic meaning(for example $0 <$

1), we cannot simply convert these nominal(unordered) variables to integers $\{0, \cdots, 6\}$. However, we can introduce dummy variables for representing these features. The most common method is One-hot Encoding. For example, in this case we can set weekdays to 0 and weekends to 1. Then we have the reorganized data $< (\boldsymbol{x}_i, y_i) >_{i=1}^{5}$, where $\boldsymbol{x} \in \mathbb{R}^2, y \in \mathbb{R}$. For every data point $\boldsymbol{x}_i = [x_{i1}, x_{i2}]$, $x_{i1} \in \mathbb{R}(distance), x_{i2} \in \{0, 1\}(weekdays\ or\ weekends)$. Now the linear regression model for commute time is given in the equation 1.3.

$$y = \boldsymbol{x} \cdot \boldsymbol{w} + \epsilon = y = w_0 + x_1 w_1 + x_2 w_2 + \epsilon \tag{1.3}$$

1.1.2 Learning Linear Models

We have the data $< (\boldsymbol{x}_i, y_i) >_{i=1}^{N}$ where $\boldsymbol{x_i} \in \mathbb{R}^D$ and $\boldsymbol{y_i} \in \mathbb{R}$. Now we have to estimate the model parameter $\boldsymbol{W}$ where $\boldsymbol{W} \in \mathbb{R}^D$ to build our linear regression model. This falls into two phases: training phase and testing/deployment phase. The training phase is to learn or estimate the parameter $\boldsymbol{W}$ from the data, which is shown in figure 1.1.

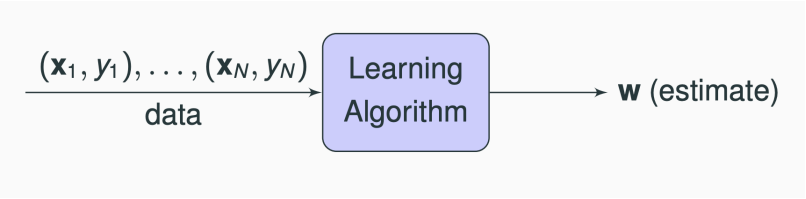

Figure 1.1: Training Phase

The testing phase is to predict the output $\hat{y}_{new}$ given a new input $\boldsymbol{x}_{new}$.

$$\hat{y} = \boldsymbol{x}_{new} \cdot \boldsymbol{w} \tag{1.4}$$

Obviously, we care about how accurate the prediction is. In another word, we care about how different the prediction $\hat{y}_{new}$ is from the actual observation y_{new}. Thus, we should keep some data aside for testing before deploying the model.

1.1.3 Least Squares Objective Function

We have the dataset $< (\boldsymbol{x}_i, y_i) >_{i=1}^{N}$ where $\boldsymbol{x_i} \in \mathbb{R}^D$ and $\boldsymbol{y_i} \in \mathbb{R}$.

The prediction is given by:

$$\hat{y}_i = \boldsymbol{x}_i \cdot \boldsymbol{w} \tag{1.5}$$

To simplify the illustration, we here only consider the input vector of 1 dimension. For example, we predict the commute time using only the

distance from the city center.

$$\hat{y}_i = \boldsymbol{x}_i \cdot \boldsymbol{w} = w_0 + x_1 w_1 \tag{1.6}$$

Where x_1 is the distance from the city center.

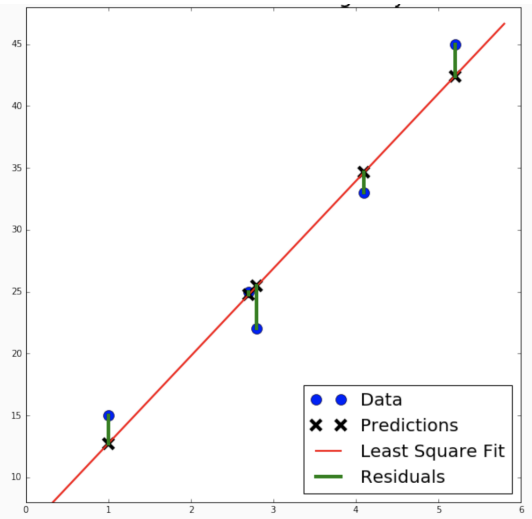

Figure 1.2: Prediction Commute Time with Distance

The performance of the model can be evaluated by the residual between the prediction and the actual observation given an input $\boldsymbol{x_i}$.

Obviously, we want to minimize the residual so that the prediction is closest possible to the actual observation. Our goal is to minimize the sum of the residuals over all data in the training dataset.

However, there exists a problem of positive and negative residual canceling each other out. Thus we often utilize the residual sum of squares(RSS) to avoid it. The residual sum of squares here is our Loss Function.

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2N}\sum_{i=1}^{N}(\hat{y} - y_i)^2 = \frac{1}{2N}\sum_{i=1}^{N}(\boldsymbol{x}_i \cdot \boldsymbol{w} - y_i)^2 \tag{1.7}$$

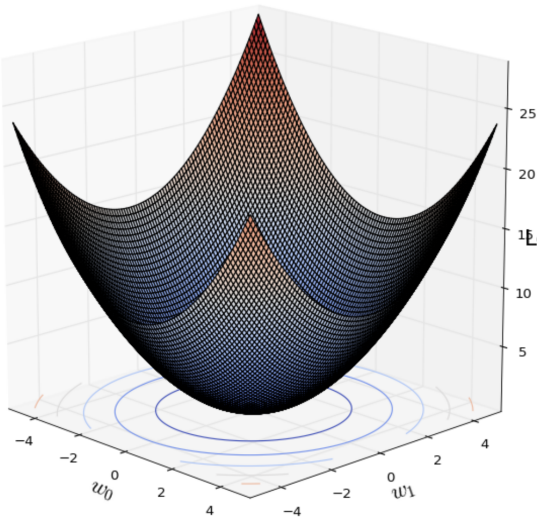

Figure 1.3: Loss Function

Thus, our goal here it to minimize the Loss Function.

We can easily obtain the solution for $\boldsymbol{W}$ which minimizes the loss function by setting the partial derivative to 0 and solving the resulting system.

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial w_0} = 2 \times \frac{1}{2N} \sum_{i=1}^{N} w_0 + x_{i1} w_1 - y_i = 0 \\ \frac{\partial \mathcal{L}}{\partial w_1} = 2 \times \frac{1}{2N} \sum_{i=1}^{N} (w_0 + x_{i1} w_1 - y_i) x_{i1} = 0 \end{cases} \tag{1.8}$$

Simplify the equation above we obtain the Normal Equation.

$$\begin{cases} w_0 + w_1 \frac{\sum_i x_i}{N} = \frac{\sum_i y_i}{N} \\ w_0 \frac{\sum_i x_i}{N} + w_1 \frac{\sum_i x_i^2}{N} = \frac{\sum_i x_i y_i}{N} \end{cases} \tag{1.9}$$

Solving for the Normal Equation we obtain:

$$\begin{cases} w_0 = \bar{y} - w_1 \bar{x} \\ w_1 = \frac{\widehat{COV}(x,y)}{\widehat{VAR}(x)} \end{cases} \tag{1.10}$$

Where $\bar{x} = \frac{\sum_i x_i}{N}$, $\bar{y} = \frac{\sum_i y_i}{N}$,$\widehat{VAR}(x) = \frac{\sum_i x_i^2}{N} - \bar{x}^2$and $\widehat{COV}(x,y) = \frac{\sum_i x_i y_i}{N} - \bar{x}\bar{y}$.

In terms of more general case, we can express everything in matrix notation. We have the data $< (\boldsymbol{x}_i, y_i) >_{i=1}^{N}$, where $\boldsymbol{x} \in \mathbb{R}^D, y \in \mathbb{R}$. Then we have the data augmented to 1:$< (\boldsymbol{x}_i, y_i) >_{i=1}^{N}$, where $\boldsymbol{x} \in \mathbb{R}^{D+1}, y \in \mathbb{R}$.

The prediction is given in the following equation.

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} \tag{1.11}$$

Here we have $\hat{\boldsymbol{y}} \in \mathbb{R}^{N\times 1}$, $\boldsymbol{X} \in \mathbb{R}^{N\times(D+1)}$ and $\boldsymbol{w} \in \mathbb{R}^{(D+1)\times 1}$.
Then the Loss Function is given by:

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}) =& \frac{1}{2N}\sum_{i=1}^{N}(\boldsymbol{x}_i^T \cdot \boldsymbol{w} - y_i)^2 = \frac{1}{2N}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \\ =& \frac{1}{2N}(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w} - \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{y}) \\ =& \frac{1}{2N}(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w} - (\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w})^T - \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{y}) \\ =& \frac{1}{2N}(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{y})\end{aligned} \tag{1.12}$$

The gradient of the Loss Function $\nabla_{\boldsymbol{w}}\mathcal{L}$ is given by:

$$\begin{aligned}\nabla_{\boldsymbol{w}}\mathcal{L} =& \frac{1}{2N}\frac{\partial(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w} - 2\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{y}^T\boldsymbol{y})}{\partial\boldsymbol{w}} \\ =& \frac{1}{2N}[\frac{\partial(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w})}{\partial\boldsymbol{w}} - 2\frac{\partial(\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w})}{\partial\boldsymbol{w}} + \frac{\partial(\boldsymbol{y}^T\boldsymbol{y})}{\partial\boldsymbol{w}}] \\ =& \frac{1}{2N}(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} + (\boldsymbol{X}^T\boldsymbol{X})^T\boldsymbol{w} - 2\boldsymbol{X}^T\boldsymbol{y}) \\ =& \frac{1}{N}(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y})\end{aligned} \tag{1.13}$$

Here we use the rules of Linear Form Expressions(i.e. $\nabla_{\boldsymbol{w}}(\boldsymbol{C}^T\boldsymbol{w}) = \boldsymbol{C}$) , Quadratic Expressions(i.e. $\nabla_{\boldsymbol{w}}(\boldsymbol{w}^T\boldsymbol{A}\boldsymbol{w}) = \boldsymbol{A}\boldsymbol{w} + \boldsymbol{A}^T\boldsymbol{w}$) and that the transpose of a scalar is itself(i.e. $(\boldsymbol{X}^T\boldsymbol{X})^T = \boldsymbol{X}^T\boldsymbol{X}$).

By setting the gradient to 0,

$$\nabla_{\boldsymbol{w}}\mathcal{L} = \frac{1}{N}(\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y}) = 0 \tag{1.14}$$

We obtain:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{1.15}$$

Thus the prediction $\hat{\boldsymbol{y}}$ is given by:

$$\hat{\boldsymbol{y}} = \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{1.16}$$

$\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ is called the hat matrix.

In terms of the computational complexity, the computational complexity of $\boldsymbol{Z} = \boldsymbol{X}^T\boldsymbol{X}$, $\boldsymbol{Z}^{-1}$, $\boldsymbol{A} = \boldsymbol{X}^T\boldsymbol{y}$ and $\boldsymbol{w} = \boldsymbol{Z}^{-1}\boldsymbol{A}$ is $O(D^2N)$, $O(D^3)$, $O(N)$ and $O(D^2)$ each respectively . Thus the overall computational complexity is $O(D^2N + D^3)$. Moreover, if $\boldsymbol{X}$ is defined by a join of several relations, the number of rows N maybe exponential in the number of relations.

$$N = O(M^{numberrelations}) \tag{1.17}$$

Thus $\boldsymbol{X}$ is sparse and can be represented in $O(M)$ space losslessly for acyclic joins which are common in practice. Then $\boldsymbol{w}$ can be computed in $O(D^2M + D^3)$

There leaves only one question. When do we expect $\boldsymbol{X}^T\boldsymbol{X}$ to be invertible?

The matrix $\boldsymbol{X}^T\boldsymbol{X} \in \mathbb{R}^{(D+1)\times(D+1)}$. $Rank(\boldsymbol{X}^T\boldsymbol{X}) = Rank(\boldsymbol{X}) \leq min\{D+1, N\}$. Thus it is invertible if $Rank(\boldsymbol{X}) = D + 1$.

Recall our toy example of predicting the computing time. If we use one-hot encoding for a feature like days. $\{x_{Mon}, \cdots, x_{Sun}\}$ stand for a 0-1 valued variables in the one-hot encoding. Thus we will always have $x_{Mon} + \cdots + x_{Sun} = 1$, which introduces a linear dependance in the columns of $\boldsymbol{X}$ reducing the rank. In this case, we cannot expect $\boldsymbol{X}^T\boldsymbol{X}$ to be invertible. We may have to drop some features to adjust the rank.

1.2 Maximum Likelihood

Recall what we have discussed in the previous section. Generally speaking ,we want to pick a model that is expected to fit our data well enough. So we choose a measure of performance of how well this model fits the data that makes "sense" and can be optimised. Then we can utilize a certain optimisation algorithm to obtain the model parameters. This is the "optimisation" view of machine learning.

In this section we will look at things in a different view - the "probabilistic" view of machine learning. In general, we want to pick a model for our data and explicitly formulate the deviation or uncertainty from the model by probability. We will use the notions from probability theory to define the suitability of various models. Then we can "find" the parameters or make predictions on unseen data utilizing these suitability criteria.

1.2.1 Maximum Likelihood Principle

Given observations $x_1, \cdots, x_N$ drawn independently from the same distribution(i.e. independently and identically distributed(i.i.d.)) p which has a parametric form, for example, the parameter vector $\boldsymbol{\theta}$. The likelihood of observing $x_1, \cdots, x_N$ equals to the probability of making these observations assuming they are generated according to p(i.e the joint probability distribution of the observations given $\boldsymbol{\theta}$: $p(x_1, \cdots, x_N|\boldsymbol{\theta})$).

The *Maximum Likelihood Principle* is to pick the parameter $\boldsymbol{\theta}$ that maximise the likelihood.

$$\boldsymbol{\theta}^* = \arg\max_{\boldsymbol{\theta}} p(x_1, \cdots, x_N|\boldsymbol{\theta}) \tag{1.18}$$

Since the observations are i.i.d.,

$$p(x_1, \cdots, x_N|\boldsymbol{\theta}) = \prod_{i=1}^{N} p(x_i|\boldsymbol{\theta}) \tag{1.19}$$

Let's look at an example here. The probability distribution of a discrete random variable X is given in Table 1.2.

X	0	1	2	3
p(X)	$\frac{2\theta}{3}$	$\frac{\theta}{3}$	$\frac{2(1-\theta)}{3}$	$\frac{1-\theta}{3}$

Table 1.2: Probability Distribution of X

If we have the observations $(3, 0, 2, 1, 3, 2, 1, 0, 2, 1)$. How can we compute the maximum likelihood estimate for θ?

The likelihood is given by:

$$p(x_1, \cdots, x_N|\boldsymbol{\theta}) = \prod_{i=1}^{10} p(x_i|\boldsymbol{\theta}) = (\frac{2\theta}{3})^2(\frac{\theta}{3})^3(\frac{2(1-\theta)}{3})^3(\frac{1-\theta}{3})^2 \tag{1.20}$$

To make computation simpler, we take the Log of the likelihood and take the derivative with respect to the parameter θ.

$$\log L(\theta) = 5\log\theta + 5\log(1-\theta) - 10\log 3 \tag{1.21}$$

In order to maximize the likelihood, we set the derivative to 0 and obtain:

$$\theta = \frac{1}{2} \tag{1.22}$$

1.2.2 Probabilistic Formulation of Linear Regression via Maximum Likelihood

Recall the linear model,

$$y = \boldsymbol{w}\boldsymbol{x} + \epsilon \tag{1.23}$$

We can see it as model y given $\boldsymbol{x}, \boldsymbol{W}$ as a random variable with mean $\boldsymbol{W}^T\boldsymbol{x}$.

$$\mathbb{E}[y|\boldsymbol{x}, \boldsymbol{w}] = \boldsymbol{w}^T\boldsymbol{x} \tag{1.24}$$

We will be specific in choosing the distribution of y given $\boldsymbol{x}, \boldsymbol{w}$.

If we assume that given $\boldsymbol{x}, \boldsymbol{w}$, y is normal with mean $\boldsymbol{w}^T\boldsymbol{x}$ and variance σ^2, we can obtain,

$$p(y|\boldsymbol{x}, \boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}^T\boldsymbol{x}, \sigma^2) = \boldsymbol{w}^T\boldsymbol{x} + \mathcal{N}(0, \sigma^2) \tag{1.25}$$

Alternatively, we may view this model as linear regression with Gaussian noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

Since the input $\boldsymbol{X} = [\boldsymbol{x_1}, \cdots, \boldsymbol{x_N}]$ is fixed, we do not need to model a distribution over $\boldsymbol{X}$.

Now we have the observed data $(\boldsymbol{X}, \boldsymbol{y})$ made of $(\boldsymbol{x_1}, y_1), \cdots, (\boldsymbol{x_N}, y_N)$. If we make an assumption that all data share the same variance σ, the likelihood of the observing data for model parameters $\boldsymbol{w}, \sigma$ is given as,

$$p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \sigma) = p(y_1, \cdots, y_N|\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{w}, \sigma) = \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i, \boldsymbol{w}, \sigma) \quad (1.26)$$

According to the model $y_i \sim \mathcal{N}(\boldsymbol{w}^T \boldsymbol{x}_i, \sigma^2)$,

$$p(y_1, \cdots, y_N|\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{w}, \sigma) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2}{2\sigma^2}} \quad (1.27)$$

Thus,

$$p(y_1, \cdots, y_N|\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{w}, \sigma) = (\frac{1}{\sqrt{2\pi\sigma^2}})^{\frac{N}{2}} e^{-\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2} \quad (1.28)$$

With matrix notations,

$$p(y_1, \cdots, y_N|\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{w}, \sigma) = (\frac{1}{\sqrt{2\pi\sigma^2}})^{\frac{N}{2}} e^{-\frac{1}{2\sigma^2} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})} \quad (1.29)$$

Every parameter set$(\boldsymbol{w}, \sigma^2)$ defines a probability distribution over observed data. Our goal here is to find parameters $\boldsymbol{w}$ and σ^2 that maximise the likelihood of observing the data.

To simplify the computation, we first take the negative log-likelihood,

$$NLL(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, \sigma) = \frac{N}{2} \log 2\pi\sigma^2 + \frac{1}{2\sigma^2} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \quad (1.30)$$

Then our goal is changed to minimise the negative log-likelihood.

We first find $\boldsymbol{W}$ that minimise the negative log-likelihood. Recall the objective function we use for the least squares estimate in the previous section,

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2N} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \quad (1.31)$$

For minimising with respect to $\boldsymbol{w}$, the two objectives are the same up to a contant additive and multiplicative factor!

Thus, the solution $\boldsymbol{w}_{ML}$ to find the maximum likelihood estimator is the same as the least squares estimator:

$$\boldsymbol{w}_{ML} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y} \quad (1.32)$$

Then the maximum likelihood estimate for σ is obtained by setting the derivative of negative log-likelihood with respect to σ^2, given by:

$$\sigma^2_{ML} = \frac{1}{N}(\boldsymbol{X}\boldsymbol{w}_{ML} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w}_{ML} - \boldsymbol{y}) \tag{1.33}$$

Thus given training data $(\boldsymbol{x_1}, y_1), \cdots, (\boldsymbol{x_N}, y_N)$, we can obtain the maximum likelihood estimate parameters $\boldsymbol{w}_{ML}$ and σ^2_{ML}.

On a new point $\boldsymbol{x_new}$, we can make prediction,

$$\hat{y} = \boldsymbol{w}_{ML}\boldsymbol{x}_{new} \tag{1.34}$$

and give confidence intervals,

$$y_{new} \sim \hat{y} + \mathcal{N}(0, \sigma^2) \tag{1.35}$$

as is shown in figure 1.4.

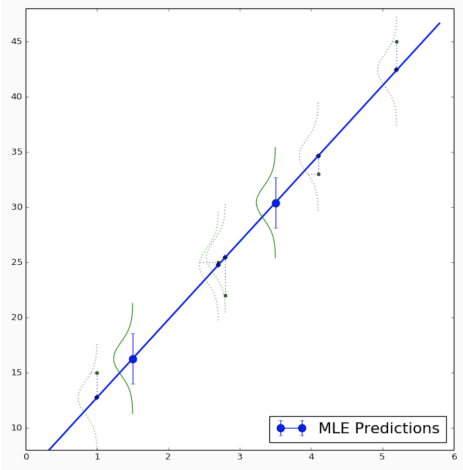

Figure 1.4: Maximum Likelihood Estimator

If the data has outliers, we can model the noise using a distribution that has heavier tails,

$$\epsilon \sim \mathcal{L}(0, b) \tag{1.36}$$

where $L(\mu, b)$ is Laplace Distribution whose density function is given by,

$$p(x) = \frac{1}{2b}e^{-\frac{|x-\mu|}{b}} \tag{1.37}$$

In this case, the likelihood of observing the data in terms of model parameters $\boldsymbol{w}$ and b is given by,

$$p(y_1, \cdots, y_N | \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{w}, b) = \prod_{i=1}^{N} \frac{1}{2b} e^{-\frac{|y_i - \boldsymbol{w}^T \boldsymbol{x}_i|}{b}} = \frac{1}{(2b)^N} e^{-\frac{1}{b}\sum_{i=1}^{N}(|y_i - \boldsymbol{w}^T \boldsymbol{x}_i|} \tag{1.38}$$

The negative log-likelihood is given by,

$$NLL(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{w}, b) = \frac{1}{b}\sum_{i=1}^{N}(|y_i - \boldsymbol{w}^T \boldsymbol{x}_i| + N \log(2b) \tag{1.39}$$

Maximum likelihood estimate can be obtained by minimising the sum of the absolute values of the residuals, which is more robust to outliers in fitting a linear model. However since absolute value function is not differentiable everywhere, there is no closed-form solution for $\boldsymbol{w}_{ML}$ and b_{ML}. It is also a non-linear objective function which is very hard to optimise. The good news is that it can be transformed into a linear objective subject to linear constraints which is a convex optimisation problem and has a unique solution. The details are discussed in Murphy's book at page 226.

1.3 Basis Expansion

1.3.1 Polynomial Basis Expansion

Consider the following one-dimensional data points generated by a quadratic function(red curve) with some noise shown in Figure 1.5.

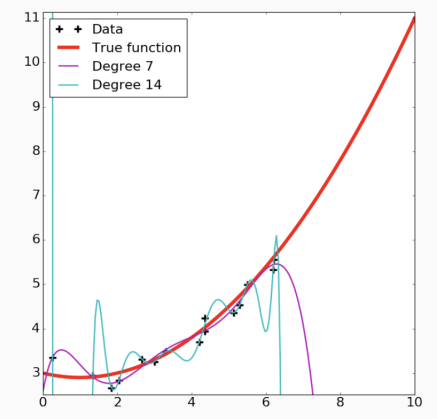

Figure 1.5: Polynomial Expansion

The linear model fitting the data points can be given by the dot product of the weight vector $\boldsymbol{w} = [w_0, w_1]^T$ and the feature vector $\boldsymbol{\psi}(x) = [1, x]^T$

which was mapped from x by the mapping function $\boldsymbol{\psi}(x)$.

$$\hat{y}_i = \boldsymbol{w}\boldsymbol{\psi}(x_i) = w_0 + w_1 x_i + \epsilon \tag{1.40}$$

Since the data points are generated by a quadratic function, the linear model with a polynomial degree of 1 is not able to fit the data very well.

Thus, we can utilize a mapping function with a higher polynomial degree of d to do a d-degree polynomial basis expansion on the input feature x. The feature vector $\boldsymbol{\psi}(x)$ now is given by,

$$\boldsymbol{\psi}(x) = [1, x, x^2, \cdots, x^d] \tag{1.41}$$

Where $d \in \mathbb{N}$.

Thus, the linear model with a polynomial degree of d is given by,

$$\hat{y}_i = \boldsymbol{w}\boldsymbol{\psi}(x) = [w_0, \cdots, w_d][1, x, x^2, \cdots, x^d]^T + \epsilon = \sum_{i=0}^{d} w_d x_i^d + \epsilon \tag{1.42}$$

The purple curve and the green curve in Figure 1.5 shows the linear model with a polynomial degree of 7 and 14 each respectively. As is shown in the figure, the linear models with higher polynomial degrees fit the data points better yet introduce a problem of overfitting. Using more data can avoid overfitting.

The polynomial expansion can also be performed in higher dimensions. In another word, we can also perform polynomial expansion on multi-dimensional input data. In this case, we are still fitting lineal model, but using more features. Using degree d polynomials in k dimensions results in $O(k^d)$ features.

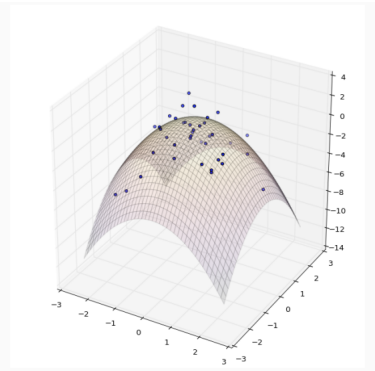

Figure 1.6: Quadratic Model $\boldsymbol{\psi(x)} = [1, x_1, x_2, x_1^2, x_2^2, x_1 x_2]$

1.4 Basis Expansion with Kernels

Recall the linear regression model with polynomial basis expansion. We want to minimize the loss function given by,

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2N}(\boldsymbol{\phi}(\boldsymbol{X})\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{\phi}(\boldsymbol{X})\boldsymbol{w} - \boldsymbol{y}) \tag{1.43}$$

Where $\boldsymbol{\phi}(\boldsymbol{X}) \in \mathbb{R}^{N \times O(k^d)}$ is the k degree polynomial expansion of $\boldsymbol{X}$.

Setting $\nabla_{\boldsymbol{w}}\mathcal{L}$ to 0 we can obtain the weight minimizing the loss function given by,

$$\boldsymbol{w} = (\boldsymbol{\phi}(\boldsymbol{X})^T\boldsymbol{\phi}(\boldsymbol{X}))^{-1}\boldsymbol{\phi}(\boldsymbol{X})^T\boldsymbol{y} \tag{1.44}$$

If we consider the basis expansion of just one input $\boldsymbol{\phi}(\boldsymbol{x}) \in \mathbb{R}^{1 \times O(k^d)}$. The computational complexity of $\boldsymbol{\phi}(\boldsymbol{x})^T\boldsymbol{\phi}(\boldsymbol{x})$ is already $O(k^d)$, let alone we have N inputs $\boldsymbol{\phi}(\boldsymbol{X}) \in \mathbb{R}^{N \times O(k^d)}$ and have to compute the inverse matrix $(\boldsymbol{\phi}(\boldsymbol{X})^T\boldsymbol{\phi}(\boldsymbol{X}))^{-1}$. In a word, this is insanely computationally-expensive.

However, we can use kernel trick to compute $\boldsymbol{\phi}(\boldsymbol{X})^T\boldsymbol{\phi}(\boldsymbol{X})$ to reduce the computational complexity.

For a certain expansion ϕ, a kernel computes the dot product,

$$\kappa(\boldsymbol{x'}, \boldsymbol{x}) : \chi \times \chi \to \mathbb{R} = \phi(\boldsymbol{x'})\phi(\boldsymbol{x}) \tag{1.45}$$

1.4.1 Polynomial Kernel

Polynomial kernel is given as:

$$\kappa_{poly}(\boldsymbol{x'}, \boldsymbol{x}) = (\boldsymbol{x'}\boldsymbol{x} + \theta)^d \tag{1.46}$$

Assume we want to find expansion function for kernel $\kappa_{poly}(\boldsymbol{x'}, \boldsymbol{x}) = (\boldsymbol{x'}\boldsymbol{x})^d$, where $\boldsymbol{x'}, \boldsymbol{x} \in \mathbb{R}$.

Recall the Multinomial theorem,

$$(y_1 + \cdots + y_k)^d = \sum_{n_i \geq 0, \sum_i n_i = d} C(d; n_1 + \cdots + n_k) \prod_{j=1}^{k} y_j^{n_j} \tag{1.47}$$

Where $C(d; n_1 + \cdots + n_k)$ is the multinomial coefficient,

$$C(d; n_1 + \cdots + n_k) = \frac{d!}{n_1! \cdots n_k!} \tag{1.48}$$

C can be viewed as the number of ways to distribute d balls into k bins, where the j^{th} bin holds $n_j \geq 0$ balls.

Now assume $y_i = x_i x_i'$, then Equation 1.45 is given as,

$$(\boldsymbol{x'}, \boldsymbol{x})^d = \sum_{n_i \geq 0, \sum_i n_i = d} \sqrt{C(d; n_1 + \cdots + n_k)} \prod_{j=1}^{k} x_j^{n_j} \sqrt{C(d; n_1 + \cdots + n_k)} \prod_{j=1}^{k} x_j'^{n_j} \tag{1.49}$$

Since the dimension of $\phi_{poly}(\boldsymbol{x})$ and $\phi_{poly}(\boldsymbol{x'})$ is $O(k^d)$, the complexity of computing $\kappa_{poly}(\boldsymbol{x'}, \boldsymbol{x})$ using expansion functions is $O(k^d)$. Alternatively, if we know that the dot product of expansion functions $\phi(\boldsymbol{x})\phi(\boldsymbol{x'})$ equals to $(\boldsymbol{x'}, \boldsymbol{x})^d$, the computational complexity can be reduced to $O(k \log d)$ using kernel $\kappa_{poly}(\boldsymbol{x'}, \boldsymbol{x})$, which is more computationally-efficient. This is called the kernel trick.

1.4.2 Radial Basis Function Kernel

A Radial Basis Function Kernel is defined as,

$$\kappa_{RBF}(\boldsymbol{x}, \boldsymbol{x'}) = \frac{1}{e^{\gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2}} \in (0, 1] \tag{1.50}$$

Where the hyperparameter γ is the width and $\boldsymbol{x'}$ the center of the distribution.

RBF acts as a similarity measure between $\boldsymbol{x}$ and $\boldsymbol{x'}$.

When $\boldsymbol{x}$ and $\boldsymbol{x'}$ are far apart(i.e. large squared Euclidian distance) or γ is very large,

$$\kappa_{RBF}(\boldsymbol{x}, \boldsymbol{x'}) \to 0, \ \gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2 \to \infty \tag{1.51}$$

When $\boldsymbol{x}$ and $\boldsymbol{x'}$ are close(i.e. small squared Euclidian distance) or γ is very small,

$$\kappa_{RBF}(\boldsymbol{x}, \boldsymbol{x'}) \to 1, \ \gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2 \to 0 \tag{1.52}$$

The question now is to find ϕ_{RBF} such that,

$$\kappa_{RBF}(\boldsymbol{x'}, \boldsymbol{x}) = \frac{1}{e^{\gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2}} = \phi_{RBF}(\boldsymbol{x})\phi_{RBF}(\boldsymbol{x'}) \tag{1.53}$$

The Taylor expansion of $e^{\boldsymbol{x}\boldsymbol{x'}}$ is given by,

$$e^{\boldsymbol{x}\boldsymbol{x'}} = \sum_{k=0}^{\infty} \frac{1}{k!} (\boldsymbol{x}\boldsymbol{x'})^k = \phi_{poly}(\boldsymbol{x})\phi_{poly}(\boldsymbol{x'}) \tag{1.54}$$

Without loss of generality, we assume $\gamma = 1$, then,

$$e^{\gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2} = e^{-(\|\boldsymbol{x}^2\| + \|\boldsymbol{x'}^2\| - 2\boldsymbol{x}\boldsymbol{x'})} = \sum_{k=0}^{\infty} (\sqrt{\frac{1}{k!}} (e^{-\|\boldsymbol{x}^2\|}) \phi_{poly}(\boldsymbol{x}) (\sqrt{\frac{1}{k!}} (e^{-\|\boldsymbol{x'}^2\|}) \phi_{poly}(\boldsymbol{x'})$$

(1.55)

Thus, we can see that $\phi_{RBF} : \mathbb{R}^d \to \mathbb{R}^\infty$ projects vectors into an infinite dimensional space. Therefore, it is not feasible to compute $\kappa_{RBF}(\boldsymbol{x'}, \boldsymbol{x})$ using ϕ_{RBF}.

1.4.3 Liner Model with RBF Kernel

Recall RBF kernel:

$$\kappa_{RBF}(\boldsymbol{x'}, \boldsymbol{x}) == \frac{1}{e^{\gamma \|\boldsymbol{x} - \boldsymbol{x'}\|^2}} \tag{1.56}$$

If we choose different centers $\mu_1, \cdots, \mu_M$, we can construct a feature map $\psi(\boldsymbol{x}) = [1, \kappa_{RBF}(\mu_1, \boldsymbol{x}), \cdots, \kappa_{RBF}(\mu_M, \boldsymbol{x})]^T$. Then the linear model is given as,

$$\hat{y} = \boldsymbol{w}\boldsymbol{\psi}(\boldsymbol{x}) + \epsilon \tag{1.57}$$

Figure 1.7 shows liner model with RBF kernel when input data is one-dimensional.

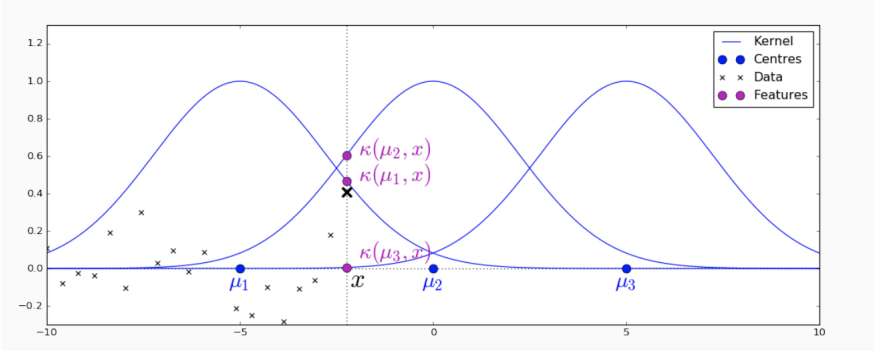

Figure 1.7: Liner Model with RBF Kernel

For hyperparameter μ, a reasonable choice for centers are data points themselves.

For hyperparameter γ, overfitting or underfitting may occur if width parameter is not chosen properly. More specifically, overfitting occurs when the kernel is too narrow(i.e. γ is very large) and vice versa.

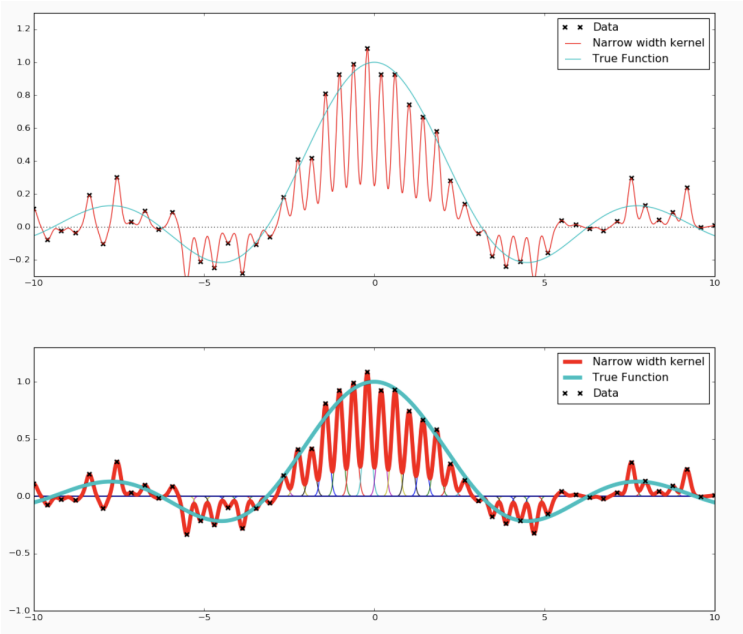

Figure 1.8: Narrow Kernel with Large γ

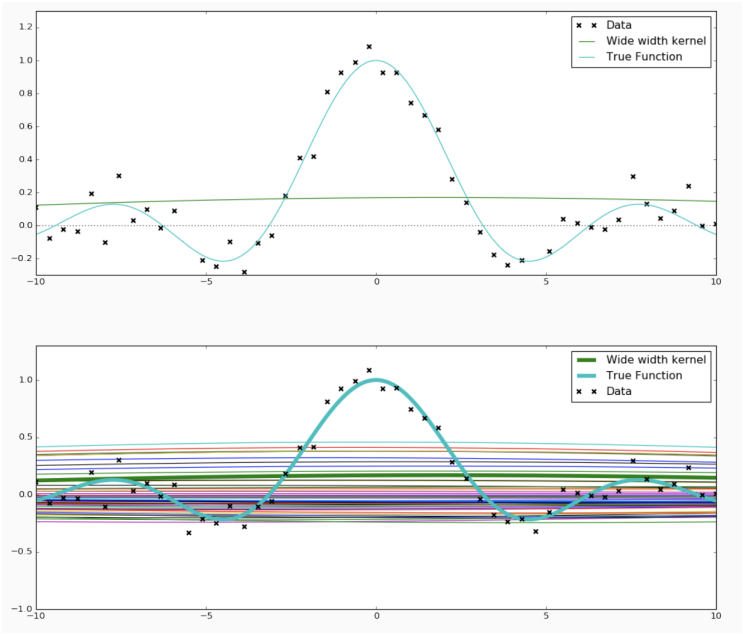

Figure 1.9: Wide Kernel with Small γ

Figure 1.10 shows the model when kernerl is well-chose.

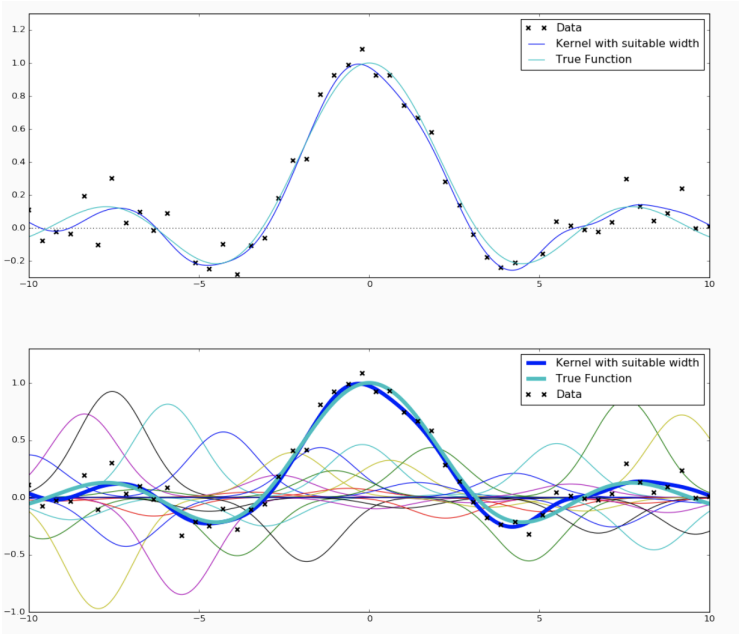

Figure 1.10: Wide Kernel with Well-chosen γ

Having more data can help reduce the overfitting if the kernel is too narrow.

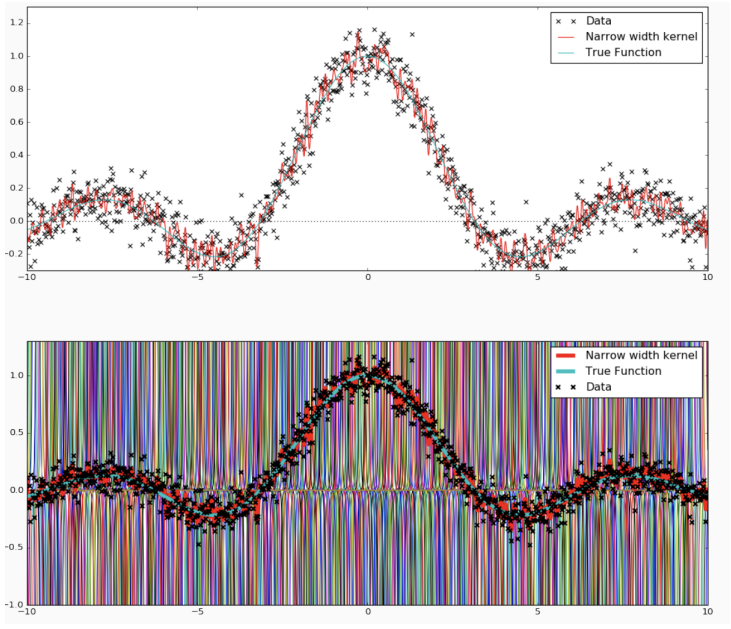

Figure 1.11: Narrow Kernel with Large γ, Big Data

However, extra data does not help reduce underfitting when the kernel

is too wide.

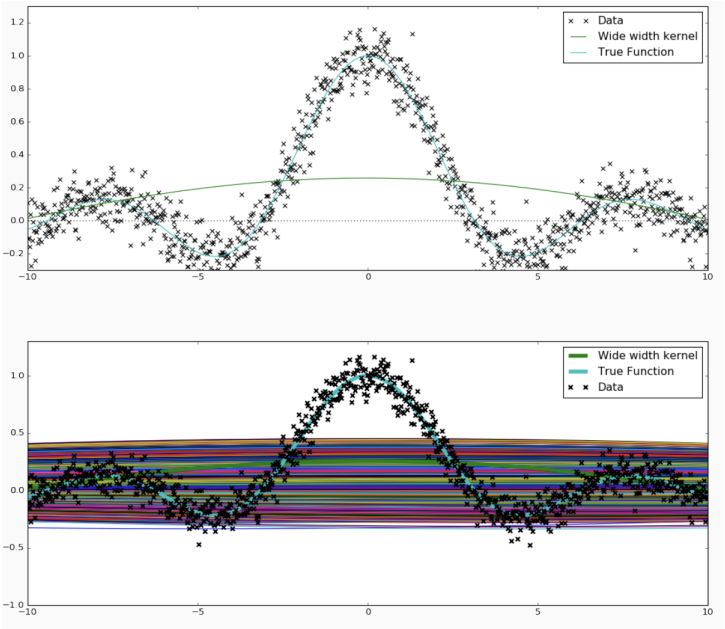

Figure 1.12: Wide Kernel with Small γ, Big Data

1.5 Bias-Variance Trade-off

1.5.1 Bias and Variance

The fundamental goal of machine learning is to generalize beyond the examples in the training data. Thus, test data is highly important.

Also, data alone is not enough. Every learner must embody knowledge/assumptions beyond the data. No learner can beat random guessing over all possible functions to be learned(Wolpert's "No Free Lunch"). Our hope is that the functions to learn in the real world are not drawn uniformly from the set of all mathematically possible function. The reasonable assumptions behind machine learning's success is that similar examples have similar classes and limited dependence or limited complexity.

Bias is the tendency to consistently learn the same wrong thing. For example, linear learner has high bias. When the frontier between two classes is not a hyperplane the learner is unable to induce it. However, decision trees do not have this problem because they can represent any Boolean function. Having high bias means underfitting.

Variance is the tendency to learn random things irrespective of the real signal. For example, decision trees suffer from high variance because when learned on different training sets generated by the same phenomenon they

are often very different, when in fact they should be the same. Having high variance means overfitting.

Thus, more powerful learners are not necessarily better than less powerful ones.

Generalization error is the combination of bias and variance,

$$\text{Generalization Error} = \text{Bias} + \text{Variance} \tag{1.58}$$

Figure 1.13 shows the example of high bias and high variance.

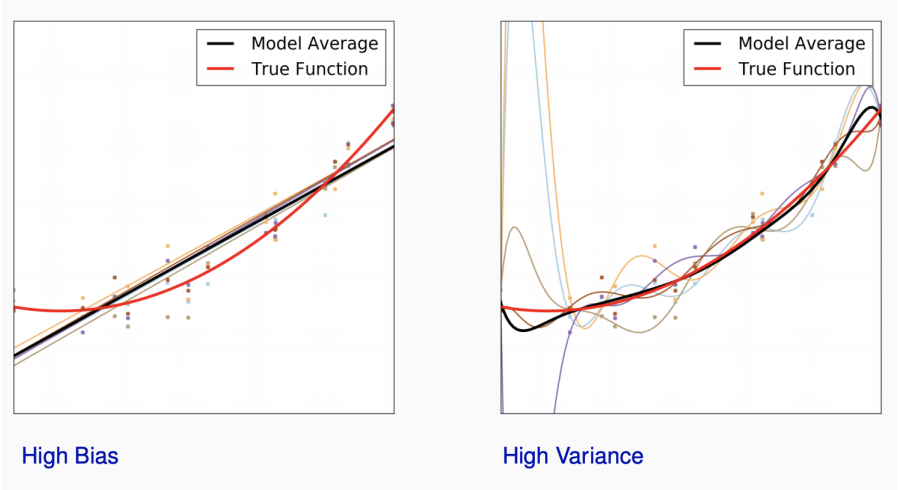

Figure 1.13: Experiment: 1-D Points Generated by a Quadratic Function

In fact, the term bias and variance are precisely defined in statistical notions. The description is in Section 5.4 in the Book *Deep Learning* by Goodfellow, Ian and et al.

1.5.2 Learning Curves

Suppose we have trained a model and used it to make predictions. In reality, theses prediction are usually very poor. The question is, how can we know whether we have high bias(underfitting) or high variance(overfitting) or neither. This is very important because it helps up to decide in which way we can improve our model. Should we add more features(e.g. higher degree polynomials, narrower kernels, etc.) to make the model more complex? Or should we simplify the model(e.g. lower degree polynomials, wider kernels, etc.) to reduce the number of parameters? Or if neither of the above works, should we try and obtain more data (which comes with higher computational and monetary cost)?

We can split the data into a training set and testing set. Then we train on increasing sizes of the data and plot the training error and test error as

a function of training data size. The curve obtained is called the learning curve.

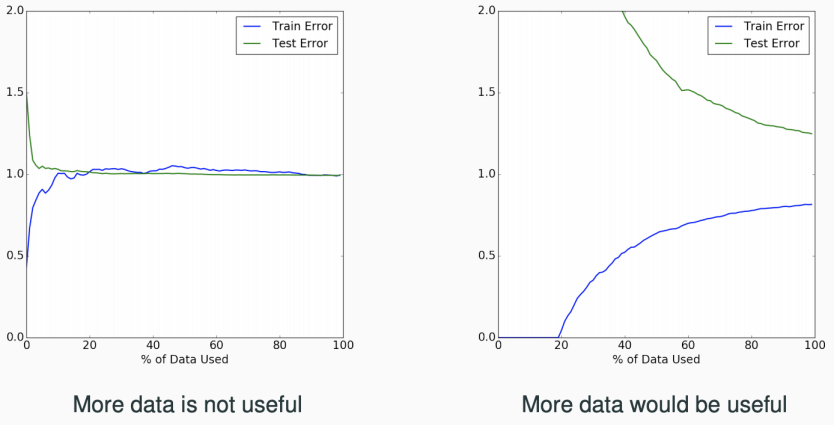

Figure 1.14: Learning Curve

1.5.3 Overfitting

When dealing with high-dimensional data(which may be caused by basis expansion) even for a linear model we have many parameters. For example with $D = 100$ input variables and using degree 10 polynomial expansion we have approximately 10^{20} parameters!

Suppose we have the data $\boldsymbol{X} \in \mathbb{R}^{100\times100}$ with every entry of $\boldsymbol{X}$ is drawn from $\mathcal{N}(0, 1)$. Let $y_i = x_{i,1} + \mathcal{N}(0, \sigma^2)$ for $\sigma = 0.2$. Thus, the label y_i only depends on the first feature.

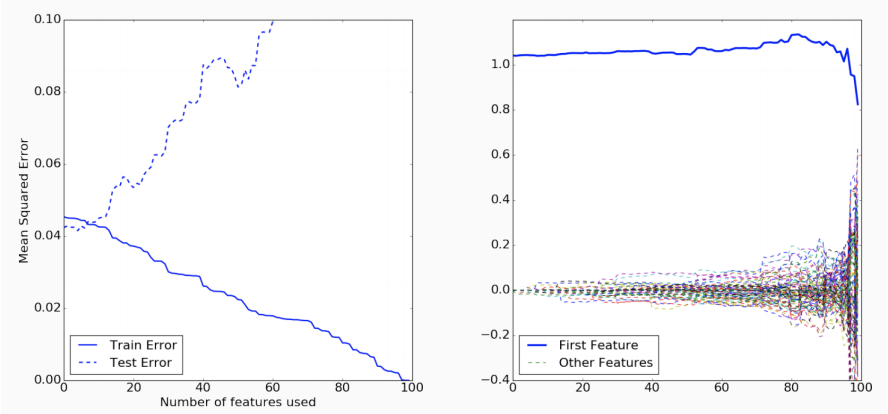

Figure 1.15: Overfitting

As is shown in the Figure 1.15, the more features are used during the training, the larger the testing error will be. In fact, we can see from the

left figure that when the number of features used is relatively small almost all the weight is on the first feature while when the the number of features used is relatively large the first feature becomes less relevant and the other features starts to fit the noise.

There are several ways to combat overfitting. For example, we can add regularization term to the evaluation function. Thus, it will penalise models with more structure and favor smaller models with less room to overfit. Also, we can do cross-validation or statistical significance test like chi-square before adding new structure to decide whether the prediction is different without this structure.

1.6 Regularization

Suppose we have data $\boldsymbol{X} \in \mathbb{R}^{N \times D}$, where $D \gg N$. Our idea to avoid overfitting is to add a penalty term for weights.

1.6.1 Ridge Regression

Recall the Least Square Estimate Objective,

$$\mathcal{L}(\boldsymbol{w}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \tag{1.59}$$

Ridge regression Objective is given by,

$$\mathcal{L}(\boldsymbol{w})_{Ridge} = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \sum_{i=1}^{D} w_i^2 \tag{1.60}$$

We add a penalty term $\lambda \sum_{i=1}^{D} w_i^2$ for all weights except w_0(since the effect of varying w_0 is output translation and not on model complexity) to control the model complexity.

However, since the translating and scaling inputs contribute to the model complexity, we should try and avoid dependence on scaling and translation of variable by standardize the input.

Feature standardization is a method used to normalize the range of independent variables or features of data. It is also known as data normalization and generally performed during the data preprocessing step.

Feature standardization makes the values of each feature in the data have zero-mean and unit-variance(i.e values drawn from $\mathcal{N}(0, 1)$).

The general method of calculation is to determine the distribution mean and standard deviation for each feature given by,

$$x' = \frac{x - \bar{x}}{\sigma} \tag{1.61}$$

Where $\bar{x}$ is the mean and σ is the standard deviation.

Suppose the data $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ with inputs standardized and output centered, the loss function of Ridge regression is given by,

$$\mathcal{L}(\boldsymbol{w})_{Ridge} = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^T\boldsymbol{w} \tag{1.62}$$

The gradient of the loss function is given by,

$$\nabla_{\boldsymbol{w}} \mathcal{L}_{Ridge} = 2((\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_D)\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y}) \tag{1.63}$$

Set the gradient to $\boldsymbol{0}$ and solve for $\boldsymbol{X}$, we obtain,

$$(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_D)\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{y} \tag{1.64}$$

Thus,

$$\boldsymbol{w}_{Ridge} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_D)^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{1.65}$$

Alternatively, we can consider the Ridge regression as a constrained optimization problem,

$$\arg\max_{\boldsymbol{w}} \ (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \text{ s.t. } \boldsymbol{w}^T\boldsymbol{w} \leq R$$

Then the former loss function $\mathcal{L}(\boldsymbol{w})_{Ridge}$ is the Lagrangian formulation of this optimization problem. Thus, the solution to Ridge regression can be viewed as a function of λ and R, as is shown in Figure 1.16.

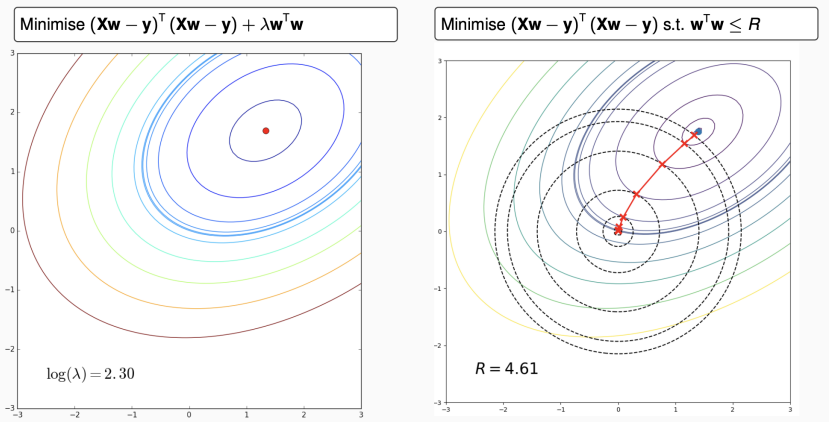

Figure 1.16: Ridge Regression v.s. Constrained Optimization

From Figure 1.16 we can see that decreasing λ equals to loose the constraint(i.e. increase R). If the constraint R is well-chosen, we will derive the same solution $\boldsymbol{w}$ for both problems.

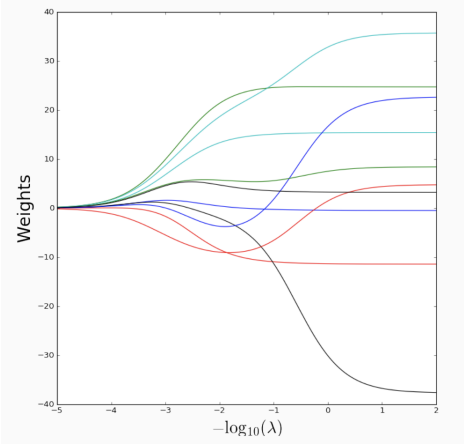

Figure 1.17: Magnitude of Weights v.s. $\log \lambda$

1.6.2 Lasso Regression

Recall the objective of Ridge regression,

$$\mathcal{L}(\boldsymbol{w})_{Ridge} = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^T\boldsymbol{w} \tag{1.66}$$

In Ridge regression, in addition to the residual sum of the squares we penalise the sum of squares of the weights. This is called l_2 regularization or weight-decay. By adding it, we will encourage fitting signal rather than just noise by penalising weights.

Similarly, we can do l_1 regularization and obtain the Lasso objective given by,

$$\mathcal{L}(\boldsymbol{w})_{Lasso} = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \|\boldsymbol{w}\|_1 \tag{1.67}$$

Thus,

$$\mathcal{L}(\boldsymbol{w})_{Lasso} = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \sum_1^D |w_i| \tag{1.68}$$

As with Ridge regression, there is a penalty on the weights. However, the absolute value function does not allow for a simple closed-form solution.

The Lasso regression can also be seen as a optimization problem,

$$\arg\max_{\boldsymbol{w}} \ (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \text{ s.t. } \sum_1^D |w_i| \leq R$$

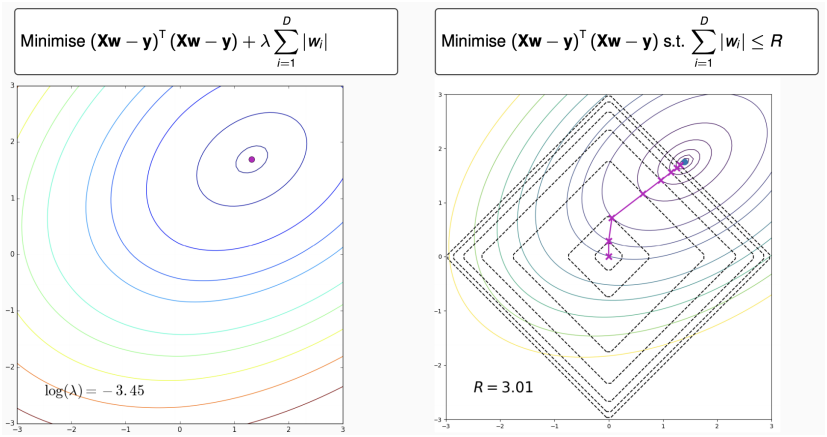

Figure 1.18: Lasso Regression v.s. Constrained Optimization

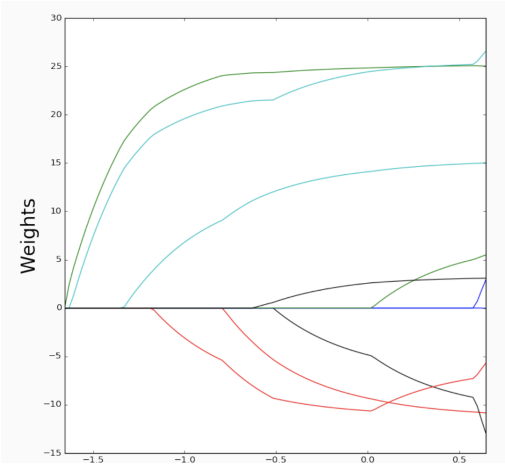

Figure 1.19: Magnitude of Weights v.s. $\log \lambda$

As is shown in Figure 1.19, in Lasso regression, weights are often exactly 0. Thus, Lasso gives a sparse model.

Recall our experiment. Suppose we have the data $\boldsymbol{X} \in \mathbb{R}^{100\times 100}$ with every entry of $\boldsymbol{X}$ is drawn from $\mathcal{N}(0,1)$. Let $y_i = x_{i,1} + \mathcal{N}(0,\sigma^2)$ for $\sigma = 0.2$. We have discussed the result of simple linear regression with no regularization before. Now let's look at some results of Ridge regression and Lasso regression.

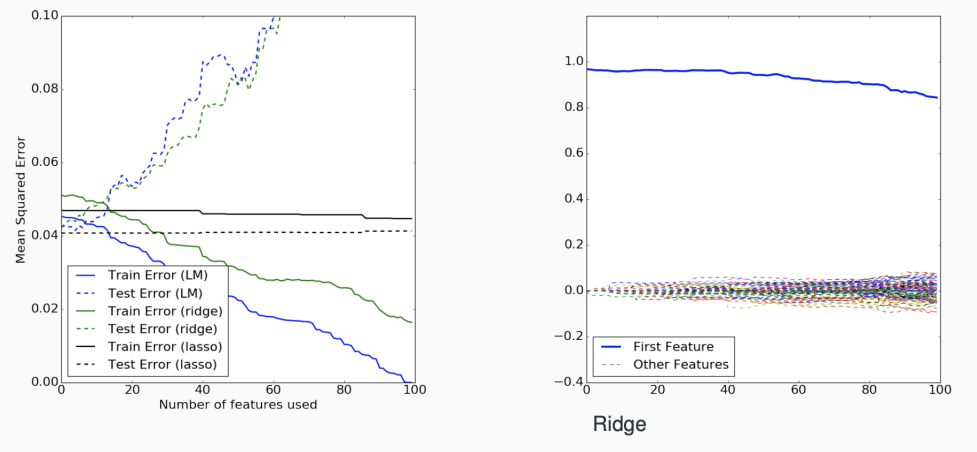

Figure 1.20: Ridge Regression Experiment

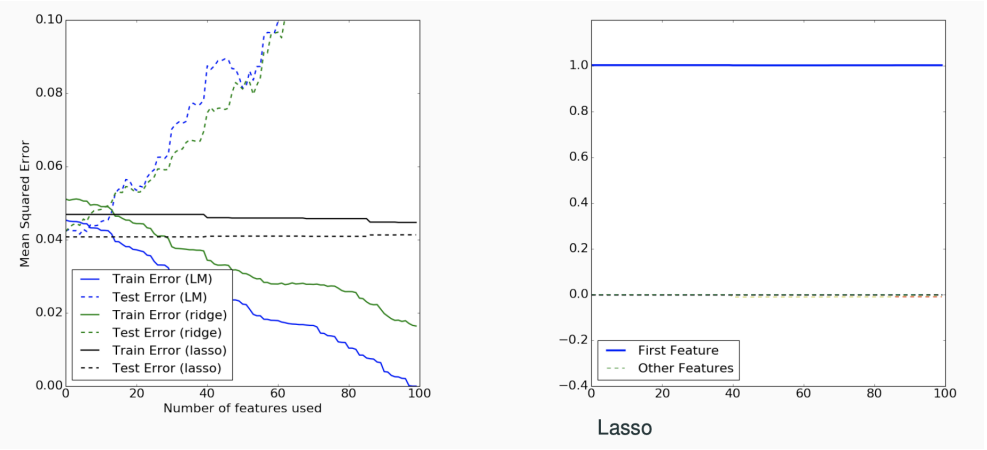

Figure 1.21: Lasso Regression Experiment

1.7 Validation

So far, we have discussed how to estimate the weight parameters $\boldsymbol{w}$. For Ridge and Lasso regression, we need to choose λ. If we performance basis expansion, we have to pick the width parameter γ and for polynomial expansion we have to choose polynomial degree d. For more complex models there may be even more hyper-parameters to choose.

Then the question is, how can we choose hyper-parameters so that our model works well. This can be done by validation.

1.7.1 Validation Error

If we have the data $\boldsymbol{X} \in \mathbb{R}^{D \times N}$, we can divide the dataset into three parts, namely training, validation and testing data.

Thus, for training stage, we can train the model using the training set and evaluate on the validation set. In this case, we will not only get a training error but also a validation error. If we have hyper-parameters to choose, for example λ, we can pick the value of λ that minimise the validation error.

Suppose we have the training data $\boldsymbol{X} \in \mathbb{R}^{2000 \times 2000}$. If we divide the data into training set and validation set(typically 80% for training and 20% for validation) and plot the training error and validation error v.s. λ for Lasso regression, we will get training and validation curves as is shown in Figure 1.22.

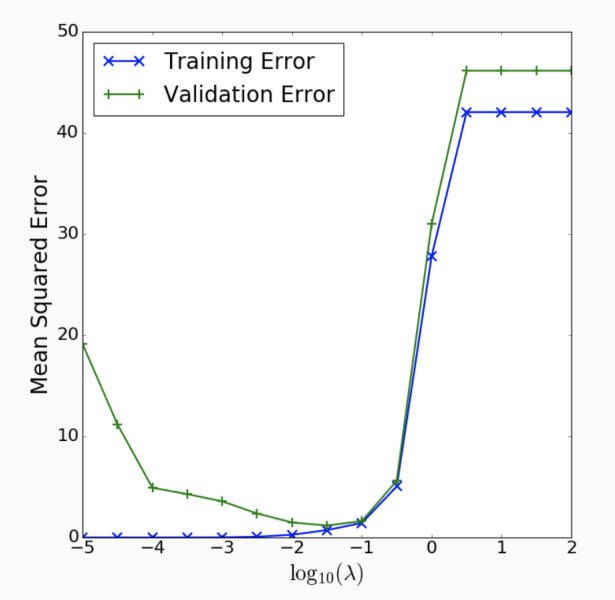

Figure 1.22: Training and Validation Curves

We can see from the U-shaped validation error curve that the left half

is overfitting(i.e. low training error and high validation error) and the right half is underfitting(i.e. high training error and high validation error). Thus, we should pick the hyper-parameter λ at the bottom of the curve.

What if we have many hyper-parameters?

The trivial approach is grid search. We first assume a small domain D_i for each hyper- parameter λ_i where $1 \leq \lambda_i \leq k$. Next we iterate over all possible combinations of hyper-parameter value $D_1 \times \cdots \times D_k$. Then we perform cross-validation for each such combination and pick the combination with the lowest validation search.

1.7.2 K-Fold Cross Validation

When the data is scarce, instead of splitting it into training and validation set, we divide the data into k folds(parts).

We use $k-1$ folds for training and 1 fold as validation. When k equals to the number of data points, it is called Leave and Out Cross Validation(LOOCV).

The validation error for fixed hyper-parameter values is the average of all runs.

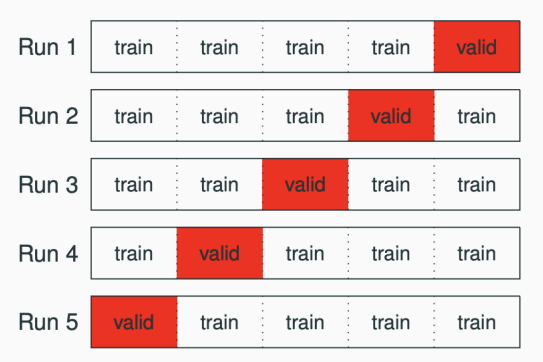

Figure 1.23: K-Fold Cross Validation

1.7.3 Kaggle Leaderboard Mechanism

Suppose we do all the things right. We train the model on the training set and choose hyper-parameters using k-fold validation. Then when we test on the test set(real world), the error is still somehow unacceptably high. This suggests that we might overfit on the validation set.

Kaggle competitions use a leaderboard mechanism, which is a classic hold-out method. The training set is publicly available. The hold-out set(which is split randomly into 30% hold-out labels and 70% test labels) is also publicly available but without the labels.

The valid submission is the list of predicted labels, one for each point in the testing set. These is also a score function such as misclassification rate to evaluate the submission.

The public ranking is given by the score on the hold-out labels(without the test labels) and the private ranking is given by the score on the test labels. The team can submit repeatedly for public ranking for quite some time before the final private ranking.

Suppose the task is to predict N binary labels, i.e. $\boldsymbol{y} \in \{0,1\}^N$. We can achieve a high score on public ranking by so-called Wacky Boosting Algorithm.

We firstly pick $\boldsymbol{y}_1, \cdots, \boldsymbol{y}_k \in \{0,1\}^N$ randomly. Next we select $\boldsymbol{Y} = \{\boldsymbol{y}_i | score(\boldsymbol{y}_i) \leq 0.5\}^N$. Then we simply output $\hat{\boldsymbol{y}} = majority(\boldsymbol{Y})$.

The problem is that there is a statistical dependence the hold-out data and the submission. Thus, submission may incorporate information about the hold-out labels realised through the leaderboard. Due to this feedback loop, the public score is no longer an unbiased estimator of the score. Eventually, the submission will overfit the hold-out set. This is an example of disconnect between theory f static data analysis and practice of interactive data analysis.

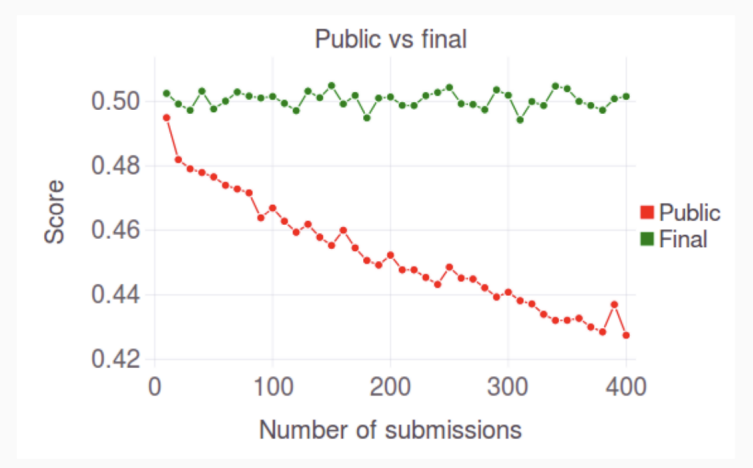

Figure 1.24: Public Score v.s. Final Score

How can we deal this problem? We can limit the rate of re-submissions, control the bit precision numbers or have winners determined on a separate test set.

1.8 Feature Selection

The goal of feature selection is to select features relevant to the model to learn given a set of possible features.

In practice, data usually contains redundant or irrelevant features which can be removed without causing information loss. Also, relevant features maybe redundant in the presence of another relevant feature(i.e. the two features are strongly correlated).

If we remove these redundant features, we can enhance the generalization by reducing overfitting, avoid the curse of dimensionality and shorten the training time. Also, simpler models are easier to interpret by the users.

1.8.1 Wrapper Methods

Wrapper method is to train a new model for each feature subset. The score is given by the count of number of mistakes on hold-out set. It is expensive yet usually provides the best performing feature set.

Let's look at an example of wrapper methods - Forward Step-wise Selection.

For n features, we ask the following sequence of n questions($1 < i \leq n$):

Q_1: What is the best 1-feature model? Let the chosen feature be f_1.

$\vdots$

Q_i: What is the best i-feature model that also has the previous selected features $f_1, \cdots, f_{i-1}$? Let the newly chosen feature be f_i.

The output is the best seen k-feature model, for $1 < k \leq n$.

The total number of models to train and test is $O(n^2)$ because at step i, it trains and tests $n - i + 1$ models. For linear regression, it is the same complexity as building one model.

1.8.2 Filter Methods

Filter methods use proxy measure as score instead of the actual test score. It exposes relationship between different features. Typical scores are mutual information and Pearson correlation coefficient. Filter methods are fast to compute yet results feature set not tunes to a specific type of models.

The mutual information for two random variables X and Y is given by,

$$I(X, Y) = \sum_x \sum_y p(X = x, Y = y) \log \frac{p(X = x, Y = y)}{p(X = x)p(Y = y)} \tag{1.69}$$

The probabilities can be empirically obtained from the training set. For example, $p(X = x)$ is the fraction of the number of samples with $X = x$ over the number of all samples. For variables with continuous domain, we first have to discretise their domains.

The feature selection approach via mutual information is to first compute the mutual information for each feature and the label(target) and then only keep the features that provide information about the output. In this approach, we get the ranking of features instead of finding the best features. The cutting-off point is obtained by using cross-validation.

Covariance for two random variables X and Y measures how the random variables change jointly,

$$cov(X, Y) = \mathbb{E}[(X - \mathbb{E}(X))(Y - \mathbb{E}(Y)))] \tag{1.70}$$

The feature selection can also be done via (Pearson) correlation coefficient.

The (Pearson) correlation coefficient normalize the covariance of variables X and Y to give a value between -1 and 1.

$$corr(X, Y) = \frac{cov(X, Y)}{\sqrt{cov(X, X)cov(Y, Y)}} \tag{1.71}$$

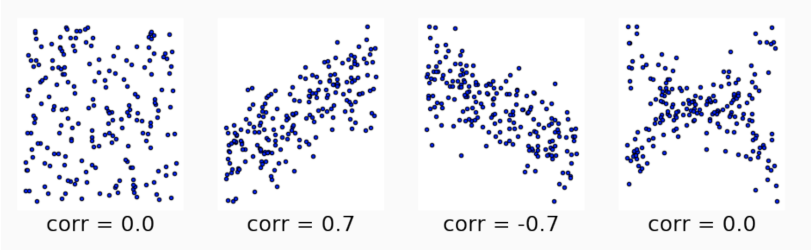

Figure 1.25: Correlation

Note that independent variables are uncorrelated, but uncorrelated variables are not necessarily independent.

2

Optimization

So far we have discussed machine learning problems with closed-form solutions such as minimization of least squares and ridge regression objectives.

However, this is not always the case. Most interesting learning problems do not admit closed-form solutions.

To solve the problems beyond the closed-form solutions, two approaches can be utilized. We can frame the objective of machine learning problem as a mathematical problem and when objectives can be formulated as convex optimization problem we can use existing black-box solver for such problems.

Also, we can use gradient-based optimization methods. These methods are not black-box and thus optimization hyper-parameters affect the performance.

In fact, most machine learning problems can be cast as optimization problems.

2.1 Convex Optimization

2.1.1 Convex Set

Definition 2.1 *A set $C \in \mathbb{R}^D$ is convex if for any given $\boldsymbol{x}, \boldsymbol{y} \in C$ and $\lambda \in [0, 1]$, it holds that $\lambda\boldsymbol{x} + (1 - \lambda)\boldsymbol{y} \in C$.*

Proposition 2.2 *Given convex sets $C_1, \cdot, C_n$, the set $\cap_{i=1}^{n} C_i$ is convex.*

Proposition 2.3 *For any L-norm $\|\cdot\|$, the set $B = \{\boldsymbol{x} \in \mathbb{R}^D : \|\boldsymbol{X}\| \leq 1\}$ is convex.*

Proof Take $\boldsymbol{x}, \boldsymbol{y} = \{\boldsymbol{x} \in \mathbb{R}^D : \|\boldsymbol{X}\| \leq 1\}$, to show $\lambda\boldsymbol{x} + (1 - \lambda)\boldsymbol{y} \in B$

$$\|\lambda\boldsymbol{x} + (1 - \lambda)\boldsymbol{y}\| \leq \lambda\|\boldsymbol{x}\| + (1 - \lambda)\|\boldsymbol{y}\| \leq 1 \tag{2.1}$$

□

Proposition 2.4 *Given $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$, the polyhedron $P = \{\boldsymbol{x} \in \mathbb{R}^n : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$ is convex.*

Proof Take $\boldsymbol{x}, \boldsymbol{y} \in P$, to show $\boldsymbol{A}(\lambda\boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \in P$

$$\boldsymbol{A}(\lambda\boldsymbol{x} + (1-\lambda)\boldsymbol{y}) = \lambda\boldsymbol{A}\boldsymbol{x} + (1-\lambda)\boldsymbol{A}\boldsymbol{y} \leq \lambda\boldsymbol{b} + (1-\lambda)\boldsymbol{b} \leq \boldsymbol{b} \tag{2.2}$$

□

Proposition 2.5 *The set of positive semi-definite(PSD) matrices is convex.*

Proof Take $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{D \times D}$ are PSD, to show $\lambda\boldsymbol{A} + (1-\lambda)\boldsymbol{B}$ is PSD.

$$\boldsymbol{x}^T(\lambda\boldsymbol{A} + (1-\lambda)\boldsymbol{B})\boldsymbol{x} = \lambda\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} + (1-\lambda)\boldsymbol{x}^T\boldsymbol{B}\boldsymbol{x} \geq 0 \tag{2.3}$$

□

2.1.2 Convex Function

Definition 2.6 *A function $f : \mathbb{R}^n \to \mathbb{R}$ defined on a convex domain is convex if for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^D$ where f is defined and $0 \leq \lambda \leq 1$,*

$$f(\lambda\boldsymbol{x} + (1-\lambda)\boldsymbol{y} \leq \lambda f(\boldsymbol{x}) + (1-\lambda)f(\boldsymbol{y}) \tag{2.4}$$

Proposition 2.7 *Affine functions: $f(\boldsymbol{x}) = \boldsymbol{b}^T \cdot \boldsymbol{x} + c$ are convex functions.*

Proposition 2.8 *Quadratic functions: $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T \cdot \boldsymbol{A} \cdot \boldsymbol{x} + \boldsymbol{b}^T \cdot \boldsymbol{x} + c$ are convex functions.*

Proposition 2.9 *Nonnegative weighted sums of convex functions: Given convex functions $f_1, \cdots, f_n$ and $w_1, \cdots, w_n \in \mathbb{R} \leq 0$, the following is a convex function*

$$f(\boldsymbol{x}) = \sum_{i=1}^{k} w_i \cdot f_i(\boldsymbol{x}) \tag{2.5}$$

Proposition 2.10 *Norms: $\|\cdot\|_p$ are convex functions except for p=0*

2.1.3 Convex Optimization

Given convex functions $f(\boldsymbol{x}), g_1(\boldsymbol{x}), \cdots, g_m(\boldsymbol{x})$ and affine functions $h_1(\boldsymbol{x}), \cdots, h_n()\boldsymbol{x}$, a convex optimization problem is of the form:

$$\begin{aligned} \text{minimize} \quad & f(\boldsymbol{x}) \\ \text{s.t.} \quad & g_i(\boldsymbol{x}) \leq 0 \\ & h_j(\boldsymbol{x}) = 0 \end{aligned} \tag{2.6}$$

The goal is to find an optimal value of a convex problem:

$$v^* = min\{f(\boldsymbol{x}) : g_i(\boldsymbol{x}) \leq 0, i \in \{1, \cdots, m\}, h_j(\boldsymbol{x}), j \in \{0, \cdots, n\}\} \quad (2.7)$$

Whenever $f(\boldsymbol{x}^*) = v^*$ then $\boldsymbol{x}^*$ is a (not necessarily unique) optimal point.

Note that for infeasible instances(feasible: fulfils all constraints g_i and h_i), $v* \stackrel{def}{=} +\infty$; for unbounded instances(unbounded: the set of feasible instances has no infimum), $v* \stackrel{def}{=} -\infty$.

$\boldsymbol{x}$ is locally optimal if:

- $\boldsymbol{x}$ is feasible and,
- There is $B > 0$ s.t. $f(\boldsymbol{x}) \leq f(\boldsymbol{y})$ for all feasible $\boldsymbol{y}$ with $\|\boldsymbol{x} - \boldsymbol{y}\|_2 \leq B$.

$\boldsymbol{x}$ is globally optimal if:

- $\boldsymbol{x}$ is feasible and,
- $f(\boldsymbol{x}) \leq f(\boldsymbol{y})$ for all feasible $\boldsymbol{y}$.

Theorem 2.11 *For a convex optimization problem, all locally optimal points are globally optimal.*

2.1.4 Classes of Convex Optimization Problem

Linear Programming

Linear programming is to look for solutions $\boldsymbol{x} \in \mathbb{R}^n$ to the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \boldsymbol{c}^T\boldsymbol{x} + d \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{e} \\ & \boldsymbol{B}\boldsymbol{x} = \boldsymbol{f} \end{aligned} \quad (2.8)$$

The interesting thing of linear programming is that the optimal is usually achieved at the vertices of the constraint polytope.

We are not going to the algorithms of how to solve linear programming problems. However, efficient algorithms exist, both in theory and practice, for tens of thousands of variables.

Let's look at an example of linear model with absolute loss. Suppose we have data $(\boldsymbol{X}, \boldsymbol{y})$ and we want to minimise the following objective:

$$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} |\boldsymbol{x}_i^T\boldsymbol{w} - y_i| \quad (2.9)$$

We would like to transform this optimisation problem into a linear program. We introduce one ζ for each data point. The linear program in the $D+N$ variables $w_1, \cdots, w_D, \zeta_i, \cdots, \zeta_N$ is given as,

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \zeta_i \\ \text{s.t.} \quad & \boldsymbol{x}_i^T \boldsymbol{w} - y_i \leq \zeta_i, \ i = 1, \cdots, N \\ & y_i - \boldsymbol{x}_i^T \boldsymbol{w} \leq \zeta_i, \ i = 1, \cdots, N \end{aligned} \tag{2.10}$$

Since a solution of this program always exists, we let $\boldsymbol{x} \in \mathbb{R}^D$ be any vector and $\zeta_i = |\boldsymbol{x}_i^T \boldsymbol{w} - y_i|$. Then, $w_1, \cdots, w_D, \zeta_i, \cdots, \zeta_N$ represent a feasible solution yet not necessarily optimal to the linear program.

Let $\boldsymbol{w}^*, \zeta^*$ be the optimal solution. Clearly, $\zeta^* \geq |\boldsymbol{x}_i^T \boldsymbol{w} - y_i|$ due to our two constraints. This also suggests that $\zeta^* \geq 0$.

Since our linear program minimize the object $\sum_{i=1}^{N} \zeta_i^*$, we take the ζ_i^* to be the smallest possible, i.e. $\zeta_i^* = |\boldsymbol{x}_i^T \boldsymbol{w}^* - y_i|$. Thus, $\boldsymbol{w}^*$ must be the optimal solution for $\mathcal{L}$. That is, the solution of this linear programming problem gives $\boldsymbol{w}$ that minimizes the objective $\mathcal{L}$.

Quadratically Constrained Quadratic Programming

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \boldsymbol{x}^T \boldsymbol{B} \boldsymbol{x} + \boldsymbol{c}^T \boldsymbol{x} + d \\ \text{s.t.} \quad & \frac{1}{2} \boldsymbol{x}^T \boldsymbol{Q}_i \boldsymbol{x} + \boldsymbol{r}_i^T \boldsymbol{x} + s_i \leq 0 \\ & \boldsymbol{A} \boldsymbol{x} = \boldsymbol{B} \end{aligned} \tag{2.11}$$

Recall the negative log-likelihood objective we discussed in the previous chapter,

$$NLL(\boldsymbol{y} | \boldsymbol{X}, \boldsymbol{w}, \sigma) = \frac{N}{2} \log 2\pi\sigma^2 + \frac{1}{2\sigma^2} (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) \tag{2.12}$$

We can form it into a convex quadratic optimisation problem with no constraints given by,

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2\sigma^2} \boldsymbol{w}^T (\boldsymbol{X}^T \boldsymbol{X}) \boldsymbol{w} - \frac{1}{\sigma^2} (\boldsymbol{X}^T \boldsymbol{y})^T \boldsymbol{w} + \\ & (\frac{1}{2\sigma^2} \boldsymbol{y}^T \boldsymbol{y} + \frac{N}{2} \log 2\pi\sigma^2) \end{aligned} \tag{2.13}$$

What about minimising the Lasso Object,

$$\mathcal{L}_{Lasso} = \boldsymbol{w}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w} - 2\boldsymbol{y}^T \boldsymbol{X} \boldsymbol{w} + \boldsymbol{y}^T \boldsymbol{y} + \lambda \sum_{i=1}^{D} |w_i| \tag{2.14}$$

In this case, we do not have a closed-form solution because of the absolute value. We cannot frame it into a linear programming problem either because we have the quadratic loss.

However, what we can do is to rephrase it into a quadratic optimisation problem with linear constraint(as what we did in transforming the absolute loss into linear programming previously).

Semi-definite Programming

$$\begin{aligned} \text{minimize} \quad & tr(\boldsymbol{c}^T\boldsymbol{x}) \\ \text{s.t.} \quad & tr(\boldsymbol{A}_i\boldsymbol{x}) = \boldsymbol{B}_i \end{aligned} \tag{2.15}$$

Where $\boldsymbol{x}$ is positive semi-definite and $tr(\boldsymbol{A})$ is the trace of the matrix $\boldsymbol{A}$.

2.2 Non-convex Optimization

Let's first recap some calculus background.

Theorem 2.12 *If a function f is differentiable, the gradient of f at a point is either 0 or perpendicular to the contour of f at that point.*

For $f : \mathbb{R}^n \to \mathbb{R}$, its gradient $\nabla f(\boldsymbol{a})$: $\mathbb{R}^n \to \mathbb{R}$ is defined as the vector,

$$\nabla f(\boldsymbol{a}) = [\frac{\partial f}{\partial x_i}(\boldsymbol{a}) \ \cdots \ \frac{\partial f}{\partial x_n}(\boldsymbol{a})]^T \tag{2.16}$$

Each component of the gradient says how fast the function changes with respect to the standard basis,

$$\frac{\partial f}{\partial x}(\boldsymbol{a}) = \lim_{h \to 0} \frac{f(\boldsymbol{a} + h\hat{\boldsymbol{i}}) - f(\boldsymbol{a})}{h} \tag{2.17}$$

Where $\hat{\boldsymbol{i}}$ is the unit vector in the direction of x, which captures information in which direction we move.

For changing with respect to the direction of a certain arbitrary vector $\boldsymbol{v}$, the directional derivative in the direction of $\boldsymbol{v}$ is give as,

$$\nabla_{\boldsymbol{v}} f(\boldsymbol{a}) = \lim_{h \to 0} \frac{f(\boldsymbol{a} + h\hat{\boldsymbol{v}}) - f(\boldsymbol{a})}{h} = \nabla f(\boldsymbol{a}) \cdot \boldsymbol{v} \tag{2.18}$$

Geometrically speaking, $\nabla_{\boldsymbol{v}} f(\boldsymbol{a})$ simply projects $\nabla f(\boldsymbol{a})$ to $\boldsymbol{v}$,

$$\nabla f(\boldsymbol{a}) \cdot \boldsymbol{v} = \|\nabla f(\boldsymbol{a})\| \|\boldsymbol{v}\| cos\theta \tag{2.19}$$

Where $cos\theta$ is the angle between $\nabla f(\boldsymbol{a})$ and $\boldsymbol{v}$. The maximum value is obtained when $cos\theta = 1$ or $\theta = 0$, where $\nabla f(\boldsymbol{a})$ and $\boldsymbol{v}$ have the same direction.

If we want to capture the curvature of the surface, we will have to consider the matrix of all second-order partial derivatives of f, namely the Hessian matrix $\boldsymbol{H}$.

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a function taking as input a vector $\boldsymbol{x} \in \mathbb{R}^n$ and outputting a scalar $f(\boldsymbol{x}) \in \mathbb{R}$. If all second partial derivatives of f exist and are continuous over the domain of the function, then the Hessian matrix $\boldsymbol{H}$ of f is a square n×n matrix defined as,

$$(\boldsymbol{H}f)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} \tag{2.20}$$

The Hessian matrix is a symmetric matrix.

If $(\boldsymbol{H}f)$ has positive(or negative) eigenvalues, what we find at $\nabla f = 0$ is local minimum(or maximum).

If $(\boldsymbol{H}f)$ has mixed eigenvalues, what we find at $\nabla f = 0$ is saddle point.

If $(\boldsymbol{H}f)$ has eigenvalues of 0, it means that there is no inverse and thus the gradient is locally unchanging.

2.2.1 Gradient Descent

Gradient descent is one of the simplest yet very general optimization algorithms. It is an iterative algorithm, producing a new vector $\boldsymbol{w}_{t+1}$ at each iteration as,

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \boldsymbol{g}_t \tag{2.21}$$

Where $\boldsymbol{g}_t$ denotes the gradient $\nabla f(\boldsymbol{w}_t)$ and $\eta_t > 0$ is the learning rate or step size.

We can easily see that at each iteration, it moves in the direction in the steepest descent. For a convex function, no matter where it starts, it will reach the same global minimum.

For deciding the number of iterations, we usually have three tests for convergence:

- Fixed number of iterations: Terminate if $t > T$
- Small increase: Terminate if $f(\boldsymbol{w}_t - \eta_t \boldsymbol{g}_t) - f(\boldsymbol{w}) < \epsilon$
- Small change: Terminate if $\|\boldsymbol{w}_{t+1} - \boldsymbol{w}_t\| \leq \epsilon$

For very large and wide datasets(i.e. very large N and D), the computational complexity of each gradient calculation is linear in N and D while that of inverting matrices in a closed-form solution is relatively much higher.

The choice of a good step-size is also crucial . It the step-size is too large, the algorithm may never converge; if the step-size is too small, the convergence may be too slow. Sometimes, we may even want a time-varying step-size.

We can choose a constant step size,

$$\eta_t = c \tag{2.22}$$

or a decaying step-size,

$$\eta_t = c/t \tag{2.23}$$

Other different rates of decay such as $\frac{1}{\sqrt{t}}$ is also common in practice.

We can also perform the backtracking line search:

- Start with $\eta_t = C/t$(usually a large value)
- Check for a decrease: Is $f(\boldsymbol{w}_t - \eta_t \boldsymbol{g}_t) < f(\boldsymbol{w})$?
- If decrease condition is not met, multiply η_t by a decaying factor, e.g. 0.5
- Repeat until the decrease condition is met

2.2.2 Stochastic Gradient Descent

Suppose we minimize the objective function over data points $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)$,

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \ell(\boldsymbol{w}; \boldsymbol{x}_i, y_i) + \lambda \mathcal{R}(\boldsymbol{w}) \tag{2.24}$$

The gradient of the objective function is,

$$\nabla_{\boldsymbol{w}} \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{w}} \ell(\boldsymbol{w}; \boldsymbol{x}_i, y_i) + \lambda \nabla_{\boldsymbol{w}} \mathcal{R}(\boldsymbol{w}) \tag{2.25}$$

For a random point $(\boldsymbol{x}_i, y_i)$, the expectation of the gradient at that point $\boldsymbol{g}_i = \nabla_{\boldsymbol{w}} \ell(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ is given by,

$$\mathbb{E}[\boldsymbol{g}_i] = \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{w}} \ell(\boldsymbol{w}; \boldsymbol{x}_i, y_i) \tag{2.26}$$

The expectation of $\boldsymbol{g}_i$ is exactly the same as the entire gradient except for the regularization term. In another word, by just taking the local gradient and doing our gradient descent approach will eventually leads to the very same outcome. This justifies using $\boldsymbol{g}_i$ instead of $\nabla_{\boldsymbol{w}} \mathcal{L}$ in our update program for gradient descent.

The updating rule particular for stochastic gradient descent is the same as for gradient descent. But instead of computing the average gradient over the entire data set, in stochastic gradient descent, we just take a single random data point and compute the gradient of the loss function for that point.

If the objective also has a regularization term, the gradient of the regularization has to be added to the overall gradient. We have to consider the scale of the data to the regularization term. If the objective is the average loss of the data set, then we can just add the regularization at that point. If however, our objective is the sum of the loss, the gradient of the regularization term has to be scaled appropriately when using stochastic gradient descent.

The point of the stochastic gradient descent is that we compute the gradient at one data point instead of all data point. This enables so-called online-learning. In this particular case, we do not to see the whole data set. Also, it is computationally cheaper. This is why stochastic gradient descent is rather appealing these days.

However, this is not necessarily always the case. There are recent papers shows doing batch gradient descent for a specific type of data that is very sparse is preferred over stochastic gradient descent. The following experiments compares the stochastic gradient descent with batch gradient descent.

Suppose we have 1000 data points for training and 1000 data points for test. There are two features drawn from the normal distribution $x_1 \in \mathcal{N}(0, 5)$ and $x_2 \in \mathcal{N}(0, 8)$. We take the labels to be centered so that there is no intercept. Thus the least-squares linear regression model is given as $f_w(\boldsymbol{x}) = x_1 w_1 + x_2 w_2$. The results of the experiment with parameters (w_1, w_2) initialized as $(-2, -3)$ and finally converging to $(1, 1)$ is shown in Figure 2.1.

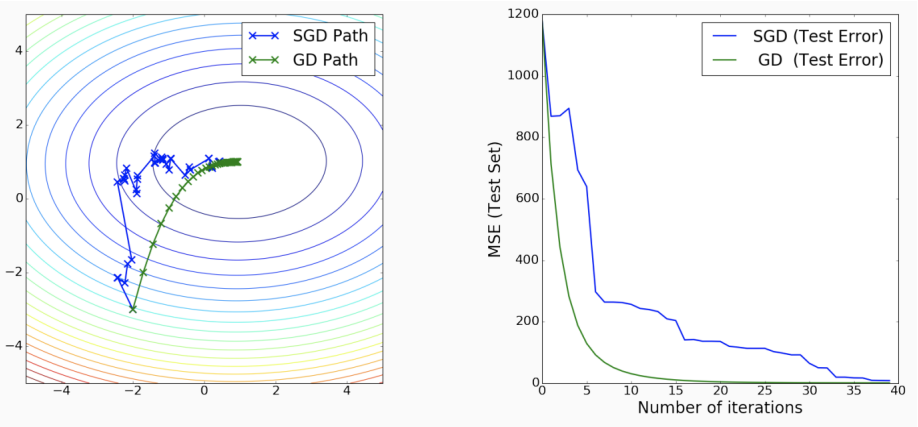

Figure 2.1: Experiment Results: SGD v.s. GD

As is shown in the Figure 2.1, it only takes gradient descent approach 15 to 20 iterations to converge while it takes stochastic gradient descent 40 iterations to reach the same result. However, there is another aspect that is equally relevant we did not take into account. For gradient descent, we need to look at the whole data set. Getting 15 or 20 of such iterations is much more computationally expensive than see 40 data points in 40 iterations of stochastic gradient descent.

In practice, what happens is that we do not necessarily use pure stochastic gradient descent. We can use what is called mini-batch gradient descent which uses batches that are fairly small. It reduces the variance in the gradients and hence it is more stable than stochastic gradient descent.

2.2.3 Sub-Gradient Descent

Recall the linear model trained with least-squares model and ℓ_1 regularization,

$$\mathcal{L}_{lasso}(\boldsymbol{w}) = \sum_{i=1}^{N} (\boldsymbol{w}^T \boldsymbol{x}_i - y_i)^2 + \lambda \sum_{i=1}^{D} |w_i| \tag{2.27}$$

When applying gradient descent on the lasso objective, we will encounter the problem that the objective function is not differentiable.

In mathematics, the sub-gradient generalizes the derivative to convex functions which are not necessarily differentiable. Sub-gradient arise in convex analysis, the study of convex functions, often in connection to convex optimization.

For a convex function f, a $\boldsymbol{g}$ satisfying,

$$f(\boldsymbol{x}) \geq f(\boldsymbol{x}_0) + \boldsymbol{g}^T(\boldsymbol{x} - \boldsymbol{x}_0) \tag{2.28}$$

is called a sub-gradient at $\boldsymbol{x}_0$.

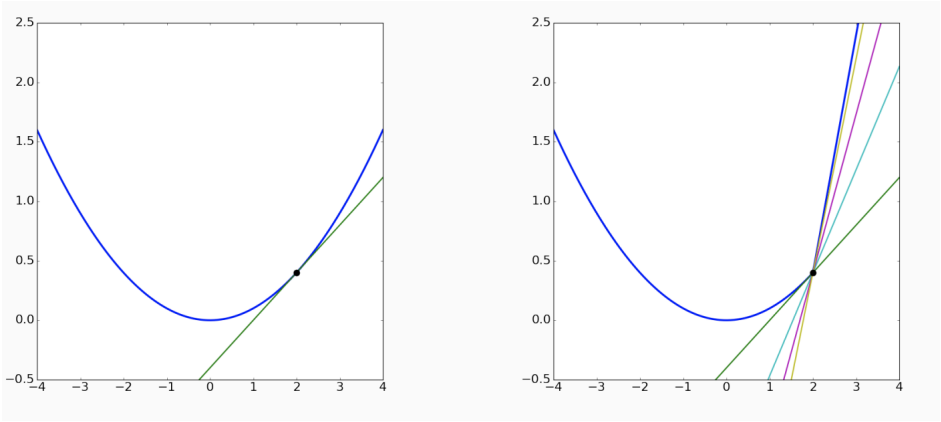

Figure 2.2: Sub-gradient

2.2.4 Constrained Convex Optimization

Recall what we did with gradient descent. We want to minimize $f(\boldsymbol{x})$ by moving in the negative gradient direction at each step,

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla f(\boldsymbol{w}_t) \tag{2.29}$$

Here we have no constraint on the parameters. We may choose them as we like so that we can reach the minimizer.

But sometimes in practice, it is important to minimize a function subject to additional constraints $\boldsymbol{w}_C \in C$. Here C is a set of possible values for our parameters $\boldsymbol{w}$.

So how can we solve such an optimization problem?

We first use the gradient step as we did before,

$$\boldsymbol{z}_{t+1} = \boldsymbol{w}_t - \eta_t \nabla f(\boldsymbol{w}_t) \tag{2.30}$$

This $\boldsymbol{z}_{t+1}$ we obtained in the equation above may happen to be outside our valid set C, i.e. it does not satisfy our constraints.

Then the way we have to proceed is that we should try to find a valid $\boldsymbol{w}_C$ in our allowed set of possible value which is closest to $\boldsymbol{z}_{t+1}$. We can express this property as,

$$\boldsymbol{w}_{t+1} = \underset{\boldsymbol{w}_C \in C}{\arg\min} \ \|\boldsymbol{z}_{t+1} - \boldsymbol{w}_C\| \tag{2.31}$$

This is called the projection step.

Suppose we want to minimize $(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$ subject to the ridge constraints $\boldsymbol{w}^T\boldsymbol{w} < R$.

Imagine after t^{th} iteration $\boldsymbol{w}_t$ is inside the allowed set defined by the ball of radius R. At $(t+1)^{th}$ iteration with the gradient step $\boldsymbol{w}_{t+1}$ goes up to outside the ball towards the minimizer. Then we have to project it back into the ball, which is shown by magenta line in Figure 2.3.

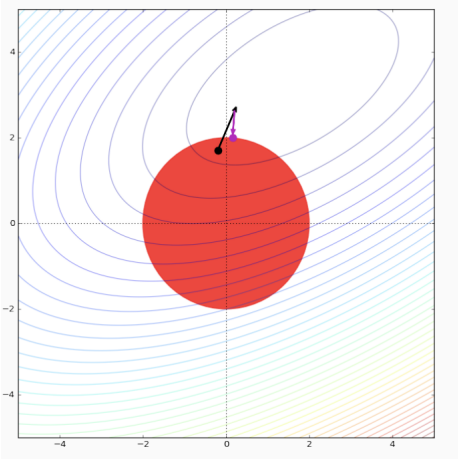

Figure 2.3: Ridge Constrained Convex Optimization

2.2.5 Second Order Methods

Gradient descent is a very general method for optimization. We can actually see gradient descent as an approach that approximates the function locally by a linear function and minimizes it instead. However, unless this linear function is constant(i.e. the gradient is 0) in which case the trajectory is the stationary point anyway, the minimum of this linear approximation will be negative infinity, which is why we have to choose carefully the step size.

In the case of Newton's method, we use the first and second-order partial derivatives. At any point, the trajectory will approximate the function by a quadratic function using the second-degree Taylor approximation. So a Newton step will take us directly to a unique stationary point of this quadratic approximation.

In optimization, the roots of f'(i.e. solutions of $f'(\boldsymbol{x}) = 0$) are stationary points(i.e. minimum/maximum/saddle points) of f, where f is a twice-differentiable function.

Newton's Method in One Dimension

- Construct a sequence of points $x_1, \cdots, x_n$ starting with an initial guess x_0
- Sequence converges towards a minimiser x^* of f using the sequence of second-order Taylor approximation of f around the iterates:

$$f(x) \approx f(x_k) + (x - x_k)f'(x_k) + \frac{1}{2}(x - x_k)^2 f''(x_k) \tag{2.32}$$

- $x_{k+1} = x^*$ is defined as the minimizer of this quadratic function.
- If f'' is positive, then the quadratic approximation is convex, and a minimizer is obtained by setting the derivative to 0:

$$0 = \frac{d(f(x_k) + (x - x_k)f'(x_k) + \frac{1}{2}(x - x_k)^2 f''(x_k))}{dx} = f'(x_k) + (x^* - x_k)f''(x_k) \tag{2.33}$$

Then,

$$x_{k+1} = x^* = x_k - f'(x_k)[f''(x_k)]^-1 \tag{2.34}$$

Geometric Interpretation of Newton's Method

At iteration k, we fit a paraboloid to the surface of f at x_k with the same slopes and curvature as the surface at x_k and go for the extremum of that paraboloid.

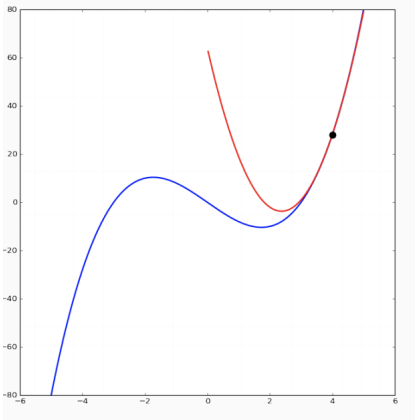

Figure 2.4: Geometric Interpretation of Newton's Method

Newton's Method in High Dimensions

- Approximate f around $\boldsymbol{x}_k$ using second-order Taylor approximation:

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x}_k) + \boldsymbol{g}_k^T(\boldsymbol{x} - \boldsymbol{x}_k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_k)\boldsymbol{H}_k(\boldsymbol{x} - \boldsymbol{x}_k) \tag{2.35}$$

  Where $\boldsymbol{H}_k$ is the Hessian of f.

- The gradient of f is given by,

$$\nabla_{\boldsymbol{x}} = \boldsymbol{g}_k + \boldsymbol{H}_k(\boldsymbol{x} - \boldsymbol{x}_k) \tag{2.36}$$

- Setting $\nabla_{\boldsymbol{x}} = 0$, we obtain $\boldsymbol{x}^* = \boldsymbol{x}_k - \boldsymbol{H}_k^{-1}\boldsymbol{g}_k$
- We move directly to the unique stationary point $\boldsymbol{x}^*$ of f
- We repeat the above iteration with $\boldsymbol{x}_{k+1} = \boldsymbol{x}^*$

Computation and Convergence

- Computational requirements at each Newton step
  - $D + \binom{D}{2}$ partial derivatives and inverse of Hessian
  - Instead: Compute $\boldsymbol{x}$ as the solution of the system of linear equations

$$\boldsymbol{H}_k(\boldsymbol{x} - \boldsymbol{x}_k) = -\boldsymbol{g}_k \tag{2.37}$$

    using factorization (e.g. Cholesky) of $\boldsymbol{H}_k$

- For convex function f
  - It converges to stationary points of the quadratic approximation
  - All stationary points are global minimum
  - Converges to unique minimizer quadratically fast if f is strongly convex
- For non-convex f
  - Stationary points may not be minimum nor in the decreasing direction of f
  - Not successful for training deep neural networks because of the abundance of saddle points for their non-convex objective functions

[../main.tex]subfiles

3

Classification

In machine learning, classification refers to a predictive modeling problem where a class label is predicted for a given example of input data. The input vector $\boldsymbol{x} = [x1, \cdots, x_D]^T$ consists of categorical and continuous features. The output y is a category,

$$y \in \{1, \cdots, C\} \tag{3.1}$$

In classification problems, two main approaches are called the generative approach and the discriminative approach.

A discriminative setting is to model the conditional distribution of the output y given the input $\boldsymbol{x}$ and the model parameters $\boldsymbol{\theta}$, symbolically, $p(y \mid \boldsymbol{x}, \boldsymbol{\theta})$.

A generative setting is to, however, model the full joint distribution of the input $\boldsymbol{x}$ and output y given the model parameters $\boldsymbol{\theta}$, symbolically, $p(\boldsymbol{x}, y \mid \boldsymbol{\theta})$.

3.1 Generative Models for Classification

Given generative model $p(\boldsymbol{x}, y \mid \boldsymbol{\theta})$ representing the joint distribution of $\boldsymbol{x}$ and y, for a new input $\boldsymbol{x}_{new}$, the conditional distribution for y, according to the Bayes rule, is given as,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{p(\boldsymbol{x}_{new} \mid y = c, \boldsymbol{\theta}) p(y = c \mid \boldsymbol{\theta})}{p(\boldsymbol{x}_{new} \mid \boldsymbol{\theta})} \tag{3.2}$$

Where $c \in \{1, \cdots, C\}$ are the possible categories(i.e. labels) for y.

The numerator is the joint probability distribution $p(\boldsymbol{x}_{new}, y = c \mid \boldsymbol{\theta})$ and the denominator is the marginal distribution $p(\boldsymbol{x}_{new} \mid \boldsymbol{\theta})$.

According to the law of total probability,

$$p(\boldsymbol{x}_{new} \mid \boldsymbol{\theta}) = \sum_{c'=1}^{C} p(\boldsymbol{x}_{new} \mid y = c', \boldsymbol{\theta}) p(y = c' \mid \boldsymbol{\theta}) \tag{3.3}$$

Thus, the conditional distribution for y according to the Bayes rule can be rephrased as,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{p(\boldsymbol{x}_{new} \mid y = c, \boldsymbol{\theta}) p(y = c \mid \boldsymbol{\theta})}{\sum_{c'=1}^{C} p(\boldsymbol{x}_{new} \mid y = c', \boldsymbol{\theta}) p(y = c' \mid \boldsymbol{\theta})} \tag{3.4}$$

To predict the most-likely class:

$$\hat{y} = \arg\max_{c} p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) \tag{3.5}$$

Suppose we have the output y only parameterized by $\boldsymbol{\pi}$. The joint distribution in our generative model can be expressed as,

$$p(\boldsymbol{x}, y \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = p(y \mid \boldsymbol{\theta}, \boldsymbol{\pi}) p(\boldsymbol{x} \mid y, \boldsymbol{\theta}, \boldsymbol{\pi}) = p(y \mid \boldsymbol{\pi}) p(\boldsymbol{x} \mid y, \boldsymbol{\theta}) \tag{3.6}$$

We use parameters π_c such that $\sum_c \pi_c = 1$ and model the marginal distribution of output y as,

$$p(y = c \mid \boldsymbol{\pi}) = \pi_c \tag{3.7}$$

For each class $c = 1, \cdots, C$, the class-conditional distributions of input $\boldsymbol{x}$ given the class label y parameterized by $\boldsymbol{\theta}$ is modeled as $p(\boldsymbol{x} \mid y = c, \boldsymbol{\theta}_c)$.

To define the model, we need to estimate the parameters π_c, θ_c for each $c \in \{1, \cdots, C\}$.

Assume we have the dataset $\mathcal{D} = ((\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_N, y_N))$, where each vector $\boldsymbol{x}_i \in \mathbb{R}^D$. Each data point is drawn independently from the same distribution $p(\boldsymbol{x}, y)$. Let N_c be the number of data points with $y_i = c$ so that $\sum_{c=1}^{C} N_c = N$.

The probability for a single data point $(\boldsymbol{x}_i, y_i)$ is given as,

$$p(\boldsymbol{x}_i, y_i \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = p(y_i \mid \boldsymbol{\pi}) p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta}) = \prod_{c=1}^{C} \pi_c^{\mathbb{1}(y_i = c)} p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta}) \tag{3.8}$$

Where $\mathbb{1}$ is the indicator function, i.e. $\mathbb{1} = 1$ for $y_i = c$ and $\mathbb{1} = 0$ otherwise.

The likelihood of the data is given by,

$$p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{i=1}^{N} p(\boldsymbol{x}_i, y_i \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{i=1}^{N} (\prod_{c=1}^{C} \pi_c^{\mathbb{1}(y_i = c)} p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta})) \tag{3.9}$$

The log-likelihood is given by,

$$\log p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{c=1}^{C} N_c \log \pi_c + \sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta}) \tag{3.10}$$

Where N_c is the number of data points falls in category $y_i = c$.

The first term $\sum_{c=1}^{C} N_c \log \pi_c$ is only dependent on parameter $\boldsymbol{\pi}$ while the second term $\sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta})$ is independent of $\boldsymbol{\pi}$.

Let us first estimate for $\boldsymbol{\pi}$. We get the constrained optimization problem,

$$\begin{aligned} \text{maximize} \quad & \sum_{c=1}^{C} N_c \log \pi_c \\ \text{s.t.} \quad & \sum_{c=1}^{C} \pi_c - 1 = 0 \end{aligned} \tag{3.11}$$

This problem can be solved by the method of Lagrange multipliers given by,

$$\Lambda(\boldsymbol{pi}, \lambda) = \sum_{c=1}^{C} N_c \log \pi_c - \lambda(\sum_{c=1}^{C} \pi_c - 1) \tag{3.12}$$

Setting the partial derivatives with respect to $\boldsymbol{pi}$ and λ to zero,

$$\begin{aligned} \frac{\partial \Lambda(\boldsymbol{\pi}, \lambda)}{\partial \pi_c} &= \frac{N_c}{\pi_c} - \lambda = 0 \\ \frac{\partial \Lambda(\boldsymbol{\pi}, \lambda)}{\partial \lambda} &= \sum_{c=1}^{C} \pi_c - 1 = 0 \end{aligned} \tag{3.13}$$

we obatin,

$$\begin{aligned} \pi_c &= \frac{N_c}{\lambda} \\ \lambda &= \sum_{c=1}^{C} N_c = N \end{aligned} \tag{3.14}$$

Thus, we get the estimation,

$$\pi_c = \frac{N_c}{N} \tag{3.15}$$

That is to say, for generative classification models, the marginal distribution $p(y \mid \boldsymbol{\pi})$ over the classes is the empirical distribution over the classes.

We next should estimate the parameters $\boldsymbol{\theta}$ of the model to choose the suitable model for the class-conditional distribution $p(\boldsymbol{x} \mid y, \boldsymbol{\theta})$.

3.1.1 Naïve Bayes Classifier

The Naïve Bayes classifier is based on Bayes rule and a set of conditional independence assumptions.

Definition 3.1 *Given three sets of random variables X, Y and Z, we say X is conditionally independent of Y given Z if and only,*

$$\forall(i, j, k) p(X = x_i \mid Y = y_j, Z = z_k) = p(X = x_i \mid Z = z_k) \tag{3.16}$$

Assume that the features are conditionally independent given the class label c. The class-conditional distribution for data point $(\boldsymbol{x}_i, y_i)$ becomes,

$$p(\boldsymbol{x}_i \mid y_i = c, \boldsymbol{\theta}) = p(\boldsymbol{x}_i \mid y_i = c, \boldsymbol{\theta}_c) = \prod_{j=1}^{D} p(x_{ij} \mid \boldsymbol{\theta}_{jc}) \tag{3.17}$$

Consider, for example, the case where the input $\boldsymbol{x}_i$ has two features x_{i1} and x_{i2}. In this case,

$$\begin{aligned} p(\boldsymbol{x}_i \mid y_i = c, \boldsymbol{\theta}_c) &= p(x_{i1}, x_{i2} \mid y = c, \boldsymbol{\theta}_c) \\ &= p(x_{i1} \mid x_{i2}, y = c, \boldsymbol{\theta}_c) p(x_{i2} \mid y = c, \boldsymbol{\theta}_c) \\ &= p(x_{i1} \mid y = c, \boldsymbol{\theta}_c) p(x_{i2} \mid y = c, \boldsymbol{\theta}_c) \end{aligned} \tag{3.18}$$

Now the joint distribution given the parameters $\boldsymbol{\theta}$ and $\boldsymbol{\pi}$ becomes,

$$\begin{aligned} p(\boldsymbol{x}_i, y_i \mid \boldsymbol{\theta}, \boldsymbol{\pi}) &= p(y_i \mid \boldsymbol{\pi}) p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta}) \\ &= \prod_{c=1}^{C} \pi_c^{\mathbb{1}(y_i=c)} p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta}) \\ &= \prod_{c=1}^{C} \pi_c^{\mathbb{1}(y_i=c)} \prod_{c=1}^{C} \prod_{j=1}^{D} p(x_{ij} \mid \boldsymbol{\theta}_{jc})^{\mathbb{1}(y_i=c)} \end{aligned} \tag{3.19}$$

The likelihood of the data is given by,

$$p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{i=1}^{N} p(\boldsymbol{x}_i, y_i \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{i=1}^{N} (\prod_{c=1}^{C} \pi_c^{\mathbb{1}(y_i=c)} \prod_{c=1}^{C} \prod_{j=1}^{D} p(x_{ij} \mid \boldsymbol{\theta}_{jc})^{\mathbb{1}(y_i=c)}) \tag{3.20}$$

The log-likelihood is given by,

$$\log p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{c=1}^{C} N_c \log \pi_c + \sum_{c=1}^{C} \sum_{j=1}^{D} \sum_{i:y_i=c} \log p(x_{ij} \mid \boldsymbol{\theta}_{jc}) \tag{3.21}$$

Since we model for individual features with naïve Bayes classifier, it is easy to mix and match different models for different features.

For example, if feature x_j is real-valued(i.e. $x_j \in \mathbb{R}$), we can use Gaussian model $\theta_{jc} = (\mu_{jc}, \sigma_{jc}^2)$ where maximum likelihood estimation for θ_{jc} is given by the empirical mean and variance. If feature x_j is categorical with categories $1, \cdots, k$. We can use multinoulli model: $x_j = \ell$ with probability $\theta_{jc,\ell}$ such that $\sum_{\ell=1}^{k} \theta_{jc,\ell} = 1$, where maximum likelihood estimation for θ_{jc} is obtained similarly to that for π_c. Also, there is no need to convert categorical variables to real-valued vectors this way.

With the naïve Bayes conditional independence assumption, naïve Bayes has at most $C \cdot D$ parameters $(\theta_{jc})_{j\in[D],c\in[C]}$. However, without the conditional independence assumption, we have to assign a probability for each of

the 2^D possible feature vectors. Thus, we have $C \cdot 2^D$ parameters which leads to either overfitting if $N \cdot 2^D$ or computationally prohibitive if $N \gg \cdot 2^D$. The 'naïve' assumption breaks the curse of dimensionality and avoids overfitting.

With the naïve Bayes classifier, we can also deal with missing data for prediction. Recall the prediction rule in a generative model,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{\pi_c \prod_{j=1}^{D} p(x_j \mid y = c, \boldsymbol{\theta}_{jc})}{\sum_{c'=1}^{C} \prod_{j=1}^{D} p(x_j \mid y = c', \boldsymbol{\theta}_{jc}) p(y = c' \mid \boldsymbol{\theta})} \tag{3.22}$$

Supposing our data point $\boldsymbol{x}_{new}$ is missing its first element x_1, we can simply skip it,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{\pi_c \prod_{j=2}^{D} p(x_j \mid y = c, \boldsymbol{\theta}_{jc})}{\sum_{c'=1}^{C} \prod_{j=2}^{D} p(x_j \mid y = c', \boldsymbol{\theta}_{jc}) p(y = c' \mid \boldsymbol{\theta})} \tag{3.23}$$

This can be done for other generative models, but marginalization requires summation/integration.

Gaussian Discriminant Analysis

Recall the form of the joint distribution in a generative model,

$$p(\boldsymbol{x}, y \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = p(y \mid \boldsymbol{\pi}) p(\boldsymbol{x} \mid y, \boldsymbol{\theta}) \tag{3.24}$$

For classes, we use parameters π_c such that that $\sum_{c=1}^{C} \pi_c = 1$,

$$p(y = c \mid \boldsymbol{\pi}) = \pi_c \tag{3.25}$$

If we model the class-conditional density for class $c \in \{1, \cdots, C\}$ as a multivariate normal distribution with mean $\boldsymbol{\mu}_c$ and covariance matrix $\boldsymbol{\Sigma}_c$,

$$p(\boldsymbol{x} \mid y = c, \boldsymbol{\theta}_c) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{3.26}$$

the model is given as,

$$p(\boldsymbol{x}, y \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{c=1}^{C} \pi_c^{\mathbb{1}(y=c)} \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \tag{3.27}$$

Note that the term 'discriminant' here refers to the shape of the boundaries that discriminate between the classes and it is still a generative model.

The model here is more suitable than naïve Bayes classifier in some cases such as predicting zebra or giraffe based on animal height and weight.

Given data $\mathcal{D} = ((\boldsymbol{x}_1), y_1), \cdots, (\boldsymbol{x}_N), y_N)$, the log-likelihood is given by,

$$\log p(\mathcal{D} \mid \boldsymbol{\theta}) = \sum_{c=1}^{C} N_c \log \pi_c + \sum_{c=1}^{C} \Big(\sum_{i: y_i = c} \log \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) \Big) \tag{3.28}$$

As is in the case of naïve Bayes classifier, we have

$$\pi_c = \frac{N_c}{N} \tag{3.29}$$

For parameters $\boldsymbol{\mu}_c$ and $\boldsymbol{\Sigma}_c$, we have:

$$\begin{aligned} \hat{\boldsymbol{\mu}}_c &= \frac{1}{N_c} \sum_{i:y_i=c} \boldsymbol{x}_i \\ \hat{\boldsymbol{\Sigma}}_c &= \frac{1}{N_c} \sum_{i:y_i=c} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_c)(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_c)^T \end{aligned} \tag{3.30}$$

We won't go into the details of the maximum likelihood estimation of these parameters. If interested, see section 4.1 from Murphy's book.

3.1.2 Quadratic Discriminant Analysis

The prediction rule for the generative model is,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{p(\boldsymbol{x}_{new} \mid y = c, \boldsymbol{\theta}) p(y = c \mid \boldsymbol{\theta})}{\sum_{c'=1}^{C} p(\boldsymbol{x}_{new} \mid y = c', \boldsymbol{\theta}) p(y = c' \mid \boldsymbol{\theta})} \tag{3.31}$$

When the densities $p(\boldsymbol{x} \mid y = c, \boldsymbol{\theta}_c)$ are multivariate normal distributions,

$$p(y = c \mid \boldsymbol{x}_{new}, \boldsymbol{\theta}) = \frac{\pi_c |2\pi\boldsymbol{\Sigma}_c|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}_c^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_c)}}{\sum_{c'=1}^{C} \pi_{c'} |2\pi\boldsymbol{\Sigma}_{c'}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{c'})^T \boldsymbol{\Sigma}_{c'}^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_{c'})}} \tag{3.32}$$

The denominator is the same for all classes $c' \in \{1, \cdots, C\}$.

The boundaries between two classes c_1 and c_2 is given by the data points $\boldsymbol{x}$ with,

$$p(y = c_1 \mid \boldsymbol{x}, \boldsymbol{\theta}) = p(y = c_2 \mid \boldsymbol{x}, \boldsymbol{\theta}) \tag{3.33}$$

This means,

$$\frac{\pi_{c_1} |2\pi\boldsymbol{\Sigma}_{c_1}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{c_1})^T \boldsymbol{\Sigma}_{c_1}^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_{c_1})}}{\pi_{c_2} |2\pi\boldsymbol{\Sigma}_{c_2}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{c_2})^T \boldsymbol{\Sigma}_{c_2}^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_{c_2})}} = 1 \tag{3.34}$$

By taking the log, we obatin,

$$\frac{1}{2}((\boldsymbol{x} - \boldsymbol{\mu}_{c_2})^T \boldsymbol{\Sigma}_{c_2}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_{c_2}) - (\boldsymbol{x} - \boldsymbol{\mu}_{c_1})^T \boldsymbol{\Sigma}_{c_1}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_{c_1})) = \log \frac{\pi_{c_2} |2\pi\boldsymbol{\Sigma}_{c_2}|^{-\frac{1}{2}}}{\pi_{c_1} |2\pi\boldsymbol{\Sigma}_{c_1}|^{-\frac{1}{2}}} \tag{3.35}$$

Boundaries are given by quadratic functions and thus piece-wise quadratic curves as is shown in Figure 3.1.

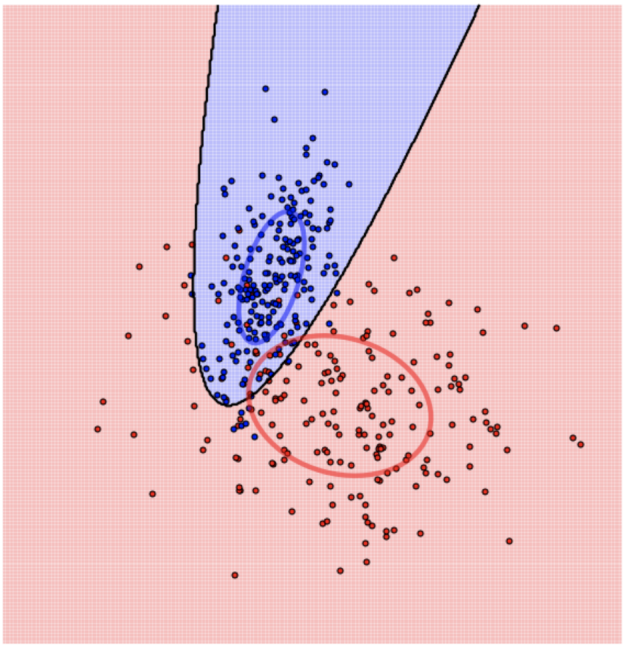

Figure 3.1: Quadratic Decision Boundaries

In quadratic discriminant analysis, the number of parameters is roughly $C \cdot D^2$. In high dimensions, this can lead to overfitting. How can we solve this problem?

We can use weight tying or parameter sharing to make the quadratic discriminant analysis problem degenerates to the linear discriminant analysis problem.

Moreover, we can use diagonal covariance matrices which corresponds to naïve Bayes. Since entries in diagonal matrices which are not along the diagonals is 0, the covariance between any two distinct features is 0, meaning that there is no correlation. This is exactly what happen with naïve Bayes - we make the conditional independence assumption that given the class labels, the features will be independent of each other.

In fact, the naïve Bayes is a special case of linear discriminant analysis, which is a special case of quadratic discriminant analysis.

Sometimes, we can use a discriminative classifier such as logistic regression and add regularization if needed.

3.1.3 Linear Discriminant Analysis

If the covariance matrix are shared or tied across different classes, denoted as $\boldsymbol{\Sigma}$, our generative model is given by,

$$\begin{aligned} p(y=c \mid \boldsymbol{x}, \boldsymbol{\theta}) &= \frac{\pi_c |2\pi\boldsymbol{\Sigma}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_c)}}{\sum_{c'=1}^{C} \pi_{c'} |2\pi\boldsymbol{\Sigma}|^{-\frac{1}{2}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{c'})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{c'})}} \\ &= \frac{\pi_c e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_c)}}{\sum_{c'=1}^{C} \pi_{c'} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{c'})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{c'})}} \\ &= \frac{e^{\boldsymbol{\mu}_c^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_c^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c + \log \pi_c} \cdot e^{-\frac{1}{2}\boldsymbol{x}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x}}}{\sum_{c'=1}^{C} e^{\boldsymbol{\mu}_{c'}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_{c'}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_{c'} + \log \pi_{c'}} \cdot e^{-\frac{1}{2}\boldsymbol{x}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x}}} \\ &= \frac{e^{\boldsymbol{\mu}_c^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_c^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c + \log \pi_c}}{\sum_{c'=1}^{C} e^{\boldsymbol{\mu}_{c'}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_{c'}^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_{c'} + \log \pi_{c'}}} \end{aligned} \tag{3.36}$$

If we let $\boldsymbol{\beta}_c = \boldsymbol{\mu}_c$ and $\gamma_c = -\frac{1}{2}\boldsymbol{\mu}_c^T \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu}_c + \log \pi_c$,

$$p(y=c \mid \boldsymbol{x}, \boldsymbol{\theta}) = \frac{e^{\boldsymbol{\beta}_c^T \boldsymbol{x} + \gamma_c}}{\sum_{c'=1}^{C} e^{\boldsymbol{\beta}_{c'}^T \boldsymbol{x} + \gamma_{c'}}} = softmax(\boldsymbol{\eta})_c \tag{3.37}$$

Where $\boldsymbol{\eta} = [\boldsymbol{\beta}_1^T \boldsymbol{x} + \gamma_1, \cdots, \boldsymbol{\beta}_C^T \boldsymbol{x} + \gamma_C]$.

The boundaries between two classes c_1 and c_2 is given by the data points $\boldsymbol{x}$ with,

$$p(y=c_1 \mid \boldsymbol{x}, \boldsymbol{\theta}) = p(y=c_2 \mid \boldsymbol{x}, \boldsymbol{\theta}) \tag{3.38}$$

This means,

$$\frac{e^{\boldsymbol{\beta}_{c_1}^T \boldsymbol{x} + \gamma_{c_1}}}{e^{\boldsymbol{\beta}_{c_2}^T \boldsymbol{x} + \gamma_{c_2}}} = 1 \tag{3.39}$$

By taking the log, we obatin,

$$\boldsymbol{\beta}_{c_1}^T \boldsymbol{x} + \gamma_{c_1} = \boldsymbol{\beta}_{c_2}^T \boldsymbol{x} + \gamma_{c_2} \tag{3.40}$$

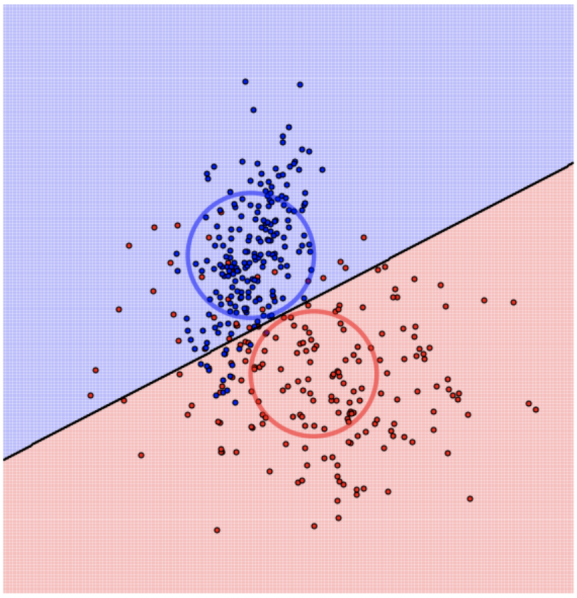

Figure 3.2: Linear Decision Boundaries

Boundaries are given by linear functions and thus piece-wise linear curves as is shown in Figure 3.2.

3.1.4 Softmax Function

Softmax function is given by,

$$softmax([z_1, \cdots, z_n])_i = \frac{e^{z_i}}{\sum_{j \in [n]} e^{z_j}} \tag{3.41}$$

It normalizes the vector values into a probability distribution consisting of probabilities proportional to their exponentials.

Softmax can be seen as a smooth approximation of the argmax function. It is commonly used in statistical mechanics as the Boltzmann distribution.

Softmax has the property that large inputs generates large probabilities. It is translation invariant(i.e. multiplying all values by the same quantity does not change the ranking) but not scale invariant(i.e. de-emphasises the max value for values $\in [0, 1]$)

Softmax is often used as the last activation function of a neural network to normalise the network output to a probability distribution over predicted output classes.

3.1.5 Two-Class Linear Decision Boundaries and Sigmoid Function

When we have only 2 classes $y \in [0, 1]$,

$$\begin{aligned} p(y = 1 \mid \boldsymbol{x}, \boldsymbol{\theta}) &= \frac{e^{\boldsymbol{\beta}_1^T \boldsymbol{x} + \gamma_1}}{e^{\boldsymbol{\beta}_1^T \boldsymbol{x} + \gamma_1} + e^{\boldsymbol{\beta}_0^T \boldsymbol{x} + \gamma_0}} \\ &= \frac{1}{1 + \frac{e^{\boldsymbol{\beta}_1^T \boldsymbol{x} + \gamma_1}}{e^{\boldsymbol{\beta}_0^T \boldsymbol{x} + \gamma_0}}} \\ &= \frac{1}{1 + e^{-((\boldsymbol{\beta}_1^T - \boldsymbol{\beta}_0^T)\boldsymbol{x} + (\gamma_1 - \gamma_0))}} \\ &= sigmoid((\boldsymbol{\beta}_1^T - \boldsymbol{\beta}_0^T)\boldsymbol{x} + (\gamma_1 - \gamma_0)) \end{aligned} \tag{3.42}$$

The sigmoid function is defined as:

$$sigmoid(t) = \frac{1}{1 + e^{-t}} = softmax([0, t]) \tag{3.43}$$

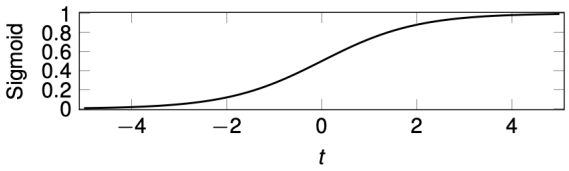

Figure 3.3: Sigmoid Function

In general, sigmoid functions refer to functions with S-shaped curves such as hyperbolic tangent and logistic(Equation 3.43) which is a special case of softmax for 1D axis in 2D space.

Sigmoid function is differentiable and has non-negative derivative at each point. It is monotonic. For $t < 0$ it is convex and for $t > 0$ it is concave.

3.2 Logistic Regression

Logistic regression is a discriminative classification method, i.e. it models the conditional distribution over the categorical output y given the input $\boldsymbol{x}$ and parameters $\boldsymbol{w}$, symbolically, $p(y \mid \boldsymbol{w}, \boldsymbol{x})$.

3.2.1 Binary-Class Logistic Regression

In the case of binary classification, it is quite convenient to use the Bernoulli random variable $X \in \{0, 1\}$.

If X is a random variable with Bernoulli distribution, i.e. $X \sim Bernoulli(\theta), \theta \in [0, 1]$ where,

$$X = \begin{cases} 1 & \text{with probability } \theta \\ 0 & \text{with probability } 1 - \theta \end{cases}$$

The probability mass function of this distribution is given as,

$$\begin{aligned} p(1 \mid \theta) &= \theta \\ p(0 \mid \theta) &= 1 - \theta \end{aligned} \tag{3.44}$$

More succinctly, it can be rephrased as,

$$p(x \mid \theta) = \theta^x (1 - \theta)^{1-x} \tag{3.45}$$

for $x \in \{0, 1\}$.

Given input $\boldsymbol{x}$ and models with parameters $\boldsymbol{w}$ producing a value $f(\boldsymbol{x}, \boldsymbol{w}) \in [0, 1]$, we model the binary class label as,

$$y \sim Bernoulli(f(\boldsymbol{x}, \boldsymbol{w})) \tag{3.46}$$

For logistic regression, it builds up on a linear model composed with a sigmoid function $\sigma(t)$,

$$p(y \mid \boldsymbol{w}, \boldsymbol{x}) = Bernoulli(\sigma(\boldsymbol{w} \cdot \boldsymbol{x})) \tag{3.47}$$

Suppose we have estimated the model parameters $\boldsymbol{w} \in \mathbb{R}^D$, for a new data point $\boldsymbol{x}_{new}$, the model gives the probability,

$$p(y_{new} = 1 \mid \boldsymbol{x}_{new}, \boldsymbol{w}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x}_{new}) \tag{3.48}$$

Recall that the sigma function $\sigma : \mathbb{R} \to (0, 1)$ is defined by,

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{3.49}$$

Thus,

$$p(y_{new} = 1 \mid \boldsymbol{x}_{new}, \boldsymbol{w}) = \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}_{new}}} \tag{3.50}$$

Let's now take a look at how to compute the decision boundaries. For any given parameter θ of the Bernoulli distribution,

$$p(y = 1 \mid \boldsymbol{x}, \boldsymbol{w}) = \frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}}} = \theta \tag{3.51}$$

Thus,

$$e^{-\boldsymbol{w} \cdot \boldsymbol{x}} = \frac{1}{\theta} - 1 \tag{3.52}$$

Subtracting 1 from the denominator,

$$\frac{1}{1 + e^{-\boldsymbol{w} \cdot \boldsymbol{x}} - 1} = \frac{1}{e^{-\boldsymbol{w} \cdot \boldsymbol{x}}} = \frac{1}{\frac{1}{\theta} - 1} = \frac{\theta}{1 - \theta} \tag{3.53}$$

Taking log on both sides,

$$\log 1 - \log e^{-\boldsymbol{w} \cdot \boldsymbol{x}} = \boldsymbol{w} \cdot \boldsymbol{x} = \log \frac{\theta}{1 - \theta} \tag{3.54}$$

And we obtain the hyper-plane,

$$\boldsymbol{w} \cdot \boldsymbol{x} = \log \frac{\theta}{1 - \theta} \tag{3.55}$$

Since $\sigma(t) \geq \frac{1}{2}$ for $\forall t \geq 0$, in order to make a prediction, we can simply use a threshold at $\frac{1}{2}$ given as,

$$\hat{y}_{new} = \mathbb{1}(\sigma(\boldsymbol{w} \cdot \boldsymbol{x}_{new}) \geq \frac{1}{2}) = \mathbb{1}(\boldsymbol{w} \cdot \boldsymbol{x}_{new} \geq 0) \tag{3.56}$$

The decision boundary is obtained when,

$$p(y = 1 \mid \boldsymbol{x}, \boldsymbol{w}) = p(y = 0 \mid \boldsymbol{x}, \boldsymbol{w}) = \frac{1}{2} \tag{3.57}$$

i.e. when,

$$\boldsymbol{w} \cdot \boldsymbol{x} = 0 \tag{3.58}$$

The decision boundary is shown in Figure 3.4.

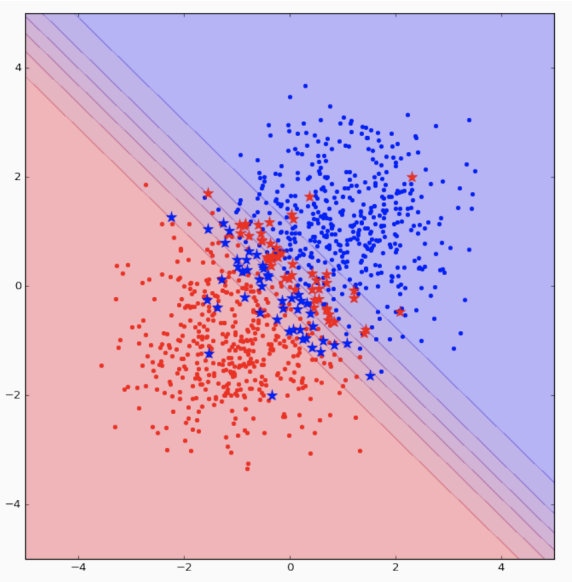

Figure 3.4: Logistic Regression Boundaries

As is shown in Figure 3.4, contour lines from bottom left to top right are for $\theta = 0.15, 0.3, 0.45, 0.6, 0.75, 0.9$ each respectively and the starred points represent mistakes made by the classifier.

Now, how can we estimate the parameters $\boldsymbol{w}$ for our logistic regression classifier?

Assume we have the dataset $\mathcal{D} = ((\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_N, y_N))$, where $\boldsymbol{x}_i \in \mathbb{R}^D$ and $y_i \in \{0, 1\}$.

The likelihood of the observing data, given the model parameters $\boldsymbol{w}$ is given by,

$$\begin{aligned} p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) &= \prod_{i=1}^{N} \sigma(\boldsymbol{w}^T \boldsymbol{x}_i)^{y_i} (1 - \sigma(\boldsymbol{w}^T \boldsymbol{x}_i))^{1-y_i} \\ &= \prod_{i=1}^{N} \mu_i^{y_i} (1 - \mu_i)^{1-y_i} \end{aligned} \tag{3.59}$$

where $\mu_i = \sigma(\boldsymbol{w}^T \boldsymbol{X}_i)$.

The negative log likelihood is given by,

$$NLL(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) = -\sum_{i=1}^{N} y_i \log \mu_i + (1 - y_i) \log(1 - \mu_i) \tag{3.60}$$

which is the cross-entropy between y_i and u_i for $y_i \in \{0, 1\}$.

The gradient of the negative log likelihood with respect to $\boldsymbol{w}$ is,

$$\nabla_{\boldsymbol{w}} NLL(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) = \sum_{i=1}^{N} \boldsymbol{x_i}(\mu_i - y_i) = \boldsymbol{X}^T(\boldsymbol{\mu} - \boldsymbol{y}) \tag{3.61}$$

The Hessian of the NLL is given as,

$$\boldsymbol{H} = \boldsymbol{X}^T \boldsymbol{S} \boldsymbol{X} \tag{3.62}$$

Where $\boldsymbol{S}$ is the diagonal matrix with entries $s_{ii} = \mu_i(1 - \mu_i)$,

$$\boldsymbol{S} = \begin{bmatrix} s_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & s_{nn} \end{bmatrix} = \begin{bmatrix} \mu_1(1-\mu_1) & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \mu_n(1-\mu_n) \end{bmatrix} \tag{3.63}$$

For $\forall \boldsymbol{z} \in \mathbb{R}^D$,

$$\begin{aligned} \boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} &= \boldsymbol{z}^T \boldsymbol{X}^T \boldsymbol{S} \boldsymbol{X} \boldsymbol{z} \\ &= (\boldsymbol{X}\boldsymbol{z})^T \boldsymbol{S} \boldsymbol{X} \boldsymbol{z} \\ &= \sum_{i=1}^{N} s_{ii} [\boldsymbol{X}\boldsymbol{z}]_i^2 \end{aligned} \tag{3.64}$$

Since $s_{ii} = \mu_i(1 - \mu_i) > 0$ for $\mu_i \in (0, 1)$,

$$\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} = \sum_{i=1}^{N} s_{ii} [\boldsymbol{X}\boldsymbol{z}]_i^2 > 0 \tag{3.65}$$

the Hessian of NLL is positive definite. Thus, NLL is convex and we can use convex optimization method to minimise NLL.

For small number D of dimensions, we can apply Newton's method to estimate $\boldsymbol{w}$.

Let $\boldsymbol{w}_t$ be the parameters after t Newton steps. The gradient and the Hessian are given by,

$$\begin{aligned} \boldsymbol{g}_t &= \boldsymbol{X}^T(\boldsymbol{\mu}_t - \boldsymbol{y}) = -\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{\mu}_t) \\ \boldsymbol{H}_t &= \boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X} \end{aligned} \tag{3.66}$$

The Newton updating rule is given by,

$$\begin{aligned} \boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \boldsymbol{H}_t^{-1} \boldsymbol{g}_t \\ &= \boldsymbol{w}_t + (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X})^{-1} \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{\mu}_t) \\ &= (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X})^{-1} (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X}) \boldsymbol{w}_t + (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X})^{-1} \boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{\mu}_t) \\ &= (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{S}_t^T [\boldsymbol{X}\boldsymbol{w}_t + \boldsymbol{S}_t^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_t)] \\ &= (\boldsymbol{X}^T \boldsymbol{S}_t \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{S}_t^T \boldsymbol{z}_t \\ &= (\boldsymbol{X}^T \boldsymbol{S}_t^{\frac{1}{2}} \boldsymbol{S}_t^{\frac{1}{2}} \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{S}_t^{\frac{1}{2}} \boldsymbol{S}_t^{\frac{1}{2}} \boldsymbol{z}_t \end{aligned} \tag{3.67}$$

Where $\boldsymbol{z}_t = \boldsymbol{X}\boldsymbol{w}_t + \boldsymbol{S}_t^{-1}(\boldsymbol{y} - \boldsymbol{\mu}_t)$.
Since the transpose of a diagonal matrix is itself,

$$\boldsymbol{w}_{t+1} = ((\boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{X})^T\boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{X})^{-1}(\boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{X})^T\boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{z}_t \tag{3.68}$$

Let $\tilde{\boldsymbol{X}} = \boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{X}$ and $\tilde{\boldsymbol{y}} = \boldsymbol{S}_t^{\frac{1}{2}}\boldsymbol{z}_t$, the equation above can be rephrased as,

$$\boldsymbol{w}_{t+1} = (\tilde{\boldsymbol{X}}^T\tilde{\boldsymbol{X}})^{-1}\tilde{\boldsymbol{X}}\tilde{\boldsymbol{y}} \tag{3.69}$$

The updating weight $\boldsymbol{w}_{t+1}$ is in the same form as the weight minimising the least square objective $\mathcal{L}(\boldsymbol{w}) = \sum_i(\boldsymbol{x}_i^T\boldsymbol{w} - y)^2$.

Thus, we can reconstruct the Newton step as the problem finding $\boldsymbol{w}$ that minimize the objective,

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}) &= \sum_i(\tilde{\boldsymbol{x}}_i^T\boldsymbol{w} - \tilde{y}_i)^2 \\ &= \sum_i[\boldsymbol{S}_{t,ii}^{\frac{1}{2}}(\boldsymbol{x}_i - z_{t,i})]^2 \\ &= \sum_i \boldsymbol{S}_{t,ii}(\boldsymbol{x}_i - z_{t,i})^2 \\ &= \boldsymbol{S}_t(\boldsymbol{X}\boldsymbol{w}_t - \boldsymbol{z}_t)^T(\boldsymbol{X}\boldsymbol{w}_t - \boldsymbol{z}_t)\end{aligned} \tag{3.70}$$

The objective above is called the weighted least squares of which we can use to compute $\boldsymbol{w}_{t+1}$ at each Newton step. The optimization method here is called the iteratively re-weighted least squares method.

- Each step requires re-weighting of the residual by a new diagonal matrix $\boldsymbol{S}_t$
- Each step uses a new vector $\boldsymbol{z}_t$ which depends on $\boldsymbol{w}_t$
- We proceed iteratively, one Newton step after another

3.2.2 Multi-Class Logistic Regression

Consider now $C > 2$ classes: $y \in \{1, \cdots, C\}$. There are parameters $\boldsymbol{w}_c$ for every class c which forms a parameter matrix $\boldsymbol{W} = [\boldsymbol{w}_1, \cdots, \boldsymbol{w}_C] \in \mathbb{R}^{D\times C}$.

The multi-class logistic regression model is given by,

$$\begin{aligned}p(y = c \mid \boldsymbol{x}, \boldsymbol{W}) &= \frac{e^{\boldsymbol{w}_c\boldsymbol{x}}}{\sum_{c'=1}^C e^{\boldsymbol{w}_{c'}\boldsymbol{x}}} \\ &= softmax([\boldsymbol{w}_1\boldsymbol{x}, \cdots, \boldsymbol{w}_C\boldsymbol{x}])\end{aligned} \tag{3.71}$$

The parameter estimation for multi-class logistic regression model is the same as in the case of binary logistic regression model. The negative log-likelihood(NLL) is still convex.

Actually, we can see binary logistic regression as a special case of multi-class logistic regression where $\boldsymbol{W} = [\boldsymbol{w}_0, \boldsymbol{w}_1] \in \mathbb{R}^{D \times 2}$,

$$softmax([\boldsymbol{w}_0, \boldsymbol{w}_1]) = \frac{e^{\boldsymbol{w}_1 \boldsymbol{x}}}{e^{\boldsymbol{w}_1 \boldsymbol{x}} + e^{\boldsymbol{w}_0 \boldsymbol{x}}} = \sigma((\boldsymbol{w}_1 - \boldsymbol{w}_0)\boldsymbol{x}) \tag{3.72}$$

3.3 Multi-class Classification and Measuring Performance

3.3.1 Multi-Class Classification

Multi-class classification refers to the problem where the number of classes $C > 2$. In practice, the following approaches are common.

The first one is called the One-vs-One approach. It trains $\frac{C(C-1)}{2}$ classifiers and each training procedure only uses on average $\frac{2}{C}$ of the training data. For a new input, the output is the most common classification for the classifiers.

The second approach is called the One-vs-Rest classifiers. It training C different classifiers, i.e. one class vs the rest $C-1$ classes and and each training procedure only uses the entire training data. It is typical for classifiers that yield class membership probability or scores. For a new input, it picks the class with the largest probability or score.

For example, if we have three classes c_1, c_2 and c_3, the One-vs-One approach trains $\frac{3\times(3-1)}{2} = 3$ classifiers which discriminate c_1 from c_2, c_1 from c_3, and c_2 from c_3 each respectively. The output of a new input is the majority vote of the prediction labels obtained on all classifiers. The One-vs-Rest approach, however, trains 3 classifiers: c_1 against c_2 and c_3; c_2 against c_1 and c_3; and c_3 against c_1 and c_2. The output of a new input is the class with largest probability or score. If we have ties, we can break it by taking the value of $\boldsymbol{w}\boldsymbol{x}_{new} + w_0$.

A more efficient method is to reduce multi-class classification problem to binary classification problem. It divides class into pairs of disjoint subsets and trains a binary classifier to separate the subsets of each pair. Then it uses an error correcting approach to determine the class label.

The classification error is defined by the number of misclassified data points. However, in classification problems, not all mistakes are equally problematic, for example, failing to detect the medical risk v.s. inaccurately predicting the chance of the risk. How can we solve this problem?

In logistic regression, we use threshold 0.5 to label a data point positive. If we want very few false positives, we can raise the threshold to 0.9, i.e. predicting something as positive only if it were 90% sure.

In generative models, we have decision boundaries when $\frac{p(y=1|\boldsymbol{x}_{new},\boldsymbol{w})}{p(y=0|\boldsymbol{x}_{new},\boldsymbol{w})} = 1$, i.e. all errors are treated equally. However, we can change the ratio if one type of the error is more costly than the other.

3.3.2 Measuring Performance Binary Classification

When measuring performance for binary classification, we use confusion matrix defined as,

And we define the following,

	Actual Labels	
Prediction	1	0
1	TN(True Positive)	FP(False Positive)
0	FN(False Negative)	TN(True Negative)

Table 3.1: Confusion Matrix

- True Positive Rate, i.e. Sensitivity, which is the ratio of true positive to actual positive, give as,

$$TPR = \frac{TP}{TP + FN} \tag{3.73}$$

- False Positive Rate, i.e. Fall-out, which is the ratio of false positive to actual negative, give as,

$$FPR = \frac{FP}{FP + TN} \tag{3.74}$$

- True Negative Rate, i.e. Specificity, which is the ratio of true negative to actual negative, give as,

$$TNR = \frac{TN}{FP + TN} = 1 - FPR \tag{3.75}$$

- Precision which is the ratio of true positives to predicted positives, given as,

$$P = \frac{TP}{TP + FP} \tag{3.76}$$

- Accuracy, given as,

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{3.77}$$

In practice, instead of analyzing many confusion matrices of different thresholds, we use receiver operating curves, or ROC curve, which gives an intuitive compact representation of all of them.

A receiver operating curve, or ROC curve, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied.

Usually, the x-axis and y-axis is FPR and TPR each respectively. It gives the relationship between the proportion of actual positive labels correctly classified to the proportion of actual negative labels incorrectly classified.

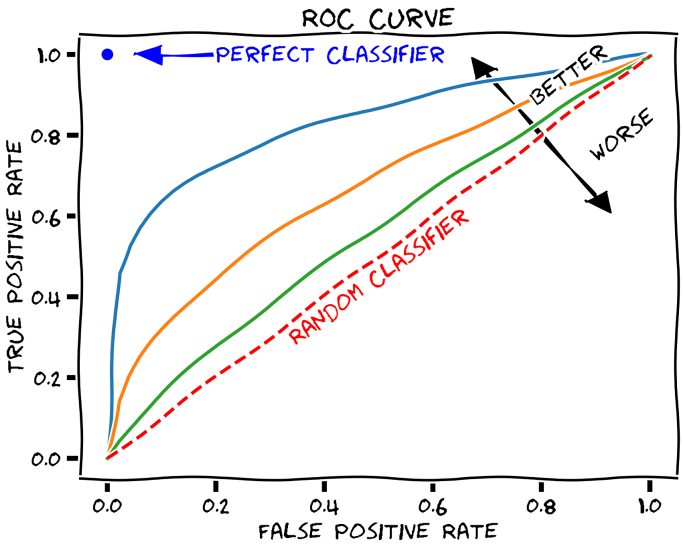

Figure 3.5: Receiver Operating Curve

The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes, i.e. the model is more preferred.

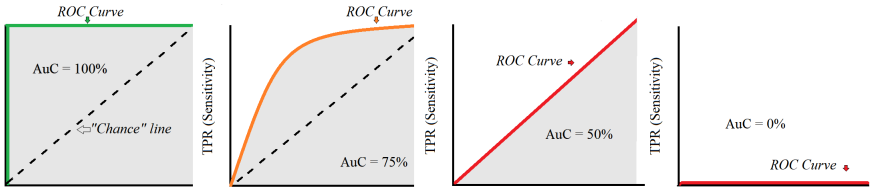

Figure 3.6: Area Under the Curve

Another metric beyond ROC curves is to replace FPR with Precision. Precision gives the proportion of positive results that were correctly classified. Precision is preferred over FPR in scenarios where there were lots of negative samples. Since precision does not include True Negatives, so it will not be affected by imbalance. For example, if we are studying a rare events, e.g., a rare disease. There are many more samples that do not observe the event than those that observe it.

3.3.3 Measuring Performance Multi-Class Classification

For multi-class classification, we generalise the confusion matrix defined as,

	Actual Labels			
Prediction	1	2	$\cdots$	K
1	N_{11}	N_{12}	$\cdots$	N_{1K}
2	N_{21}	N_{22}	$\cdots$	N_{2K}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
K	N_{K1}	N_{K2}	$\cdots$	N_{KK}

Table 3.2: Confusion Matrix

Where $N_{i,j}$ is the items of class j in the dataset that are predicted to be of class i.

A good multi-class classifier should have large diagonal entries and small off-diagonal entries, i.e. such that,

$$\arg\max_{\boldsymbol{\theta}} \quad \mathrm{tr}(\boldsymbol{C}) \tag{3.78}$$

Where $\mathrm{tr}(\boldsymbol{C})$ is the trace of the confusion matrix.

3.4 Support Vector Machines

SVM is a popular discriminative model for classification. However, it is not a natural probabilistic interpretation.

Recall binary classification, our goal is to : find a linear separator. Suppose our data is linearly separable if there exists a linear separator that classifies all points correctly, as is shown in Figure 3.7. Which separator should be picked?

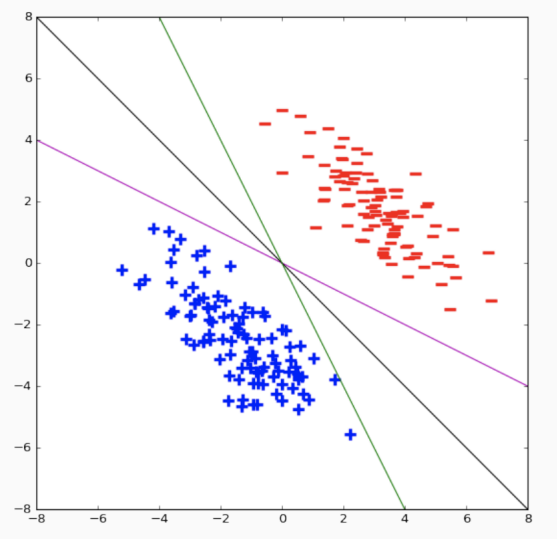

Figure 3.7: Linear Separators

3.4.1 Maximum Margin Principle

The margin for a data point refers to its distance to the separating hyperplane.

The maximum margin principle is to pick the separating boundary that maximises the smallest margin, i.e. the least distance between data and boundary. That is, to maximise the distance of the closest point from the decision boundary. The points that are closest to the decision boundary are called support vectors, as is shown in Figure 3.8.

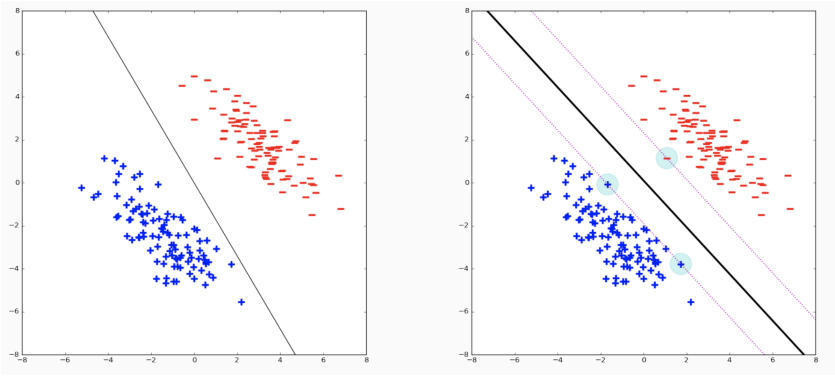

Figure 3.8: Support Vectors

Given a hyper-plane $H \equiv \boldsymbol{w}\boldsymbol{x} + w_0 = 0$ and a point $\boldsymbol{x} \in \mathbb{R}^D$, the distance of point $\boldsymbol{x}$ from H is given by,

$$\frac{|\boldsymbol{w}\boldsymbol{x} + w_0|}{\|\boldsymbol{w}\|_2} \tag{3.79}$$

All points on the positive half-space created by H, labeled as $y = +1$, satisfy,

$$\boldsymbol{w}\boldsymbol{x} + w_0 > 0 \tag{3.80}$$

All points on the negative half-space created by H, labeled as $y = -1$, satisfy,

$$\boldsymbol{w}\boldsymbol{x} + w_0 < 0 \tag{3.81}$$

Our goal of maximising the distance of the closest point, i.e. support vectors, from the decision boundary can be rephrased as finding the distance $\|\boldsymbol{x} - \boldsymbol{x}^*\|$ between the point $\boldsymbol{x}^*$ and the hyper-plane $\boldsymbol{w}\boldsymbol{x} + w_0 = 0$, where the point $\boldsymbol{x}$ on the hyper-plane that is closest to $\boldsymbol{x}^*$ gives the distance.

Equivalently, we seek for $\boldsymbol{x}$ that optimises the following problem,

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 \\ \text{s.t.} \quad & \boldsymbol{w}\boldsymbol{x} + w_0 = 0 \end{aligned} \tag{3.82}$$

We can solve this problem using the Lagrangian multiplier, given by,

$$\begin{aligned} \Lambda &= \|\boldsymbol{x} - \boldsymbol{x}^*\|_2^2 - 2\lambda(\boldsymbol{w}\boldsymbol{x} + w_0) \\ &= \|\boldsymbol{x}\|_2^2 - 2\boldsymbol{x}(\boldsymbol{x}^* + \lambda\boldsymbol{w}) + 2\lambda w_0 + \|\boldsymbol{x}^*\|_2^2 \end{aligned} \tag{3.83}$$

Setting its gradient,

$$\nabla_{\boldsymbol{x}}\Lambda(\boldsymbol{x}, \lambda) = 2\boldsymbol{x} - 2(\boldsymbol{x}^* + \lambda\boldsymbol{w}) = 0 \tag{3.84}$$

to 0, we obtained the critical point,

$$\boldsymbol{x} = \boldsymbol{x}^* + \lambda \boldsymbol{w} \tag{3.85}$$

By substituting $\boldsymbol{x}$ into the hyper-plane equation, we obtain,

$$\lambda = -\frac{\boldsymbol{w}\boldsymbol{x}^* + w_0}{\|\boldsymbol{w}\|_2^2} \tag{3.86}$$

Then, the distance between $\boldsymbol{x}^*$ and $\boldsymbol{x}$ is given by,

$$\begin{aligned} \|\boldsymbol{x} - \boldsymbol{x}^*\|_2 &= \|\boldsymbol{x}^* + \lambda \boldsymbol{w} - \boldsymbol{x}^*\|_2 \\ &= \|\lambda \boldsymbol{w}\|_2 \\ &= |\lambda| \|\boldsymbol{w}\|_2 \\ &= \frac{|\boldsymbol{w}\boldsymbol{x}^* + w_0|}{\|\boldsymbol{w}\|_2} \end{aligned} \tag{3.87}$$

Assume that the dataset $\mathcal{D} = ((\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_N, y_N))$ is linearly separable. For $y_i = +1$ we have $\boldsymbol{w}\boldsymbol{x}_i + w_0 > 0$ and for $y_i = -1$ we have $\boldsymbol{w}\boldsymbol{x}_i + w_0 < 0$. Thus, most compactly, for all points in the dataset $\mathcal{D}$ we have,

$$y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) > 0 \tag{3.88}$$

Since the dataset $\mathcal{D}$ is finite, we have a small enough number ϵ that satisfies,

$$y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq \epsilon \tag{3.89}$$

Dividing ϵ on both sides, we obtain,

$$y_i(\frac{\boldsymbol{w}}{\epsilon}\boldsymbol{x}_i + \frac{w_0}{\epsilon}) \geq 1 \tag{3.90}$$

Let $\hat{\boldsymbol{w}} = \frac{\boldsymbol{w}}{\epsilon}$ and $\hat{w_0} = \frac{w_0}{\epsilon}$, we have,

$$y_i(\hat{\boldsymbol{w}}\boldsymbol{x}_i + \hat{w_0}) \geq 1 \tag{3.91}$$

To simplify the notation, we have,

$$y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 1 \tag{3.92}$$

Recall that the margin of data point $\boldsymbol{x}^*$ to hyper-plane is $\frac{|\boldsymbol{w}\boldsymbol{x}^* + w_0|}{\|\boldsymbol{w}\|_2}$. Thus, we have,

$$\frac{|\boldsymbol{w}\boldsymbol{x}^* + w_0|}{\|\boldsymbol{w}\|_2} = \frac{y^*(\boldsymbol{w}\boldsymbol{x}^* + w_0)}{\|\boldsymbol{w}\|_2} \geq \frac{1}{\|\boldsymbol{w}\|_2} \tag{3.93}$$

Since the maximum margin principle is to pick the boundary that maximises the margin, we have the optimization problem given by,

$$\begin{aligned} \text{maximize} \quad & \frac{1}{\|\boldsymbol{w}\|_2} \\ \text{s.t.} \quad & y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 1, \quad i \in [N] \end{aligned} \tag{3.94}$$

Equivalently, the optimization problem can be rephrased as,

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 1, \quad i \in [N] \end{aligned} \tag{3.95}$$

The constant factor $\frac{1}{2}$ is added without loss of generality.

We can easily observe that the objective is a convex quadratic function and the constraints are convex linear functions. Also, the feasible set as defined by the constraints is convex as well. Thus, our optimisation problem is convex quadratic which can be solved using generic convex optimisation methods.

If our data is linearly-separable, we will find the SVM classifier with no classification error on the training set.

However, if our data is not linearly-separable, as is shown in Figure 3.9, the quadratic program from the previous slides has no feasible solution.

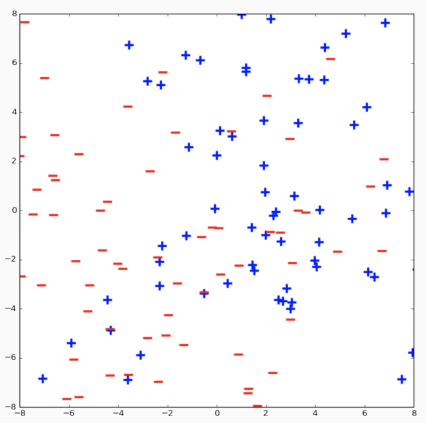

Figure 3.9: Linearly Non-separable Data

We can solve this problem by adding a relaxation term to the previous optimisation problem and find a separator that makes the least mistakes on the training error. However, minimising the number of misclassifications is NP-hard. Equivalently, we can use a proxy for least mistakes, that is to satisfy as many of the N constraints as possible.

Alternatively, we allow all constraints to be satisfied with some slack, given by,

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_{i=1}^{N}\zeta_i \\ \text{s.t.} \quad & y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 1 - \zeta_i, \quad i \in [N] \\ & \zeta_i \geq 0 \quad i \in [N] \end{aligned} \tag{3.96}$$

Feasible solution to the optimization problem always exists thanks to slack variables ζ_i and ζ_i can be as large, i.e. constraints can be violated as much, as necessary.

Why we have $\zeta_i \geq 0$?

Assume $\zeta_i \ll 0$ and $\boldsymbol{x}_i$ is correctly classified by a huge margin. This would compensate the penalty incurred on misclassified points. That is to say, positive ζ_i which is induced by misclassified points and negative ζ_i which is induced by correctly classified points will cancel each other out. Thus, we need $\zeta_i \geq 0$ to ensure there is no bonus for correct classification by a huge margin.

3.4.2 Hinge Loss Optimization

The optimal solution to the optimization problem above must satisfy either $\zeta_i = 0$, which is the ideal case, where no slack is needed to classify $\boldsymbol{x}_i$ or that ζ_i is the minimum which satisfies the constraint $\zeta_i = 1 - y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 0$. Equivalently,

$$\zeta_i = \max(0, 1 - y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0)) \stackrel{def}{=} \ell_{hinge}(\boldsymbol{w}, w_0; \boldsymbol{x}_i, y_i) \tag{3.97}$$

Our SVM formulation becomes equivalent to minimising the objective function,

$$\mathcal{L}_{SVM} = \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_{i=1}^{N}\ell_{hinge}(\boldsymbol{w}, w_0; \boldsymbol{x}_i, y_i) \tag{3.98}$$

which is the Hinge loss with ℓ_2 regularization.

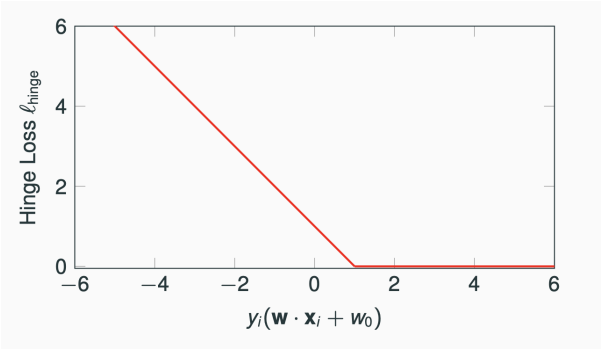

Figure 3.10: Hinge Loss

Recall the the logistic loss function, we can see some similarities between them as is shown in Figure 3.11.

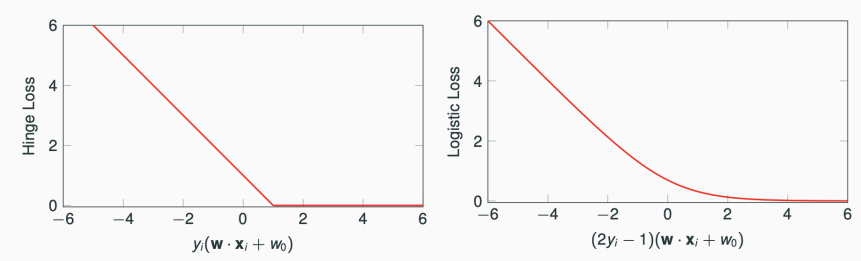

Figure 3.11: Hinge Loss v.s. Logistic Loss

3.4.3 Dual SVM Formulation

Constrained Optimisation with Inequalities

Recall the primary form of the optimization problem

$$\begin{array}{ll} \min & F(\boldsymbol{z}) \\ \text{s.t.} & g_i(\boldsymbol{z}) \geq 0, \quad i = 1, \ldots, m \\ & h_j(\boldsymbol{z}) = 0, \quad j = 1, \ldots, \ell \end{array} .$$

The Lagrange is

$$\Lambda(\boldsymbol{z}; \alpha, \mu) = F(\boldsymbol{z}) - \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{z}) - \sum_{j=1}^{\ell} \mu_j h_j(\boldsymbol{z})$$

For convex problems, Karush-Kuhn-Tucker (KKT) conditions:

- Dual feasibility:

$$\alpha_i > 0 \quad \text{for} \quad i = 1, \ldots, m$$

- Primal feasibility:

$$g_i(\boldsymbol{z}) \geq 0 \quad \text{for} \quad i = 1, \ldots, m$$

$$h_j(\boldsymbol{z}) = 0 \quad \text{for} \quad j = 1, \ldots, \ell$$

- Complementary slackness

$$\alpha_i g_i(\boldsymbol{z}) = 0 \quad \text{for} \quad i = 1, \ldots, m$$

are necessary and sufficient for a critical point of Λ to be the minimum of the original constrained optimisation.

For non-convex problems: KKT are necessary but not sufficient.

Dual SVM Formulation: The Linearly Separable Case

Recall our optimization problem in SVM.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\|\boldsymbol{w}\|_2^2 \\ \text{s.t.} \quad & y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) \geq 1, \quad i \in [N] \end{aligned}$$

The Lagrange function is

$$\Lambda(\boldsymbol{w}; w_0, \alpha) = \frac{1}{2}\|\boldsymbol{w}\|_2^2 - \sum_{i=1}^{N} \alpha_i (y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) - 1)$$

and its gradients with respect to the primal variables are

$$\frac{\partial \Lambda}{\partial w_0} = -\sum_{i=1}^{N} \alpha_i y_i$$

$$\nabla_{\boldsymbol{w}} = \boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

At optimality, these gradients are 0, yielding

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

$$\boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i = 0$$

Optimal $\boldsymbol{w}$ obtained as

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

At optimality

$$y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) = 1 \quad \text{for} \quad i \in [N]$$

We can obtain w_0 using the above equality constraint for any i:

$$y_i^2(\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j \cdot \boldsymbol{x}_i + w_0) = y_i$$

Since $y_i^2 = (\pm 1)^2 = 1$,

$$\sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j \cdot \boldsymbol{x}_i + w_0 = y_i$$

Thus,

$$w_0 = y_i - \sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j \cdot \boldsymbol{x}_i$$

It is numerically more stable to have w0 the average over all possible values:

$$w_0 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \sum_{j=1}^{N} \alpha_j y_j \boldsymbol{x}_j \cdot \boldsymbol{x}_i)$$

Next we plug the optimal $\boldsymbol{w}$ and constraint

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

into Lagrangian.

$$\begin{aligned}
\Lambda(\boldsymbol{w}; w_0, \alpha) &= \frac{1}{2}\|\boldsymbol{w}\|_2^2 - \sum_{i=1}^{N} \alpha_i (y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) - 1) \\
&= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{w}\boldsymbol{x}_i - \sum_{i=1}^{N} \alpha_i y_i w_0 + \sum_{i=1}^{N} \alpha_i \\
&= \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j (\boldsymbol{x}_i \boldsymbol{x}_j) - \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j (\boldsymbol{x}_i \boldsymbol{x}_j) + \sum_{i=1}^{N} \alpha_i \\
&= \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j (\boldsymbol{x}_i \boldsymbol{x}_j) \\
&= g(\alpha)
\end{aligned} \tag{3.99}$$

To find critical points of Λ satisfying the KKT conditions, it is sufficient to find the critical points of g that satisfy the constraints:

$$\alpha_i \geq 0$$

and

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

Thus, the dual form SVM optimization problem becomes:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j (\boldsymbol{x}_i \boldsymbol{x}_j) \\ \text{s.t.} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \quad \text{and} \quad \alpha_i \geq 0 \quad \text{for} \quad i \in [N] \end{aligned}$$

This is quadratic concave problem with N variables and very simple box constraints plus one zero-sum constraint.

There are several reasons why SVM dual formulation is preferred.

- $D \approx D$ or D: Basis expansion to capture non-linear discriminating boundaries
- Natural kernelisation: Replace dot product $\boldsymbol{x}_i \boldsymbol{x}_j$ by kernel $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)$
- The number of non-zero variables α_i can be much smaller in practice. The complementary slackness conditions in KKT

  $$\alpha_i (y_i (\boldsymbol{w} \boldsymbol{x}_i + w_0) - 1) = 0$$

  are satisfied when $\alpha_i = 0$ or $y_i(\boldsymbol{w}\boldsymbol{x}_i + w_0) = 1$. Thus, only the points $\boldsymbol{x}_i$ with $\alpha_i > 0$ contribute to the solution $\boldsymbol{w}$

  $$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

  That is to say such vectors $\boldsymbol{x}_i$ form the support of the solution, i.e. are support vectors. Such vectors $\boldsymbol{x}_i$ are points with least margin.

Dual SVM Formulation: The Non-Linearly Separable Case

Recall the primary objective in the non-linearly separable case.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2} \|\boldsymbol{w}\|_2^2 + C \sum_{i=1}^{N} \zeta_i \\ \text{s.t.} \quad & y_i (\boldsymbol{w} \boldsymbol{x}_i + w_0) \geq 1 - \zeta_i, \quad i \in [N] \\ & \zeta_i \geq 0 \quad i \in [N] \end{aligned} \tag{3.100}$$

The Lagrange is

$$\Lambda(\boldsymbol{w}; w_0, \alpha) = \frac{1}{2} \|\boldsymbol{w}\|_2^2 + C \sum_{i=1}^{N} \zeta_i - \sum_{i=1}^{N} \alpha_i (y_i (\boldsymbol{w} \boldsymbol{x}_i + w_0) - (1 - \zeta_i)) - \sum_{i=1}^{N} \mu_i \zeta_i$$

The derivatives with respect to $\boldsymbol{w}$, w_0 and ζ_i are:

$$\nabla_{\boldsymbol{w}}\Lambda = \boldsymbol{w} - \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

$$\frac{\partial \Lambda}{\partial w_0} = -\sum_{i=1}^{N} \alpha_i y_i$$

$$\frac{\partial \Lambda}{\partial \zeta_i} = C - \alpha_i - \mu_i$$

For (KKT) dual feasibility constraints, we require $\alpha \geq 0$ and $\mu_i \geq 0$.

Setting the derivatives to 0, substituting the resulting expressions in Λ (and simplifying), we get the optimization problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i y_i \alpha_j y_j (\boldsymbol{x}_i \boldsymbol{x}_j) \\ \text{s.t.} \quad & \sum_{i=1}^{N} \alpha_i y_i = 0 \quad \text{and} \quad 0 \leq \alpha_i \leq C \quad \text{for} \quad i \in [N] \end{aligned}$$

Making Predictions using SVM Dual

Recall the optimal solution for $\boldsymbol{w}$ expressed using the dual variables $\boldsymbol{\alpha}$

$$\boldsymbol{w} = \sum_{i=1}^{N} \alpha_i y_i \boldsymbol{x}_i$$

The bias w_0 is also expressible using $\boldsymbol{\alpha}$.

Prediction on a new point $\boldsymbol{x}_{new}$ requires inner products with the support vectors

$$\boldsymbol{w}\boldsymbol{x}_{new} = \sum_{i=1}^{N} \alpha_i y_i (\boldsymbol{x}_i \boldsymbol{x}_{new})$$

We can as well using blackbox access to a function $\kappa(\cdot, \cdot)$ that maps two inputs $\boldsymbol{x}$, $\boldsymbol{x}'$ to their inner product $\boldsymbol{x} \cdot \boldsymbol{x}'$. This is a kernel function

3.4.4 Kernel Methods

A function κ is a kernel that computes the dot product for some expansion ϕ:

$$\kappa(\boldsymbol{x'}, \boldsymbol{x}) : \chi \times \chi \to \mathbb{R}, \quad \kappa(\boldsymbol{x'}, \boldsymbol{x}) = \phi(\boldsymbol{x'})\phi(\boldsymbol{x})$$

$\kappa(\boldsymbol{x'}, \boldsymbol{x})$ is some measure of similarity between $\boldsymbol{x}$ and $\boldsymbol{x'}$.

Mercer Kernels

The gram matrix of a kernel is defined as,

$$\boldsymbol{K} = \begin{pmatrix} \kappa(\boldsymbol{x}_1, \boldsymbol{x}_1) & \kappa(\boldsymbol{x}_1, \boldsymbol{x}_2) & \cdots & \kappa(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \kappa(\boldsymbol{x}_2, \boldsymbol{x}_1) & \kappa(\boldsymbol{x}_2, \boldsymbol{x}_2) & \cdots & \kappa(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\boldsymbol{x}_N, \boldsymbol{x}_1) & \kappa(\boldsymbol{x}_N, \boldsymbol{x}_2) & \cdots & \kappa(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{pmatrix} \tag{1}$$

A kernel is Mercer or positive definite kernel if the Gram matrix is always positive semi-definite.

Kernel

$$\kappa(\boldsymbol{x'}, \boldsymbol{x}) = \boldsymbol{x'}^T \boldsymbol{x}$$

is a Mercer kernel.

Kernel Engineering

We can build kernels using simpler kernels as building blocks.

Given kernels κ_1, κ_2, the following are kernels:

- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = c\kappa_1(\boldsymbol{x'}, \boldsymbol{x})$ where $c > 0$
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = f(\boldsymbol{x'})\kappa_1(\boldsymbol{x'}, \boldsymbol{x})f(\boldsymbol{x})$ where f is a function.
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = q[\kappa_1(\boldsymbol{x'}, \boldsymbol{x})]$ where q is a polynomial with non-negative coefficients.
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = e^{\kappa_1(\boldsymbol{x'}, \boldsymbol{x})}$
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = \kappa_1(\boldsymbol{x'}, \boldsymbol{x})\kappa_2(\boldsymbol{x'}, \boldsymbol{x})$
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = \kappa_a(\boldsymbol{x'}_a, \boldsymbol{x}_a)\kappa_b(\boldsymbol{x'}_b, \boldsymbol{x}_b)$ where $\boldsymbol{x'} = (\boldsymbol{x'}_a, \boldsymbol{x'}_b)$, $\boldsymbol{x} = (\boldsymbol{x}_a, \boldsymbol{x}_b)$, κ_a, κ_b are kernels.
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = \kappa_a(\boldsymbol{x'}_a, \boldsymbol{x}_a) + \kappa_b(\boldsymbol{x'}_b, \boldsymbol{x}_b)$
- $\kappa(\boldsymbol{x'}, \boldsymbol{x}) = \boldsymbol{x'}^T \boldsymbol{A} \boldsymbol{x}$ where $\boldsymbol{A}$ is a symmetric positive semi-definite matrix.

For example, we can use kernel engineering to show that RBF kernel κ_{RBF} is kernel by:

$$\begin{aligned} \kappa_{RBF} = e^{-\frac{\|\boldsymbol{x}-\boldsymbol{x'}\|^2}{2\sigma^2}} &= e^{\frac{-\boldsymbol{x}^T\boldsymbol{x}}{2\sigma^2}} e^{\frac{\boldsymbol{x}^T\boldsymbol{x'}}{\sigma^2}} e^{\frac{-\boldsymbol{x'}^T\boldsymbol{x'}}{2\sigma^2}} \\ &= f(\boldsymbol{x}) e^{c\kappa_1(\boldsymbol{x'}, \boldsymbol{x})} f(\boldsymbol{x'}) \end{aligned} \tag{3.101}$$

4

Neural Network

4.1 Feed-forward Neural Networks

A perceptron is an algorithm for supervised learning of binary classifiers. It has an input of a feature vector $\boldsymbol{x}$ and a label y and outputs $\text{sign}(\boldsymbol{w}\boldsymbol{x} + b)$, where $\boldsymbol{w}$ is the weight parameter vector and b is the bias.

An artificial neuron is the generalisation of the perceptron. For any activation function $f : \mathbb{R} \to \mathbb{R}$, it has an input of a feature vector $\boldsymbol{x}$ and a label y and outputs $f(\boldsymbol{w}\boldsymbol{x} + b)$.

An artificial neuron in a neural network computes a linear function of its input and is then composed with a non-linear activation function, as is shown in Figure 4.1.

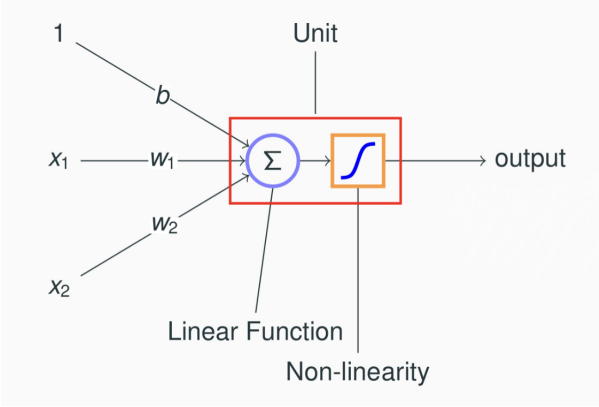

Figure 4.1: Neural Network with One Artificial Neuron

The output for regression is $\hat{y} = \mathbb{E}[y \mid \boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}]$ and the output for classification is $\hat{y} = Pr(y = 1 \mid \boldsymbol{x}, \boldsymbol{w}, \boldsymbol{b})$.

For logistic regression, the non-linear activation function is the sigmoid

function,

$$\sigma(z) = \frac{1}{1 + e^{-1}} \tag{4.1}$$

and the separating surface is linear.

An artificial neural network refers to the composition of artificial neurons into a network, where the artificial neurons have continuous activation functions and the network has a well-chosen loss function that, for any input-output, is differentiable.

Artificial neural networks are typically composed of input layers, hidden layers and output layers. A hidden layer is located between the input and output of the algorithm, in which the function applies weights to the inputs and directs them through an activation function as the output. In short, the hidden layers perform nonlinear transformations of the inputs entered into the network.

In theory, the neural networks can compute any function that a computer can.

A feed-forward network is a network with no recurrent connections. It is an important distinction because in a feed forward network the gradient is clearly defined and computable through backpropagation (i.e. chain rule).

A fully-connected network, or a fully-connected layer in a network, is one such that neurons are organised in a layered directed-acyclic graph, i.e. no loops. This, for example, contrasts with convolutional layers, where each output neuron depends on a subset of the input neurons.

Figure 3.2 shows a fully-connected feed-forward network.

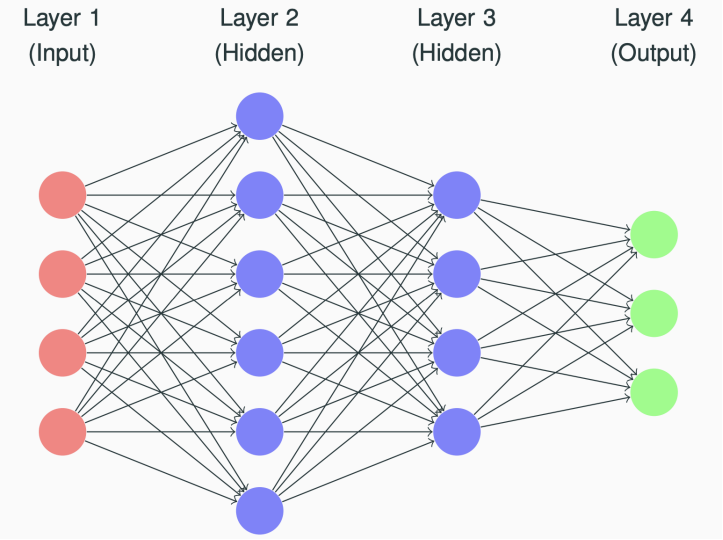

Figure 4.2: Fully-connected Feed-forward Network

4.1.1 Multi-layer Perceptrons

Multi-layer perceptron(MLP) is a type of feed-forward networks. Though it is called multi-layer perceptron, the units composed of it are not necessarily perceptrons.

Figure 4.3 shows a three-layered MLP. It first maps the feature vector $\boldsymbol{x}$ to non-linear features and then applies logistic regression to compute $\hat{y}$.

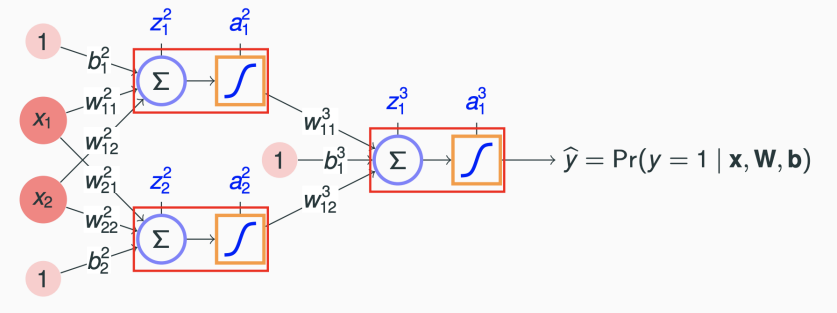

Figure 4.3: Three-Layered MLP

Let's first look at some notations used in nerual networks.

- w_{ij}^{ℓ}: weight for the connection to neuron i in layer ℓ from neuron j in layer $\ell - 1$
- b_i^{ℓ}: bias term for neuron i in layer in layer ℓ
- z_i^{ℓ}: pre-activation for neuron i in layer in layer ℓ, i.e. sum of weighted outputs from layer $\ell - 1$
- a_i^{ℓ}: activation for neuron i in layer in layer ℓ, i.e. non-linear function of pre-activation z_i^{ℓ}

Consider the three-layered MLP shown in Figure 4.3

In layer 1, i.e. the input layer, we have pre-activation = activation = input feature vector, given as,

$$\boldsymbol{a}^1 = \boldsymbol{z}^1 = \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \tag{4.2}$$

In layer 2, i.e. the hidden layer, we have the weight matrix:

$$\boldsymbol{W}^2 = \begin{pmatrix} w_{11}^2 & w_{12}^2 \\ w_{21}^2 & w_{22}^2 \end{pmatrix} \tag{4.3}$$

bias vector:

$$\boldsymbol{b}^2 = \begin{pmatrix} b_1^2 \\ b_2^2 \end{pmatrix} \tag{4.4}$$

pre-activation:

$$\boldsymbol{z}^2 = \begin{pmatrix} z_1^2 \\ z_2^2 \end{pmatrix} = \boldsymbol{W}^2\boldsymbol{a}^1 + \boldsymbol{b}^2 = \begin{pmatrix} w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2 \\ w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2 \end{pmatrix} \tag{4.5}$$

and activation:

$$\boldsymbol{a}^2 = \begin{pmatrix} a_1^2 \\ a_2^2 \end{pmatrix} = \tanh(\boldsymbol{z}^2) = \begin{pmatrix} \tanh(w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2) \\ \tanh(w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2) \end{pmatrix} \tag{4.6}$$

In layer 3, i.e. the output layer, we have the weight matrix:

$$\boldsymbol{W}^3 = \begin{pmatrix} w_{11}^3 & w_{12}^3 \end{pmatrix} \tag{4.7}$$

bias vector:

$$\boldsymbol{b}^3 = \begin{pmatrix} b_1^3 \end{pmatrix} \tag{4.8}$$

preactivation:

$$\boldsymbol{z}^3 = \begin{pmatrix} z_1^3 \end{pmatrix} = \boldsymbol{W}^3\boldsymbol{a}^2 + \boldsymbol{b}^3 = \begin{pmatrix} w_{11}^3 a_1^2 + w_{12}^3 a_2^2 + b_1^3 \end{pmatrix} \tag{4.9}$$

and activation:

$$\boldsymbol{a}^3 = \begin{pmatrix} a_1^3 \end{pmatrix} = \sigma(\boldsymbol{z}^3) = \begin{pmatrix} \sigma(w_{11}^3 a_1^2 + w_{12}^3 a_2^2 + b_1^3) \end{pmatrix} \tag{4.10}$$

The three-layered MLP gives the output:

$$\hat{y} = \boldsymbol{a}^3 = \sigma(\boldsymbol{z}^3) \tag{4.11}$$

Suppose we have the data shown in Figure 4.4 and apply logistic regression to classifier these data points into two classes: red and blue points.

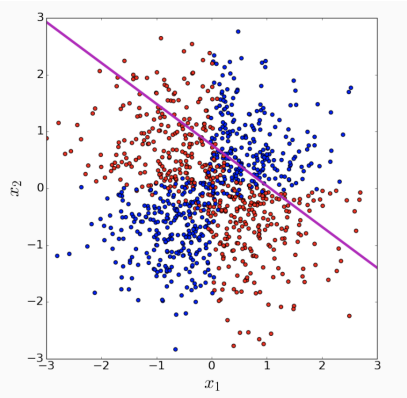

Figure 4.4: Data with Logistic Regression

We can see that the data is not linearly-separable, i.e. no linear separator can separate the blue from the red. Thus, the accuracy of logistic regression $\approx 50\%$.

If we apply our three-layered MPL to classify the data and scatter plot the input feature and the activation of the hidden layer, i.e. layer 2, the results are shown in Figure 4.5.

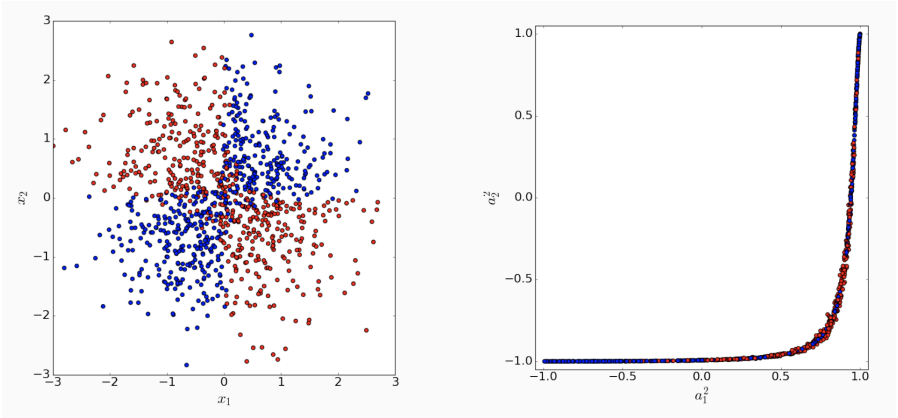

Figure 4.5: Input Feature $\boldsymbol{x}$ v.s. The Activation of the Hidden Layer $\boldsymbol{a}^2$

We can see that what the hidden do is to Mapping input features $\boldsymbol{x}$ to non-linear features $\boldsymbol{a}^2$, where

$$\boldsymbol{a}^2 = \begin{pmatrix} a_1^2 \\ a_2^2 \end{pmatrix} = \tanh(\boldsymbol{W}^2\boldsymbol{a}^1 + \boldsymbol{b}^2) = \begin{pmatrix} \tanh(w_{11}^2 x_1 + w_{12}^2 x_2 + b_1^2) \\ \tanh(w_{21}^2 x_1 + w_{22}^2 x_2 + b_2^2) \end{pmatrix} \quad (4.12)$$

The non-linear features $\boldsymbol{a}^2$ after the mapping becomes linearly-separable. We can find a linear separator to separate them, as is shown in Fugure 4.6.

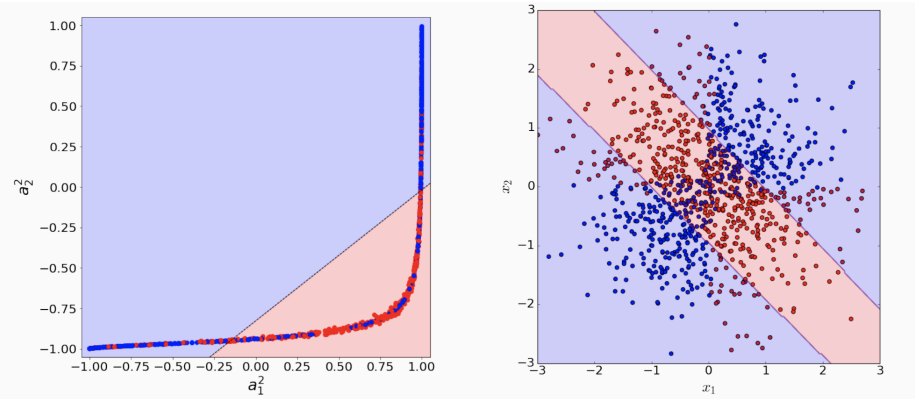

Figure 4.6: Data after Mapping with Linear Separator

We can easily see that the linear separator in the (a_1, a_2) plane becomes non-linear in the (x_1, x_2) plane.

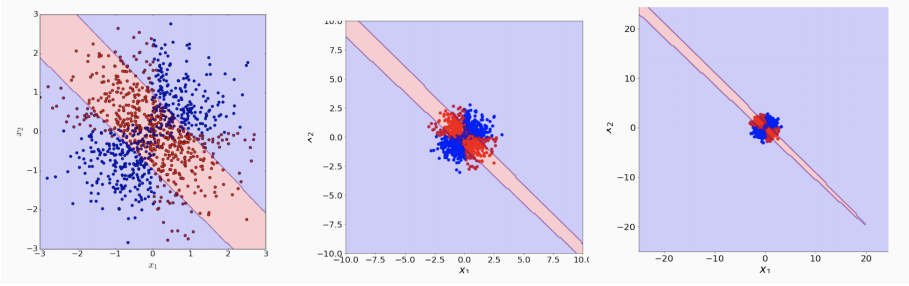

Figure 4.7: Separator in (x_1, x_2) Plane

4.1.2 Back-Propagation

Now let's consider how to learn the MLP parameters.

We first compute the derivatives of the loss function with respect to all parameters of the MLP for each data point. Then, we average the derivatives over the training data points. Since this can be computationally very expensive in practice, we commonly only average over a mini-batch instead of the entire dataset. Finally, we perform a gradient descent step to update the model parameters.

Recall the chain rule for multivariate calculus.

Suppose we have functions: $f : \mathbb{R}^n \to \mathbb{R}^k$, $g : \mathbb{R}^k \to \mathbb{R}^m$, $h = f \circ g : \mathbb{R}^n \to \mathbb{R}^m$. For $\boldsymbol{x} \in \mathbb{R}^n$, let $\boldsymbol{z} = f(\boldsymbol{x})$,

$$\frac{\partial h}{\partial \boldsymbol{x}} = \frac{\partial h}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} \tag{4.13}$$

where $\frac{\partial h}{\partial \boldsymbol{x}}$ is an $m \times n$ Jacobian matrix, $\frac{\partial h}{\partial \boldsymbol{z}}$ is an $m \times k$ Jacobian matrix and $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is an $k \times n$ Jacobian matrix.

Figure 4.8 shows an example of a three-layered MLP.

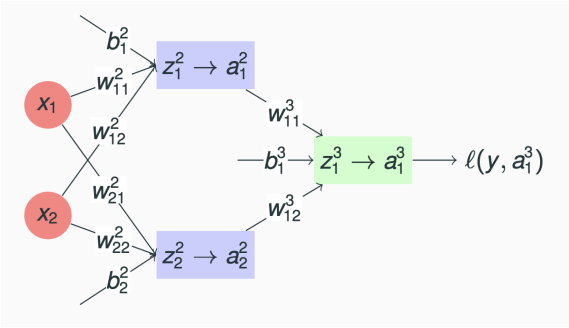

Figure 4.8: Separator in (x_1, x_2) Plane

We want the derivatives of the loss function with respect to the weights $\frac{\partial \ell}{\partial \boldsymbol{W}^\ell}$ and bias $\frac{\partial \ell}{\partial \boldsymbol{b}^\ell}$ in each layer, i.e. $\frac{\partial \ell}{\partial \boldsymbol{W}^2}$, $\frac{\partial \ell}{\partial \boldsymbol{b}^2}$, $\frac{\partial \ell}{\partial \boldsymbol{W}^3}$, and $\frac{\partial \ell}{\partial \boldsymbol{b}^3}$. It suffices to compute the derivatives of the loss function with respect to the activation $\frac{\partial \ell}{\partial \boldsymbol{z}^\ell}$ at each layer and then derive $\frac{\partial \ell}{\partial \boldsymbol{W}^\ell}$ and $\frac{\partial \ell}{\partial \boldsymbol{b}^\ell}$ from $\frac{\partial \ell}{\partial \boldsymbol{z}^\ell}$ using the chain rule.

Recall for a vector $\boldsymbol{z} \in \mathbb{R}^n$ and a function $f : \mathbb{R}^n \to \mathbb{R}$, the derivative $\frac{\partial f}{\partial \boldsymbol{z}} \in \mathbb{R}^n$ is a row vector defined as,

$$\frac{\partial f}{\partial \boldsymbol{z}} = \begin{pmatrix} \frac{\partial f}{\partial z_1} & \cdots & \frac{\partial f}{\partial z_n} \end{pmatrix} \tag{4.14}$$

For a matrix $\boldsymbol{W} \in \mathbb{R}^{n \times m}$ and a function $f : \mathbb{R}^{n \times m} \to \mathbb{R}$, the derivative $\frac{\partial f}{\partial \boldsymbol{W}} \in \mathbb{R}; n \times m$ is given as,

$$\frac{\partial f}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial f}{\partial w_{11}} & \cdots & \frac{\partial f}{\partial w_{1m}} \\ \vdots & \vdots & \vdots \\ \frac{\partial f}{\partial w_{n1}} & \cdots & \frac{\partial f}{\partial w_{nm}} \end{pmatrix}_{n \times m} \tag{4.15}$$

Let's now compute the derivatives in our three-layered MLP.

The derivatives of the loss function with respect to the weights in layer 2 is given by,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^2} = \begin{pmatrix} \frac{\partial \ell}{\partial w_{11}^2} & \frac{\partial \ell}{\partial w_{12}^2} \\ \frac{\partial \ell}{\partial w_{21}^2} & \frac{\partial \ell}{\partial w_{22}^2} \end{pmatrix}_{2 \times 2} \tag{4.16}$$

Assume that $\frac{\partial \ell}{\partial \boldsymbol{z}^2}$ is known. Using the chain rule, we have:

$$\frac{\partial \ell}{\partial w_{ij}^2} = \frac{\partial \ell}{\partial z_i^2} \frac{\partial z_i^2}{\partial w_{ij}^2} \tag{4.17}$$

Recall that,

$$z_i^2 = \sum_j w_{ij}^2 x_j + b_i^2, \quad i, j \in \{1, 2\} \tag{4.18}$$

Thus,

$$\frac{\partial \ell}{\partial w_{ij}^2} = \frac{\partial \ell}{\partial z_i^2} x_j \tag{4.19}$$

Then,

$$\begin{aligned} \frac{\partial \ell}{\partial \boldsymbol{W}^2} &= \begin{pmatrix} \frac{\partial \ell}{\partial z_1^2} x_1 & \frac{\partial \ell}{\partial z_1^2} x_2 \\ \frac{\partial \ell}{\partial z_2^2} x_1 & \frac{\partial \ell}{\partial z_2^2} x_2 \end{pmatrix}_{2\times 2} \\ &= (\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \begin{pmatrix} \frac{\partial \ell}{\partial z_1^2} & \frac{\partial \ell}{\partial z_2^2} \end{pmatrix})^T \\ &= (\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \frac{\partial \ell}{\partial \boldsymbol{z}^2})^T \end{aligned} \tag{4.20}$$

Since $\begin{pmatrix} x_1 & x_2 \end{pmatrix}^T = \boldsymbol{a}^1$, the equation above can be rephrased as,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^2} = (\boldsymbol{a}^1 \frac{\partial \ell}{\partial \boldsymbol{z}^2})^T \tag{4.21}$$

The derivatives of the loss function with respect to the bias in layer 2 is given by,

$$\frac{\partial \ell}{\partial \boldsymbol{b}^2} = \begin{pmatrix} b_1^2 & b_2^2 \end{pmatrix} = \begin{pmatrix} \frac{\partial \ell}{\partial b_1^2} & \frac{\partial \ell}{\partial b_2^2} \end{pmatrix} \tag{4.22}$$

Since,

$$\frac{\partial \ell}{\partial b_i^2} = \frac{\partial \ell}{\partial z_i^2} \times 1 \tag{4.23}$$

we have,

$$\frac{\partial \ell}{\partial \boldsymbol{b}^2} = \begin{pmatrix} \frac{\partial \ell}{\partial z_1^2} & \frac{\partial \ell}{\partial z_2^2} \end{pmatrix} = \frac{\partial \ell}{\partial \boldsymbol{z}^2} \tag{4.24}$$

Similarly, we can compute the derivatives of the loss function with respect to the weights and bias in layer 3 given by,

$$\frac{\partial \ell}{\partial \boldsymbol{W}^3} = (\boldsymbol{a}^2 \frac{\partial \ell}{\partial \boldsymbol{z}^3})^T \tag{4.25}$$

$$\frac{\partial \ell}{\partial \boldsymbol{b}^3} = \frac{\partial \ell}{\partial \boldsymbol{z}^3} \tag{4.26}$$

each respectively.

Suppose we use the cross-entropy loss function in our example,

$$\ell(\boldsymbol{a}^3, y) = -[y \log a_1^3 + (1-y)\log(1-a_1^3)] \tag{4.27}$$

Then we have,

$$\begin{aligned} \frac{\partial \ell}{\partial \boldsymbol{a}^3} &= \left(\tfrac{\partial \ell}{\partial a_1^3}\right) \\ &= \left(-\left[\tfrac{y}{a_1^3} + \tfrac{1-y}{1-a_1^3}(-1)\right]\right) \\ &= \left(\tfrac{a_1^3 - y}{a_1^3(1-a_1^3)}\right) \end{aligned} \tag{4.28}$$

Our choice of the activation function is $a_1^3 = \sigma(z_1^3)$. Thus,

$$\frac{\partial \boldsymbol{a}^3}{\partial \boldsymbol{z}^3} = \left(\sigma'(z_1^3)\right) = \left(a_1^3(1-a_1^3)\right) \tag{4.29}$$

Then,

$$\frac{\partial \ell}{\partial \boldsymbol{z}^3} = \frac{\partial \ell}{\partial \boldsymbol{a}^3}\frac{\partial \boldsymbol{a}^3}{\partial \boldsymbol{z}^3} = \left(a_1^3 - y\right) \tag{4.30}$$

Now we compute $\frac{\partial \ell}{\partial \boldsymbol{z}^2}$.
Since,

$$z_i^3 = w_{ij}^3 a_j^2 + b_i^3 \tag{4.31}$$

we have,

$$\frac{\partial z_i^3}{\partial a_j^2} = w_{ij}^3 \tag{4.32}$$

Thus,

$$\frac{\partial \boldsymbol{z}^3}{\partial \boldsymbol{a}^2} = \boldsymbol{W}^3 \tag{4.33}$$

Recall for function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$, the derivative $\frac{\partial f}{\partial \boldsymbol{z}} \in \mathbb{R}^{m \times n}$ is given as,

$$\frac{\partial f}{\partial \boldsymbol{z}} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial z_1} & \cdots & \frac{\partial f_m}{\partial z_n} \end{pmatrix}_{m \times n} \tag{4.34}$$

Since $a_i^2 = \tanh z_i^2$, we have

$$\frac{\partial \boldsymbol{a}^2}{\partial \boldsymbol{z}^2} = \begin{pmatrix} \frac{\partial a_1^2}{\partial z_1^2} & \frac{\partial a_1^2}{\partial z_2^2} \\ \frac{\partial a_2^2}{\partial z_1^2} & \frac{\partial a_2^2}{\partial z_1^2} \end{pmatrix} = \begin{pmatrix} 1-(a_1^2)^2 & 0 \\ 0 & 1-(a_1^2)^2 \end{pmatrix} \tag{4.35}$$

Thus,

$$\frac{\partial \ell}{\partial \boldsymbol{z}^2} = \frac{\partial \ell}{\partial \boldsymbol{z}^3}\frac{\partial \boldsymbol{z}^3}{\partial \boldsymbol{a}^2}\frac{\partial \boldsymbol{a}^2}{\partial \boldsymbol{z}^2} = \left(a_1^3 - y\right)\boldsymbol{W}^3 \begin{pmatrix} 1-(a_1^2)^2 & 0 \\ 0 & 1-(a_1^2)^2 \end{pmatrix} \tag{4.36}$$

More generally, consider that all layers are fully connected, i.e. every neuron in layer $\ell - 1$ has a connection with every unit in layer ℓ, and each layer consists of a linear function and non-linear activation.

We have the following forward equations:

$$\boldsymbol{a}^1 = \boldsymbol{x}_{input} \tag{4.37}$$

$$\boldsymbol{z}^\ell = \boldsymbol{W}^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell \tag{4.38}$$

$$\boldsymbol{a}^\ell = f_\ell(\boldsymbol{z}^\ell) \tag{4.39}$$

where f_ℓ may map $\boldsymbol{z}^\ell$ to $\boldsymbol{a}^\ell$ element-wise or vector-wise.

$$\ell(\boldsymbol{a}^L, y) = -\sum_{c=1}^{C} \mathbb{1}(y = c) \log a_c^L \tag{4.40}$$

where ℓ is the cross entropy loss function.

If there are n_L (output) neurons in layer L, then $\frac{\partial \ell}{\partial \boldsymbol{a}^L}$ and $\frac{\partial \ell}{\partial \boldsymbol{z}^L}$ are row vectors with n_L elements and $\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L}$ is the $n_L \times n_L$ Jacobian matrix given by,

$$\begin{pmatrix} \frac{\partial a_1^L}{\partial z_1^L} & \cdots & \frac{\partial a_1^L}{\partial z_{n_L}^L} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_{n_L}^L}{\partial z_1^L} & \cdots & \frac{\partial a_{n_L}^L}{\partial z_{n_L}^L} \end{pmatrix}$$

If f_L is applied element-wise, e.g., sigmoid, then this matrix is diagonal.

To compute the derivatives with respect to the network parameters, we have the following back-propagation equations:

$$\frac{\partial \ell}{\partial \boldsymbol{z}^L} = \frac{\partial \ell}{\partial \boldsymbol{a}^L}\frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \tag{4.41}$$

where layer L is the last layer, i.e. output layer.

$$\frac{\partial \ell}{\partial \boldsymbol{z}^\ell} = \frac{\partial \ell}{\partial \boldsymbol{z}^{\ell+1}}\boldsymbol{W}^{\ell+1}\frac{\partial \boldsymbol{a}^\ell}{\partial \boldsymbol{z}^\ell} \tag{4.42}$$

$$\frac{\partial \ell}{\partial \boldsymbol{W}^\ell} = (\boldsymbol{a}^{\ell-1}\frac{\partial \ell}{\partial \boldsymbol{z}^\ell})^T \tag{4.43}$$

$$\frac{\partial \ell}{\partial \boldsymbol{b}^\ell} = 1 \times \frac{\partial \ell}{\partial \boldsymbol{z}^\ell} = \frac{\partial \ell}{\partial \boldsymbol{z}^\ell} \tag{4.44}$$

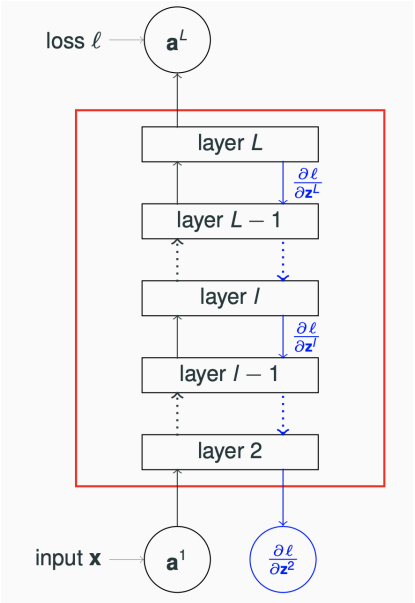

Figure 4.9: Flowchart of Back-Propagation

The running time to compute the gradient for a single data point is as many matrix multiplications as there are fully connected layers and performed twice during forward and backward pass.

The space requirement is big enough to store vectors $\boldsymbol{a}^\ell$, $\boldsymbol{z}^\ell$, and $\frac{\partial \ell}{\partial \boldsymbol{z}^L}$ for each layer.

We can process multiple examples together if training on minibatch. In this case, we need to make sure that all parameters fit in GPU memory.

4.2 Training Neural Networks

4.2.1 Initialising Weights and Biases

Initialising is important when minimising non-convex functions. We may get very different results depending on where we start the optimisation.

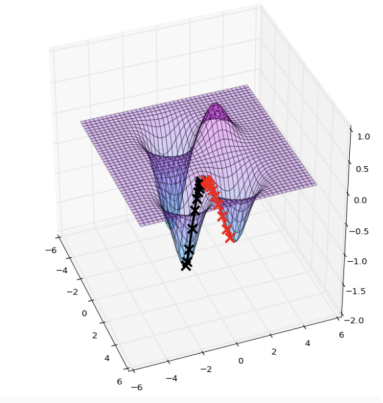

Figure 4.10: Non-Convex Optimization Surface

Weight initialisation for sigmoid/ReLU units:

- Suppose there are D weights $w_1, \cdots, w_D$.
- Draw w_i randomly from $\mathcal{N}(0, \frac{1}{D})$.

Bias initialisation:

- For sigmoid, use a random value around 0.
- For ReLU, use a small positive constant

4.2.2 Saturation and Vanishing Gradient

Saturation happens when pre-activations are in range where the activation function is very flat.

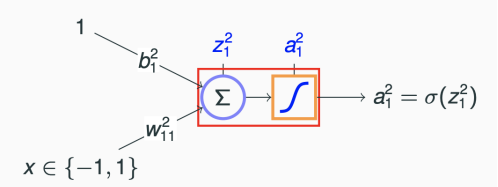

Figure 4.11: NOT Gate

For example, consider the NOT gate shown in Figure 4.10. If we use the squared loss function $\ell(a_1^2, y) = (a_1^2 - y)^2$, then,

$$\frac{\partial \ell}{\partial z_1^2} = 2(a_1^2 - y)\frac{\partial a_1^2}{\partial z_1^2} = 2(a_1^2 - y)\sigma'(z_1^2) \tag{4.45}$$

Recall that $\sigma'(z_1^2) = \sigma(z_1^2)(1 - \sigma(z_1^2))$. If $x = -1$, $w_{11}^2 = 5$ and $b_1^2 = 0$, then $\sigma'(z_1^2) = 0$. Gradient steps may not make much progress, as the gradient is very small yet the loss function $\ell(a_1^2, y) = (a_1^2 - y)^2$ has a relatively large value.

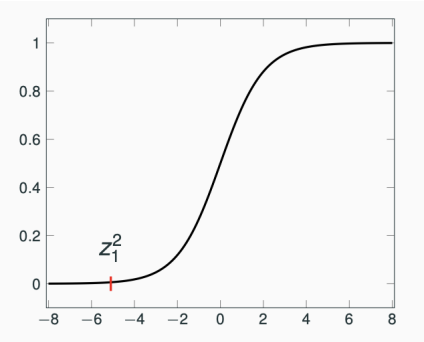

Figure 4.12: Saturation

Sometimes we can fix this problem by changing the squared loss to cross-entropy loss. Recall the cross-entropy loss $\ell(a_1^2, y) = -[y \log a_1^2 + (1 - y)\log(1 - a_1^2)]$. Thus, $\frac{\partial \ell}{\partial a_1^2} = \frac{a_1^2 - y}{a_1^2(1 - a_1^2)}$. Then $\frac{\partial \ell}{\partial z_1^2} = \frac{\partial \ell}{\partial a_1^2}\frac{\partial a_1^2}{\partial z_1^2} = a_1^2 - y$. Large gap between a_1^2 and y also means large magnitude of the gradient.

Saturation is inherent to many activation functions such as sigmoid and tanh.

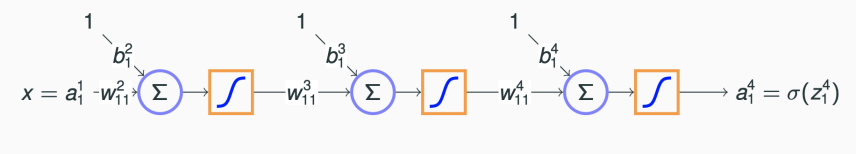

Figure 4.13: Four-layered Neural Network

For neural networks with many layers, as is shown in Figure 4.12, if we use sigmoid function as the activation function for all the hidden layers, we need to multiply many gradients $\sigma'(z_i^\ell)$ where $\sigma'(z) \in [0, \frac{1}{4}]$ when doing back-propagation. Thus, the gradient product $\approx$ in case of many layers.

The product with weights w_{ij}^ℓ may avoid vanishing yet may explode the gradient.

4.2.3 Rectified Linear Unit

Rectified linear unit, i.e. ReLU, is defined as,

$$\begin{aligned} f(z) &= \begin{cases} 0 & if\ z \leq 0 \\ z & if\ z \geq 0 \end{cases} \\ &= \max(0, z) \end{aligned} \tag{4.46}$$

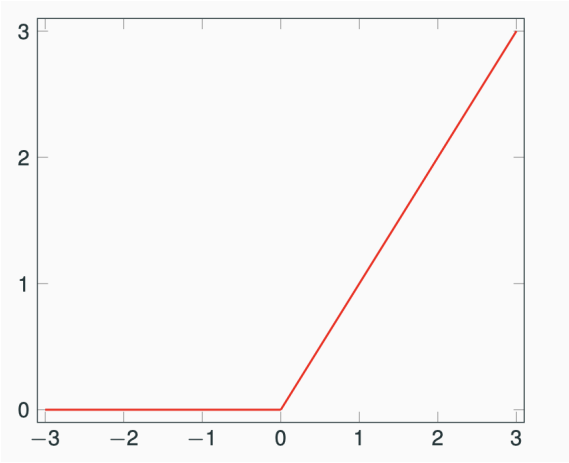

Figure 4.14: Rectified Linear Unit

The rectifier only saturates on one side and has derivative 1 for large z, so there is no saturation. To some extent, this solves the saturation and vanishing gradient problems.

Also, the neurons are sparsely activated, i.e. some might not activate at all. This is often desirable because not all neurons should fire for each input. For example, neurons that identifies animal eyes should not activate for building images. In practice, concise models often have better predictive power and less overfitting.

However, there still exists a dying ReLU problem, i.e. once a neuron gets negative, it is unlikely to recover and always outputs 0 as its pre-activation remains negative. Then those dying neurons become useless and hence a large part of network does nothing.

Possible reasons of the dying ReLU problem is large negative bias or too high learning rate. Thus, lowering the learning rate can solve it to some extent. If not, we can use leaky ReLU, where the slope for negative input is a constant.

Leaky ReLU is given as,

$$f(z) = \begin{cases} cz & if\ z \leq 0 \\ z & if\ z \geq 0 \end{cases} \tag{4.47}$$

Where c is a constant, usually chosen small.

Leaky ReLU can solves the dying ReLU problem since it does not have zero-slope parts. However, the performance of Leaky ReLU is not always superior to ReLU.

We can also use parametric ReLU where the slope for negative input is a network parameter a, given as,

$$f(z) = \begin{cases} az & if\ z \leq 0 \\ z & if\ z \geq 0 \end{cases} \tag{4.48}$$

4.2.4 Overfitting and Avoiding Overfitting

Deep neural networks have a lot of parameters. A fully connected layers with $n_1, \cdots, n_L$ units have at least $n_1n_2 + n_2n_3 + \cdots + n_{L-1}n_L$ parameters. For example, MLP for digit recognition has 2 million parameters and 60,000 training images and for image detection, the famous neural net by Krizhevsky, Sutskever, Hinton(2012), has 60 million parameters and 1.2 million training images.

Beyond regularisation, we can prevent overfitting in deep neural networks by early stopping, adding data and dropout.

Early Stopping

Assume the training uses iterative optimisation methods. The idea of early stopping is to keep aside a validation set and measure performance of the classifier after each gradient step. When validation error stops decreasing, i.e. a local minimum is reached, we stop the training. In this case, we need to compute the validation error after each iteration which is computationally more expensive. Thus, we can do this every few iterations to reduce overhead.

Adding Data

Typically, getting additional data is either impossible or expensive. However, we can fake the data. For example, images can be translated/rotated slightly and their brightness can be changed, etc.

Figure 4.15: Image Data Augmentation

We can also add data by adversarial training:

- Take trained (or partially trained model)
- Create examples by modifications "imperceptible to the human eye", but where the model fails
- Linear perturbation of non-linear models:

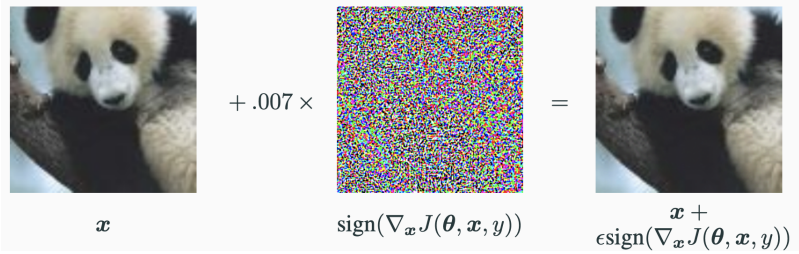

Figure 4.16: Linear Perturbation of Non-linear Models

Bagging and Dropout

Suppse we have the data set $\mathcal{D}$, sample $\mathcal{D}_1, \cdots, \mathcal{D}_k$ of size N from $\mathcal{D}$ with replacement. The bagging method is that we train k classifiers $f_1, \cdots, f_k$ on $\mathcal{D}_1, \cdots, \mathcal{D}_k$ and predict using majority (or average if using regression).

However, this bagging approach is not practical for deep networks. A variant of bagging that works for neural networks is dropout.

The drop-out method is, during each training step, a fraction of the randomly-chosen hidden neurons are ignored and weights and biases of the others are updated in the gradient step. At each gradient update step, the neurons ignored are different. This can prevent so-called co-adaptation among different neurons. At test time, the entire network is used. All weights need to be scaled: halved if dropout rate was 1/2.

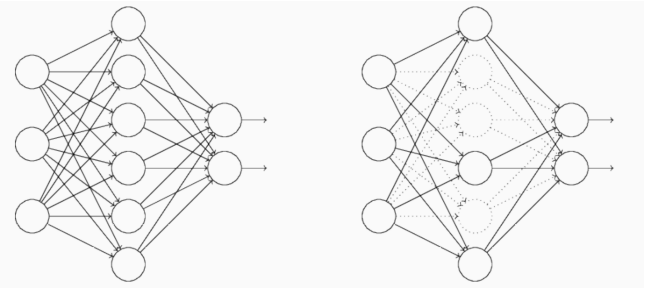

Figure 4.17: Dropout

Further ideas on avoiding overfitting include:

- Use parameter sharing (weight tying) in the model which will also reduce the computational cost.
- Exploit invariances to translation, rotation which is important to achieve high accuracy.
- Exploit locality in images, audio, text which is important to achieve high accuracy.

These are all achieved in convolutional neural networks, which will be discussed in the next section.

4.3 Convolutional Neural Networks

Convolutional neural networks, based on works of Fukushima, LeCun, Hinton (1980s), is one of the key reasons behind the recent popularity of machine learning. It makes significant advances in image, audio, signal processing and leads to highly accurate image recognition.

4.3.1 Convolution

Convolution operation on two real-valued functions. It expresses how the shape of one function is modified by the other. In neural networks, it is common to use cross-correlation instead.

A typical case is the convolution of an input $2-D$ image tensor I with $2-D$ filter (or kernel) tensor K. Tensor I maps coordinates to, eg, black/white or RGB colour code and tensor K is made up of weight parameters that are learned.

Tensor K convolving across tensor I gives the feature map $S(i,j)$, given by,

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{4.49}$$

where $\sum_m \sum_n I(i+m, j+n)K(m,n)$ is the summation of the Hadamard, i.e. element-wise, products of the image and the kernel.

It is common to have a non-linear activation function applied to S.

Figure 4.18 shows an example of an input image $I \in \mathbb{R}^{3\times4}$ convolving with a kernel $K \in \mathbb{R}^{2\times2}$.

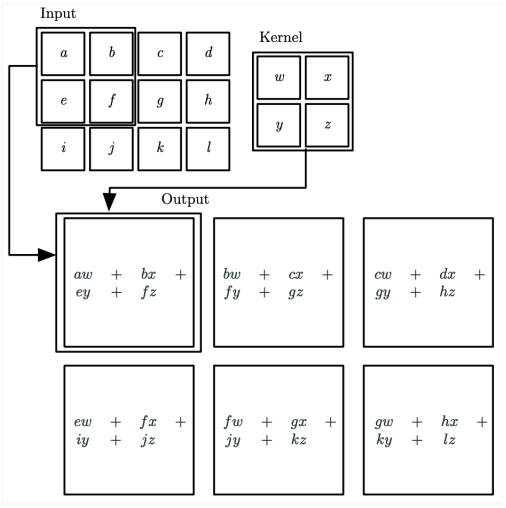

Figure 4.18: Convolution

We slide filter K over input image I one column at a time and move to the next row when finishing with one.

We compute $S(i,j)$ for $i \in \{0,1,2\}, j \in \{0,1\}$, given by,

$$\begin{aligned} S(i,j) =& I(i,j)K(0,0) + I(i+1,j)K(1,0) + \\ & I(i,j+1)K(0,1) + I(i+1,j+1)K(1,1) \end{aligned} \tag{4.50}$$

In this case, we use a stride of 1, i.e. sliding in steps of 1 columns/rows.

The convolution of the image $I^{\ell-1} \in \mathbb{R}^{m_{\ell-1} \times n_{\ell-1}}$ with a kernel $K \in \mathbb{R}^{k_1 \times k_2}$ and a stride s will give an output image of $I^{\ell} \in \mathbb{R}^{(\frac{m_{\ell-1}-k_1}{s}+1)\times(\frac{n_{\ell-1}-k_2}{s}+1)}$.

In order to retain the original size of the input image, we will have to add columns and rows with entries of 0 so that the input image $I^{\ell-1} \in \mathbb{R}^{(m_{\ell-1}+(k_1-s))\times(n_{\ell-1}+(k_2-s))}$, which is called padding.

Convolutional filters or kernels can detect patterns in images.

For example, to detect black-white vertical edges, we can use the filter,

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$$

and bias $b = -1$ with ReLU activation.

The output for one application is:

$$\begin{aligned} & \max(0, \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} - 1) = \\ & \max(0, -a + b - c + d - 1) \end{aligned} \tag{4.51}$$

This output is > 0 when the input matrix is

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

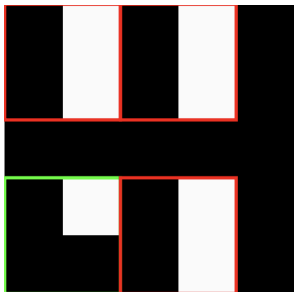

Figure 4.19: Black-White Vertical Edges Detector

To detect black L shapes, we can use the filter,

$$\begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}$$

and bias $b = 0$ with ReLU activation.

The output for one application is:

$$\begin{aligned} &\max(0, \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix}) = \\ &\max(0, -a + b - c - d) \end{aligned} \tag{4.52}$$

This output is > 0 when the input matrix is

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

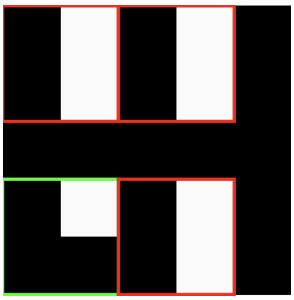

Figure 4.20: Black L Shapes Detector

Basic patterns detected by filters placed in the first layers of convnet are edges, textures, curves, objects, colors and shapes, i.e. corners, circles and squares. More sophisticated objects such as eyes and ears are detectable in later layers. In even deeper layers, things like full dogs, cats, lizards, and birds can be detected.

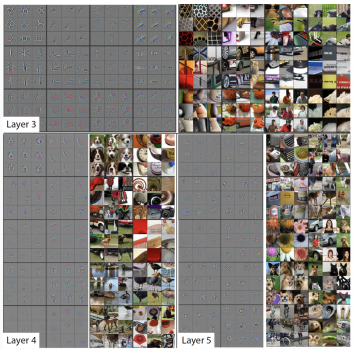

Figure 4.21: Patterns Detected

4.3.2 Max-Pooling

Max-pooling is a sample-based discretization process which generalise the model by combining several values into a single one. It decreases the chance of overfitting, reduces the number of parameters, and provides basic translation invariance and therefore more robust matching of features in the presence of small distorsions.

Once we know a specific feature is in the input (its activation value is high), its exact location is less important relative to the other features.

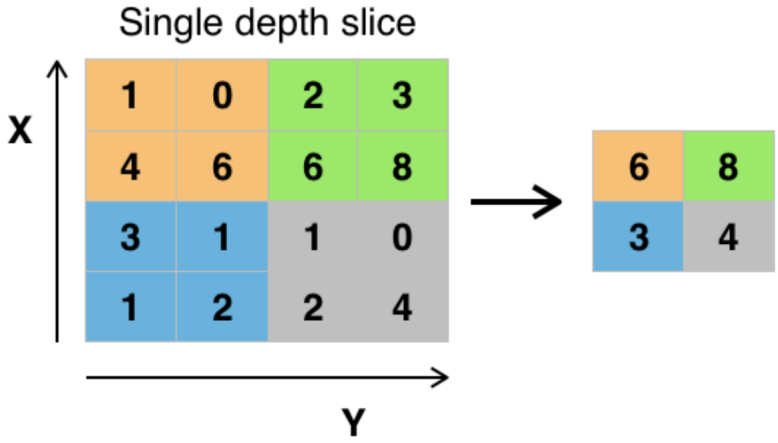

Figure 4.22: Max-Pooling

In addition to max-pooling, we also use min-pooling and average pooling which takes the minumum and average of the activation value each respectively. Average pooling is a softer version of max-pooling. For activations crossing several pooling neighbourhoods, average pooling gives a strong signal in the middle and soft at edges. It vives more information on where the feature edges are localised but might not extract good features if all not all inputs are needed.

4.3.3 Some Popular Convolutional Neural Networks

LeNet-5 (1998)

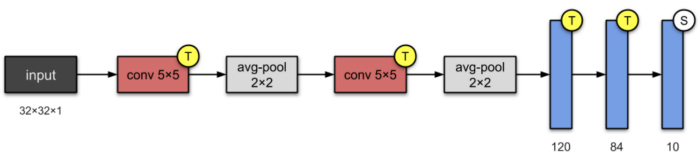

Figure 4.23: LeNet-5

- Two convolutional layers with 5×5 filers and tanh non-linear activation
- Two 2×2 average-pooling layers
- Three fully connected layers, 2 with tanh, one with softmax at end
- 60K parameters in total

AlexNet (2012)

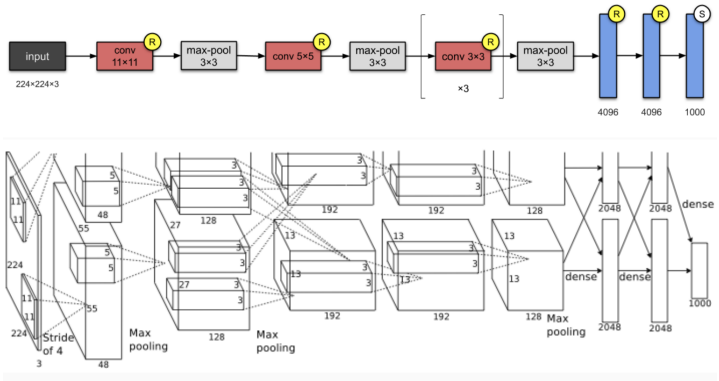

Figure 4.24: AlexNet

- Five convolutional layers with ReLU non-linear activation
- Two 3×3 max-pool layers
- Three fully connected layers, 2 with ReLU, one with softmax at end
- 60M parameters in total

VGG-16 (2014)

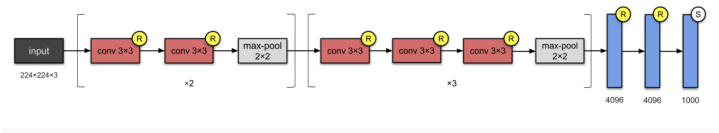

Figure 4.25: VGG-16

- Thirteen convolutional layers with 3×3 filers and ReLU non-linear activation
- Five 2×2 max-pool layers

- Three fully connected layers, 2 with ReLU, one with softmax at end
- 138M parameters in total

Inception-ResNet-V2 (2016)

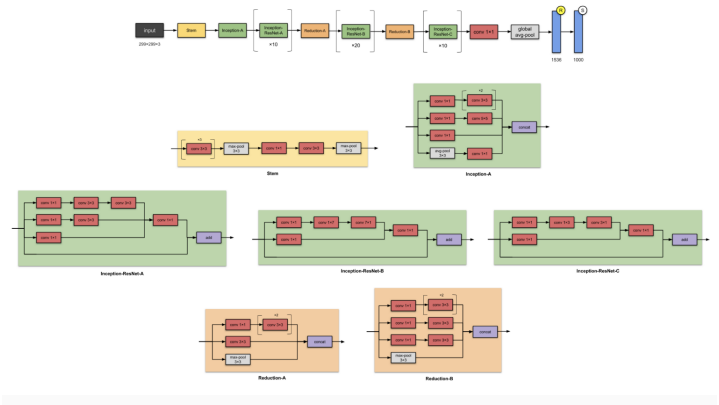

Figure 4.26: Inception-ResNet-V2

- Google-scale construction
- 56M parameters in total

4.3.4 Back-propagation in Convolutional Neural Networks

Consider convolutional layer between layer ℓ and $\ell+1$. The output of the layer ℓ has the shape $m_\ell \times n_\ell \times F_\ell$, with each entry indexed as $a^\ell_{i,j,f}$, where $m_\ell \times n_\ell$ is the width and height of the output image each respectively and F_ℓ is the number of the filters.

For the $\ell+1$ layer, we apply $F_{\ell+1}$ filters, each a tensor of shape $W_f \times H_f \times F_\ell$. The weights used in convolutional layers are shared as the filters convolve across the input layer. We assume no zero-padding and stride of 1 in each direction and index the entries in this convolutional layer, i.e. layer $\ell+1$, as $z^{\ell+1}_{i',j',f'}$. Then we have,

$$z^{\ell+1}_{i',j',f'} = b^{\ell+1,f'} + \sum_{i=1}^{W_{f'}} \sum_{j=1}^{H_{f'}} \sum_{f=1}^{F_\ell} a^\ell_{i'+i-1,j'+j-1,f} w^{\ell+1,f'}_{i,j,f} \tag{4.53}$$

where $b^{\ell+1,f'}$ is the bias term of the f'-th filter at layer $\ell+1$. Thus,

$$\frac{\partial z^{\ell+1}_{i',j',f'}}{\partial w^{\ell+1,f'}_{i,j,f}} = a^\ell_{i'+i-1,j'+j-1,f} \tag{4.54}$$

Then,

$$\frac{\partial \ell}{\partial w_{i,j,f}^{\ell+1,f'}} = \sum_{i',j'} \frac{\partial \ell}{\partial z_{i',j',f'}^{\ell+1}} a_{i'+i-1,j'+j-1,f}^{\ell} \tag{4.55}$$

Backpropagation also needs to compute $\frac{\partial \ell}{\partial z_{i,j,f}^{\ell}}$ using $\frac{\partial \ell}{\partial z_{i',j',f'}^{\ell+1}}$.

$$\frac{\partial z_{i',j',f'}^{\ell+1}}{a_{i,j,f}^{\ell}} = w_{i-i'+1,j-j'+1,f}^{\ell+1,f'} \tag{4.56}$$

Thus,

$$\frac{\partial \ell}{a_{i,j,f}^{\ell}} = \sum i', j', f' \frac{\partial \ell}{\partial z_{i',j',f'}^{\ell+1}} w_{i-i'+1,j-j'+1,f}^{\ell+1,f'} \tag{4.57}$$

Then,

$$\frac{\partial \ell}{z_{i,j,f}^{\ell}} = f'(z_{i,j,f}^{\ell}) \sum i', j', f' \frac{\partial \ell}{\partial z_{i',j',f'}^{\ell+1}} w_{i-i'+1,j-j'+1,f}^{\ell+1,f'} \tag{4.58}$$

For max-pooling layer,

$$s_{i',j'}^{\ell+1} = \max_{i,j \in \Omega(i',j')} a_{i,j}^{\ell} \tag{4.59}$$

Thus,

$$\frac{\partial s_{i',j'}^{\ell+1}}{\partial a_{i,j}^{\ell}} = \mathbb{1}((i,j) = \underset{\tilde{i},\tilde{j} \in \Omega(i',j')}{\arg\max}\ a_{\tilde{i},\tilde{j}}^{\ell} \tag{4.60}$$

In a word, the partial derivative is 1 precisely for the coordinates of the max value in the set and 0 otherwise.

4.3.5 Number of Parameters in Convolutional Neural Networks

Consider the convolutional layer with c^{ℓ} input channels and $c^{\ell+1}$ output channels.

The number of filters is $c^{\ell} \times c^{\ell+1}$ and the number of bias is $c^{\ell+1}$.

Suppose we use filters, i.e. kernels, of size $k_1 \times k_2$. The number of parameters of this convolutional layer is given as ,

$$\#\ parameters = c^{\ell} \times c^{\ell+1} \times k_1 \times k_2 + c^{\ell+1} \tag{4.61}$$

5

Unsupervised Learning

5.1 Clustering

Often data can be grouped together into subsets that are coherent. However, this grouping may be subjective. It is hard to define a general framework.

There are two types of clustering algorithms: feature-based where points are represented as vectors in $\mathbb{R}^D$ and (Dis)similarity-based where only pairwise (dis)similarities are known.

Two types of clustering methods: flat which partitions the data into k clusters and hierarchical which organises data as clusters, clusters of clusters, and so on.

5.1.1 k-Means Formulation of Clustering

The goal is to partition the data into subsets $C_1, \cdots, C_k$ where k is fixed in advance.

The quality of a partition defined by,

$$W(C) = \frac{1}{2} \sum_{j=1}^{k} \frac{1}{|c_j|} \sum_{i,i' \in C_j} d(\boldsymbol{x}_i, \boldsymbol{x}_{i'}) \tag{5.1}$$

If we use the Euclidean distance:

$$d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|_2^2$$

then the k-means objective is to minimise the sum of squares of distances to the mean within each cluster, i.e. to minimise jointly over partitions $C_1, \cdots, C_k$ and $\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_k$, given as,

$$W(C) = \sum_{j=1}^{k} \sum_{i \in C_j} \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2^2 \tag{5.2}$$

where $\boldsymbol{\mu}_j = \frac{1}{|c_j|} \sum_{i \in C_j} \boldsymbol{x}_i$.

This problem is NP-hard even for $k = 2$ for points in $\mathbb{R}^D$.

However, if we fix means $\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_k$, finding a partition $(C_j)_{j=1}^k$ that minimises $W(C)$

$$C_j = \{i \mid \|\boldsymbol{x}_i - \boldsymbol{\mu}_j\|_2 = \min_{j'} \|\boldsymbol{x}_i - \boldsymbol{\mu}'_j\|_2\}$$

is easy. Also, if we fix the clusters $C_1, \cdots, C_k$, minimising W with respect to $(\boldsymbol{\mu}_j)_{j=1}^k$

$$\boldsymbol{\mu}_j = \frac{1}{|c_j|} \sum_{i \in C_j} \boldsymbol{x}_i$$

is easy.

Thus, alternative minimisation can be performed by iteratively running these assignment and update steps.

5.1.2 The k-Means Algorithm

The k-means algorithm is used to partition a given set of observations into a predefined amount of k clusters. The algorithm starts with a random set of k center-points

$$\boldsymbol{\mu}_1, \cdots, \boldsymbol{\mu}_k$$

During each update step, all observations $\boldsymbol{x}$ are assigned to their nearest center-point (see equation 5.3). In the standard algorithm, only one assignment to one center is possible. If multiple centers have the same distance to the observation, a random one would be chosen.

$$S_i^{(t)} = \{\boldsymbol{x_p} : \left\|\boldsymbol{x_p} - \boldsymbol{\mu}_i^{(t)}\right\|^2 \leq \left\|\boldsymbol{x_p} - \boldsymbol{\mu}_j^{(t)}\right\|^2 \ \forall j, 1 \leq j \leq k\} \tag{5.3}$$

Afterwards, the center-points are repositioned by calculating the mean of the assigned observations to the respective center-points,

$$\boldsymbol{\mu}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\boldsymbol{x_j} \in S_i^{(t)}} \boldsymbol{x_j} \tag{5.4}$$

The update process reoccurs until all observations remain at the assigned center-points and therefore the center-points would not be updated anymore.

This algorithm is also called the Lloyd's algorithm.

Note that ties can be broken arbitrarily and choosing k random data points to be the initial k-means is a good idea.

Because the objective function W decreases every time a new partition is used, there are only finitely many partitions. Thus, the algorithm always converge.

Regardless of the initialization, the algorithm always terminate after at most k^N iterations where N is the number of input points and k is the number of clusters.

Convergence may be very slow in the worst-case, but typically fast on real-world instances. However, convergence is likely to be a local minimum. We can run the algorithm multiple times with random initialisation to tackle with this problem.

For the parameter k, larger k lowers the objective but might give poorer clustering, as is shown in Figure 5.1.

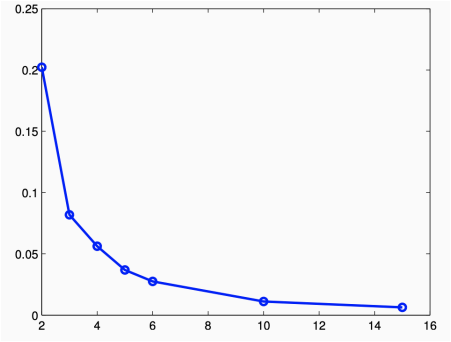

Figure 5.1: MSE on Test v.s. K

We can choose a suitable k by identifying a "kink" or "elbow" in the curve.

5.1.3 Beyond k-Means

k-medoids

In k-medoids algorithm, actual data points are assigned as cluster means rather than Euclidean averages. We can also use any distance between points beyond the squared Euclidean distance. In particular, we can use the Euclidean distance or any other ℓ_p norm

k-center

In k-center algorithm, the objective is to maximize over all dissimilarities between data and anchor. However, for k-means, it's the sum of all the dissimilarities in each cluster. In both cases, the anchors or centres take the role of means.

5.1.4 Transforming Input Formats

Clustering algorithms may work on different data formats. The data can be given in Euclidean space or as matrix of (dis)similarities. In this section, we will discuss how to transform Euclidean distance to dissimilarity and vice versa.

The weighted dissimilarity between (real-valued) attributes is,

$$d(\boldsymbol{x}, \boldsymbol{x}') = f(\sum_{i=1}^{D} w_i d_i(x_i, x_i'))$$

The simplest setting is $w_i = 1$, $d_i(x_i, x_i') = (x_i - x_i')^2$ and $f(z) = \sqrt{z}$, which corresponds to the Euclidean distance.

Weights allow us to emphasise features differently. If features are ordinal or categorical then define distance suitably as

$$d_i(x_i, x_i') = 1$$

if $x_i = x_i'$ and 0 otherwise.

In practice, it may be easier to define (dis)similarity between objects than embed them in Euclidean space. For example, for DNA sequences, we can use Hamming distance as measure of dissimilarity and for text data, we can use cosine kernel as measure of similarity.

Since algorithms such as k-means require however points to be in Euclidean space, we need to transform (dis)similarity measures into Euclidean space

Multidimensional Scaling gives a way to find an embedding of the data in Euclidean space that (approximately) respects the original distance/similarity.

Assume an ideal case where we are given the dissimilarity matrix $\boldsymbol{D}$ with pairwise Euclidean distances of points $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N$

$$D_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2$$

for $i, j \in [N]$.

Since distances are preserved under translation, rotation, reflection, etc. We cannot recover $\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$ exactly. However, we can aim to determine $\boldsymbol{X}$ up to these transformations.

If D_{ij} is the distance between points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, then

$$\begin{aligned} D_{ij} &= \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2 \\ &= \boldsymbol{x}_i^T \boldsymbol{x}_i - 2\boldsymbol{x}_i^T \boldsymbol{x}_j + \boldsymbol{x}_j^T \boldsymbol{x}_j \\ &= M_{ii} - 2Mij + Mjj \end{aligned} \tag{5.5}$$

where $\boldsymbol{M} = \boldsymbol{x}\boldsymbol{x}^T$ is is the $N \times N$ matrix of dot products: $Mij = \boldsymbol{x}_i^T \boldsymbol{x}_j$.

If

$$\sum_i \boldsymbol{x}_i = 0$$

$\boldsymbol{M}$ can be uniquely recovered from $\boldsymbol{D}$. Otherwise, $\boldsymbol{M}$ can be recovered from $\boldsymbol{D}$ up to some translation.

Now we have $\boldsymbol{M}$, To obtain points $\hat{\boldsymbol{x}_1}, \cdots, \hat{\boldsymbol{x}_N}$ from $\boldsymbol{M}$, we use Singular Value Decomposition of $\boldsymbol{M}$.

The Singular Value Decomposition of a matrix $\boldsymbol{X} \in \mathbb{R}^{D \times D}$ is

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

SVD is thin SVD if $\boldsymbol{U} \in \mathbb{R}^{N\times D}$ and $\boldsymbol{V} \in \mathbb{R}^{D\times D}$ are orthonormal matrices; $\boldsymbol{\Sigma} \in \mathbb{D}^{D\times D}$ is diagonal with entries along the diagonal satisfying

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_D \geq 0$$

and $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}_D$.

SVD is full SVD $\boldsymbol{U} \in \mathbb{R}^{N\times N}$, $\boldsymbol{\Sigma} \in \mathbb{R}^{N\times D}$ and $\boldsymbol{V} \in \mathbb{R}^{D\times D}$.

Eigendecomposition: if $\boldsymbol{M}$ is squared and $\boldsymbol{U} = \boldsymbol{V}$

$$\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^T$$

If $\boldsymbol{M}$ is square, symmetric and positive semi-definite, starting from M, we can reconstruct $\hat{\boldsymbol{X}}$ using the eigendecomposition of $\boldsymbol{M}$

$$\boldsymbol{M} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^T$$

Since $\boldsymbol{\Sigma}$ has non-negative diagonal entries, so we can take its square root.

By letting

$$\hat{\boldsymbol{X}} = \boldsymbol{U}\boldsymbol{\Sigma}^{\frac{1}{2}}$$

, we obtain

$$\hat{\boldsymbol{X}}\hat{\boldsymbol{X}}^T = \boldsymbol{U}\boldsymbol{\Sigma}^{\frac{1}{2}}(\boldsymbol{U}\boldsymbol{\Sigma}^{\frac{1}{2}})^T$$

We thus find the points $\hat{\boldsymbol{x}}_i$ as rows of $\boldsymbol{U}\boldsymbol{\Sigma}^{\frac{1}{2}}$.

In practice, we can use Mercer kernel to define the similarity and thus the matrix $\boldsymbol{M}$ is positive semi-definite. Then the above derivation works.

In general, recovering the data points from dissimilarity matrix is to solve the optimisation problem with (non-convex) objective:

$$\underset{\hat{\boldsymbol{x}_1},\cdots,\hat{\boldsymbol{x}_N}}{\arg\min} \sum_{i\neq j}(\|\hat{\boldsymbol{x}}_i - \hat{\boldsymbol{x}}_j\|_2^2 - D_{ij})^2$$

This objective function is called the stress function. It gives the degree to which it is not possible to find a Euclidean embedding for $\boldsymbol{D}$. There is no closed-form solution to this optimization problem. Local optimum is obtained via gradient descent.

5.1.5 Hierarchical Clustering

Hierarchical structured data exists all around us. For example, measurements of different species and individuals within species, top-level and low-level categories in news articles and country, canton, town level data are all hierarchical structured data.

There are two algorithmic strategies for clustering: agglomerative and divisive algorithms. Agglomerative algorithm refers to bottom-up method,where

clusters are formed by merging smaller clusters. The most popular agglomerative algorithm is called the Linkage algorithm. Divisive algorithm, on the other hand, refers to top-down method, where clusters are formed by splitting larger clusters.

We can visualise the clustering as a dendrogram or binary tree, as is shown in the Figure 5.2.

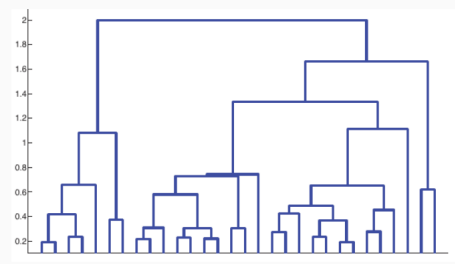

Figure 5.2: Visualizing the Clustering as a Dendrogram

Cutting the dendrogram at some level gives a partition of data.

To find hierarchical clusters we need to define dissimilarity at cluster level, not just at data points.

Suppose we have dissimilarity at data point level, e.g., $d(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x} - \boldsymbol{x}'\|$.

Different ways to define dissimilarity at cluster level, say C and C' are given by

- Single Linkage

$$D(C, C') = \min_{\boldsymbol{x} \in C, \boldsymbol{x}' \in C'} d(\boldsymbol{x}, \boldsymbol{x}')$$

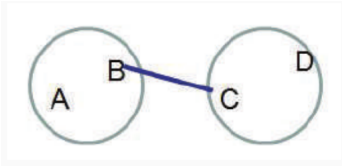

Figure 5.3: Single Linkage

- Complete Linkage

$$D(C, C') = \max_{\boldsymbol{x} \in C, \boldsymbol{x}' \in C'} d(\boldsymbol{x}, \boldsymbol{x}')$$

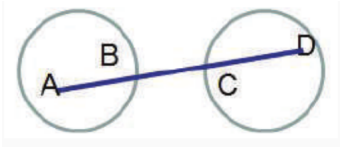

Figure 5.4: Complete Linkage

- Average Linkage

$$D(C, C') = \frac{1}{|C| \cdot |C'|} \sum_{\boldsymbol{x} \in C, \boldsymbol{x}' \in C'} d(\boldsymbol{x}, \boldsymbol{x}')$$

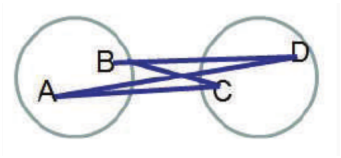

Figure 5.5: Average Linkage

Linkage-based Clustering Algorithm

The algorithm starts with randomly initialized singleton clusters

$$C_i = \{i\}$$

and clusters available for merging

$$S = \{1, \cdots, N\}$$

During each update step, Pick 2 most similar clusters

$$(j, k) = \arg\min_{j,k \in S} \; D(j, k)$$

For some new index l, let

$$C_l = C_j \cup C_k$$

Break if

$$C_l = \{1, \cdots, N\}$$

Otherwise, update

$$S := (S \backslash \{j, k\}) \cup \{l\}$$

Then update $D(i, l)$ for all $i \in S$ using desired linkage property.

5.1.6 Spectral Clustering

K-means will typically form clusters that are spherical, elliptical and convex. To find non-convex clusters, spectral clustering is a (related) alternative that often works better.

Similarity Graph Construction

To perform spectral clustering, we need to construct k-nearest neighbour graph from data by assigning one node for every point in dataset. Then (i, j) is an edge if either i is among the k nearest neighbours of j or vice versa.

The weight of edge (i, j), if it exists, is given by similarity measure $s_{i,j}$

$$s_{i,j} = e^{-\frac{\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}{\sigma}}$$

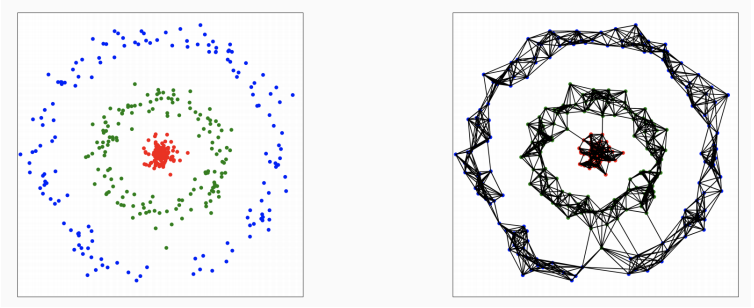

Figure 5.6: Spectral Clustering: Graph Construction

Partitioning a Graph into Two Clusters

Min-Cut Partition graph into two sets A and B such that weight of edges connecting vertices in A to vertices in B is minimum.

$$\text{cut}(A, B) := \sum_{i \in A, j \in B} s_{ij}$$

Mincut can give bad cuts (only one node on one side of the cut).

Normalized-Cut Partition graph into two sets A and B such that weight of edges connecting vertices in A to vertices in B is minimum and the size of A and B are very similar.

$$\text{Ncut}(A, B) := \text{cut}(A, B)(\frac{1}{\text{vol(A)}} + \frac{1}{\text{vol(B)}})$$

where

$$\frac{1}{\text{vol(A)}}$$

is the sum of the edges connecting all pairs of vertices in A.

Normalized cut is typically NP-hard to compute. Relaxations of these problems give eigenvectors of Laplacian.

Spectral Clustering Algorithm

Suppose we have the weighted graph with weighted adjacency matrix $\boldsymbol{W}$ as input and number k of clusters to construct.

We start with construct Laplacian

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{W}$$

where $\boldsymbol{W}$ is the weighted adjacency matrix and $\boldsymbol{D}$ is (diagonal) degree matrix with entries

$$D_{ii} = \sum_j w_{ij}$$

Then we find the first k(hyper-parameter) eigenvectors of the matrix $\boldsymbol{L}$

$$\boldsymbol{v}_1 = \mathbf{1}, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_k$$

Next we build the feature matrix

$$\boldsymbol{V} = [\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_k] \in \mathbb{R}^{N \times k}$$

with the eigenvalues as columns.

We can interpret the rows of $\boldsymbol{V}$ as new data points $\boldsymbol{z}_i \in \mathbb{R}^k$ and cluster the points $\boldsymbol{z}_i$ with the k-means algorithm in $\mathbb{R}^k$.

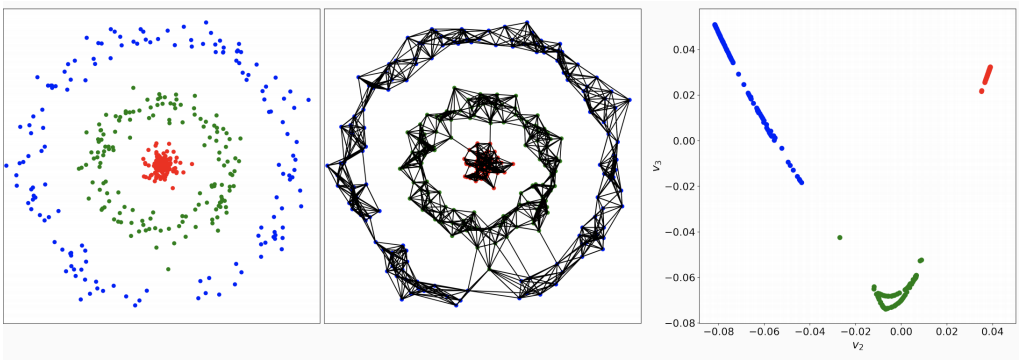

Figure 5.7: Spectral Clustering: $k = 2$

5.2 Principal Component Analysis

Real-world data may have correlated variables, i.e. when one variable changes, the other changes as well. Data synthesised by different teams may have redundancy.

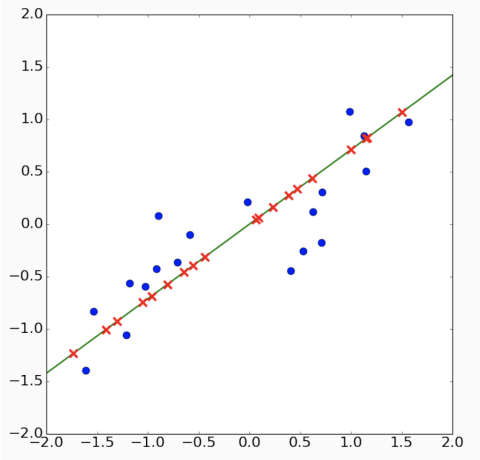

Figure 5.8: Dimensionality Reduction: Principal Component Analysis

Dimensionality reduction projects the data into a lower dimensional subspace while still capturing the essence of the original data. It is a preprocessing step before applying other learning algorithms.

As is shown in Figure 5.8, the red points are orthogonal projections of the blue points onto a vector representing the 1st principal component. The 2D points are projected onto 1D.

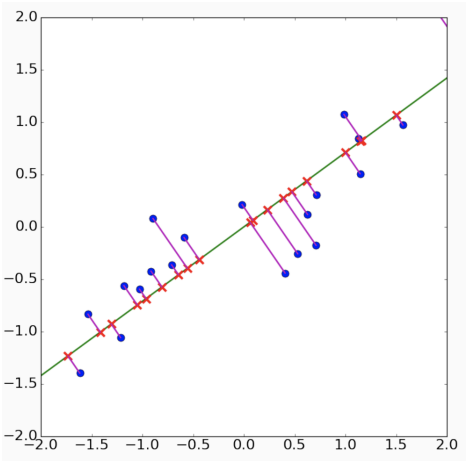

Figure 5.9: Principal Component Analysis: Reconstruction Errors

Principal Component Analysis (PCA) used for dimensionality reduction identifies a small number of directions which explain most variation in data.

The magenta lines shown in Figure 5.9 are the distances of the blue points from the first principal component. They are also called reconstruction errors.

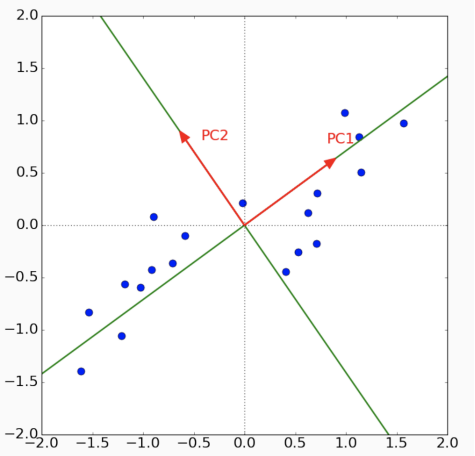

Figure 5.10: Principal Component Analysis

The second principal component is computed as the first principal component after the correlation with first principal component has been subtracted from the points. The second principal component is orthogonal to the first principal component.

5.2.1 PCA: Maximum Variance View

PCA keeps subspace with maximum variance. If we align the points with the principal components and move as much of the variance as possible using an orthogonal transformation into the first few dimensions, the values in the remaining dimensions tend to be small and may be dropped with minimal loss of information, i.e. PCA is the optimal orthogonal transformation that keeps the subspace with the largest "variance".

PCA is a linear dimensionality reduction technique searching for the directions of maximum variance in the data $(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N)$. It finds a set of orthonormal vectors $(\boldsymbol{v}_1, \cdots, \boldsymbol{v}_k)$ such that the first principal component (PC) $\boldsymbol{v}_1$ is the direction of largest variance, the second PC $\boldsymbol{v}_2$ is the direction of largest variance orthogonal to $\boldsymbol{v}_1$ and the i^{th} PC $\boldsymbol{v}_i$ is the direction of largest variance orthogonal to $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_{i-1}$.

Suppose we are independent and identically distributed data

$$\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$$

PCA can be rephrased as the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \|\boldsymbol{X}\boldsymbol{v}_1\|^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_1\| = 1 \end{aligned}$$

where $\boldsymbol{v}_1 \in \mathbb{R}^D$.

The solution is

$$\begin{aligned} \boldsymbol{v}_1 &= \underset{\boldsymbol{v}_1:\|\boldsymbol{v}_1\|=1}{\arg\max} \quad \boldsymbol{v}_1^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{v}_1 \\ &= \underset{\boldsymbol{v}_1}{\arg\max} \quad \frac{\boldsymbol{v}_1^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{v}_1}{\boldsymbol{v}_1^T \boldsymbol{v}_1} \end{aligned} \tag{5.6}$$

Recall the Rayleigh quotient, for positive semi-definite $\boldsymbol{M}$, the max value of

$$R(\boldsymbol{M}, \boldsymbol{v}) = \frac{\boldsymbol{v}^T \boldsymbol{M} \boldsymbol{v}}{\boldsymbol{v}^T \boldsymbol{v}}$$

is the largest eigenvalue of $\boldsymbol{M}$ which is attained for $\boldsymbol{v}$ being the corresponding eigenvector.

Since $\boldsymbol{X}^T \boldsymbol{X}$ is positive semi-definite, the solution to our problem is the corresponding eigenvector of the largest eigenvalue of $\boldsymbol{X}^T \boldsymbol{X}$.

Let

$$\boldsymbol{z} = \boldsymbol{X}\boldsymbol{v}_1 = [\boldsymbol{x}_1^T \boldsymbol{v}_1, \cdots, \boldsymbol{x}_N^T \boldsymbol{v}_1]$$

Thus,

$$z_i = \boldsymbol{x}_i^T \boldsymbol{v}_1$$

$\|\boldsymbol{X}\boldsymbol{v}_1\|^2$ is the variance of the projections of data onto $\boldsymbol{v}_1$ (mod constant factor $\frac{1}{N}$)

$$\|\boldsymbol{X}\boldsymbol{v}_1\|^2 = \sum_{i=1}^{N} z_i^2 = \boldsymbol{z}^T \boldsymbol{z} = \boldsymbol{v}_1^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{v}_1$$

This assumes the data is centered

$$\sum_i \boldsymbol{x}_i = \boldsymbol{0}$$

We can find $\boldsymbol{v}_2, \cdots, \boldsymbol{v}_k$ that are all successively orthogonal to previous directions and maximise (as yet unexplained) variance

$$\hat{\boldsymbol{X}}_j = \boldsymbol{X} - \sum_{s=1}^{j-1} \boldsymbol{X} \boldsymbol{v}_s \boldsymbol{v}_s^T \quad j \in [k] \tag{5.7}$$

5.2.2 PCA: Best Reconstruction View

Suppose we are independent and identically distributed data

$$\boldsymbol{X} = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N]$$

PCA in case of one PC can be rephrased as the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \|\boldsymbol{x}_i - \tilde{\boldsymbol{x}_i}\|^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_1\| = 1 \end{aligned}$$

where $\boldsymbol{v}_1$ is the direction of projection and $\tilde{\boldsymbol{x}_i}$ is the mapping of the point $\boldsymbol{x}_i$ given by

$$\tilde{\boldsymbol{x}_i} = \frac{\boldsymbol{v}_1 \boldsymbol{x}_i}{\|\boldsymbol{v}_1\|} \boldsymbol{v}_1 = (\boldsymbol{v}_1 \boldsymbol{x}_i) \boldsymbol{v}_1 \tag{5.8}$$

Let's now look at the equivalence between the objectives of the maximum variance view and best reconstruction view in the case of one PC.

$$\begin{aligned} \sum_{i=1}^{N} \|\boldsymbol{x}_i - \tilde{\boldsymbol{x}_i}\|^2 &= \sum_{i=1}^{N} \|\boldsymbol{x}_i - \tilde{\boldsymbol{x}_i}\|^2 \\ &= \sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2 - 2(\boldsymbol{x}_i \tilde{\boldsymbol{x}_i}) + \|\tilde{\boldsymbol{x}_i}\|^2 \\ &= \sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2 - 2(\boldsymbol{x}_i \tilde{\boldsymbol{x}_i}) + (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \|\boldsymbol{v}_1\|^2 \\ &= \sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2 - 2(\boldsymbol{x}_i \tilde{\boldsymbol{x}_i}) + (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \times 1 \\ &= \sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2 - \sum_{i=1}^{N} (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \end{aligned} \tag{5.9}$$

Thus, our objective of the best reconstruction view in the case of one PC becomes,

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2 - \sum_{i=1}^{N} (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_1\| = 1 \end{aligned}$$

Since $\sum_{i=1}^{N} \|\boldsymbol{x}_i\|^2$ is a constant given by the inputs(thus is not to be optimized), this objective is equivalent to:

$$\begin{aligned} \text{minimize} \quad & -\sum_{i=1}^{N} (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_1\| = 1 \end{aligned}$$

Equivalently,

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} (\boldsymbol{v}_1 \boldsymbol{x}_i)^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_1\| = 1 \end{aligned}$$

So the same $\boldsymbol{v}_1$ satisfies the two objectives of the maximum variance view and best reconstruction view in the case of one PC.

Finding Principal Components using Singular Value Decomposition

Suppose $\boldsymbol{V}_k \in \mathbb{R}^{D \times k}$ is such that columns of $\boldsymbol{V}_k$ are orthogonal, then projection of data $\boldsymbol{X}$ onto subspace is defined by

$$\boldsymbol{Z}_k = \boldsymbol{X}\boldsymbol{V}_k \tag{5.10}$$

PCA in case of k PCs can be rephrased as the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{V}_k \boldsymbol{V}_k^T \boldsymbol{x}_i\|^2 \\ \text{subject to} \quad & \|\boldsymbol{v}_i\| = 1 \quad i \in [k] \end{aligned}$$

Recall the SVD of $\boldsymbol{X}$:

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

where

- $\boldsymbol{\Sigma} \in \mathbb{R}^{N \times D}$ is diagonal with singular values.

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_D \geq 0$$

- $\boldsymbol{V} \in \mathbb{R}^{D \times D}$ is orthonormal matrix with right singular vectors.
- $\boldsymbol{U} \in \mathbb{R}^{N \times N}$ is orthonormal matrix with left singular vectors.

Since $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$,

$$\boldsymbol{X}^T\boldsymbol{X} = (\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T) = \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{\Sigma}\boldsymbol{V}^T \tag{5.11}$$

where

$$\boldsymbol{\Sigma}^T\boldsymbol{\Sigma} = \boldsymbol{\Sigma}^2 \in \mathbb{R}^{D \times D}$$

The eigenvectors of $\boldsymbol{X}^T\boldsymbol{X}$ are the right singular vectors $\boldsymbol{v}_i$ of $\boldsymbol{X}$ and the eigenvalues of $\boldsymbol{X}^T\boldsymbol{X}$ are the squares of the singular values σ_i of $\boldsymbol{X}$.

Similarly, the eigenvectors $\boldsymbol{u}_j$ of $\boldsymbol{X}^T\boldsymbol{X}$ with eigenvalue σ^2 are the left singular vectors of $\boldsymbol{X}$ (this can be used in case $D > N$).

Now, the PCA transformation is given as

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Sigma} \tag{5.12}$$

We can truncate Z. For $\boldsymbol{Z}_k \in \mathbb{R}^{N\times k}$, we only need the first k largest singular values and their singular vectors

$$\boldsymbol{Z}_k = \boldsymbol{U}_k\boldsymbol{\Sigma}_k = \boldsymbol{X}\boldsymbol{V}_k$$

Then the reconstruction is given by

$$\tilde{\boldsymbol{X}} = \boldsymbol{Z}_k\boldsymbol{V}_k^T = \boldsymbol{U}_k\boldsymbol{\Sigma}_k\boldsymbol{V}_k^T \tag{5.13}$$

where $\|\cdot\|_F$ is the Frobenius norm

$$\|\boldsymbol{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{i,j}|^2}$$

The reconstruction error

$$\begin{aligned}\sum_{i=1}^{N}\|\boldsymbol{x}_i - \boldsymbol{V}_k\boldsymbol{V}_k^T\boldsymbol{x}_i\|^2 &= \|\boldsymbol{X} - \tilde{\boldsymbol{X}}\|_F \\ &= \|\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T - \boldsymbol{U}_k\boldsymbol{\Sigma}_k\boldsymbol{V}_k^T\|_F \\ &= \|\sum_{j=1}^{D}\sigma_j\boldsymbol{u}_j\boldsymbol{v}_j^T - \sum_{j=1}^{k}\sigma_j\boldsymbol{u}_j\boldsymbol{v}_j^T\|_F \\ &= \sum_{j=k+1}^{D}\sigma_j^2\end{aligned} \tag{5.14}$$

This corresponds to the Eckart-Young theorem: $\boldsymbol{Z}_k$ is the nearest possible matrix of rank k to $\boldsymbol{X}$, i.e. the difference between the two has the smallest Frobenius norm.

Algorithm for Finding PCs (when $N > D$)

- Variant 1: Construct $\boldsymbol{X}^T\boldsymbol{X}$ in $O(D^2N)$ and its eigenvectors in $O(D^3)$.
- Variant 2: Iterative methods to get top k singular (right) vectors directly:
  - Initiate $\boldsymbol{v}_0$ to be random unit norm vector
  - Iterative Update:

$$\boldsymbol{v}^{t+1} = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{v}^t$$
$$\boldsymbol{v}^{t+1} = \frac{\boldsymbol{v}^{t+1}}{\|\boldsymbol{v}^{t+1}\|}$$

- Update step takes $O(ND)$ time (compute $\boldsymbol{X}\boldsymbol{v}^T$ first, then $\boldsymbol{X}^T(\boldsymbol{X}\boldsymbol{v}^T)$).
- This gives the singular vector corresponding to the largest singular value.
- Subsequent singular vectors obtained by choosing $\boldsymbol{v}_0$ orthogonal to previously identified singular vectors (this needs to be done at each iteration to avoid numerical errors creeping in).

Algorithm for Finding PCs (when $D \gg N$)

Constructing the matrix $\boldsymbol{X}\boldsymbol{X}^T$ takes time $O(N^2D)$. Eigenvectors of $\boldsymbol{X}\boldsymbol{X}^T$ can be computed in time $O(N^3)$.

The eigenvectors give the 'left' singular vectors, $\boldsymbol{u}_i$ of $\boldsymbol{X}$. To obtain $\boldsymbol{v}_i$, we use the fact that

$$\boldsymbol{v}_i = \sigma^{-1}\boldsymbol{X}^T\boldsymbol{u}_i$$

Iterative method can be used directly as in the case when $N > D$.

5.2.3 How Many Principal Components to Pick

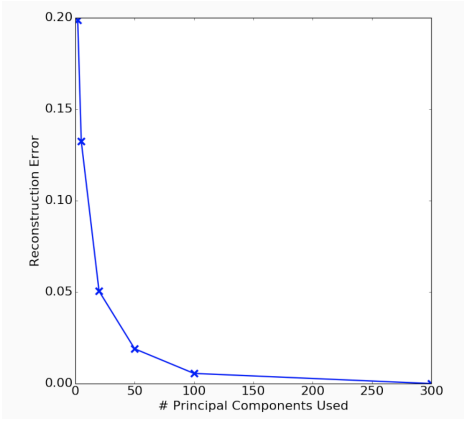

Figure 5.11: Principal Component Used v.s. Reconstruction Errors

We can choose a suitable number of PCs by identifying an 'elbow' in the curve of reconstruction error v.s. the number of PCs.

If SVD is computed, fix relative error threshold $0 \leq t \leq 1$and choose k such that

$$\frac{\|\boldsymbol{X} - \boldsymbol{X}_k\|_F^2}{\|\boldsymbol{X}\|_F^2} \leq t$$

where $\|\cdot\|_F$ is the Frobenius norm.