COMS W4771: Machine Learning Spring 2023

Machine Learning

Prof. Daniel Hsu, Dr. Nakul Verma *Scribe: Ekene Ezeunala*

Contents

§1 Lecture 01—16th January, 2023

Logistics. Lectures[1] will be held on Tuesdays and Thursdays from 10:10am to 11:25am in CSB 451. The textbooks for the course are [HTF09], [Bis06], [Mur12], [GBCB16], [Hal10], and [SSBD14]. The course will be graded based on homeworks (40%), a midterm (30%), and a final exam (30%)—the last two will be held in class. The prerequisites for the course are probability, statistics, linear algebra, data structures, and algorithms.

§1.1 Introduction

Machine learning is the study of making machines learn a concept without having to explicitly program it. In machine learning we are concerned with constructing algorithms that can learn from input data (and possibly make predictions on new data), find interesting patterns in data, etc. We are also interested in analysing these algorithms to understand the limits of learning. For example, are there problems that are inherently hard to learn? Is there a problem that cannot be learned? Are there problems that can be learned, but not efficiently?

In this course, we will:

- Abstractly study a prediction problem and come up with a solution which is simultaneously applicable to a wide range of problems.
- Explore different (successful) paradigms and algorithms that have been successful in prediction tasks.

Here are some prediction problems we might be interested in:

1. Handwritten character recognition: Given an image of a handwritten digit, predict the digit.
2. Spam detection: Given an email, predict whether it is spam or not.
3. Medical diagnosis: Given a patient's symptoms, predict the disease.
4. Credit scoring: Given a person's financial history, predict whether they will default on a loan.
5. Object recognition: Given an image, detect and classify objects in the image.

For each of these problems, there are certain commonalities that carry across; in particular, there is:

- An input $x = (x^{(1)}, \dots, x^{(d)}) \in \mathcal{X}$, which is a vector of features coming from some (usually real) space $\mathcal{X}$. For example, in the case of handwritten character recognition, x is a vector of pixel values, and $\mathcal{X}$ is the space of all possible pixel values.
- An output $y \in \mathcal{Y}$, which is a label or a value that we want to predict. For example, in the case of handwritten character recognition, y is the actual digit that was written.
- A learnable mapping $f \colon \mathcal{X} \to \mathcal{Y}$ that we want to learn. For example, in the case of handwritten character recognition, f is a function that takes an image and returns the digit that was written.

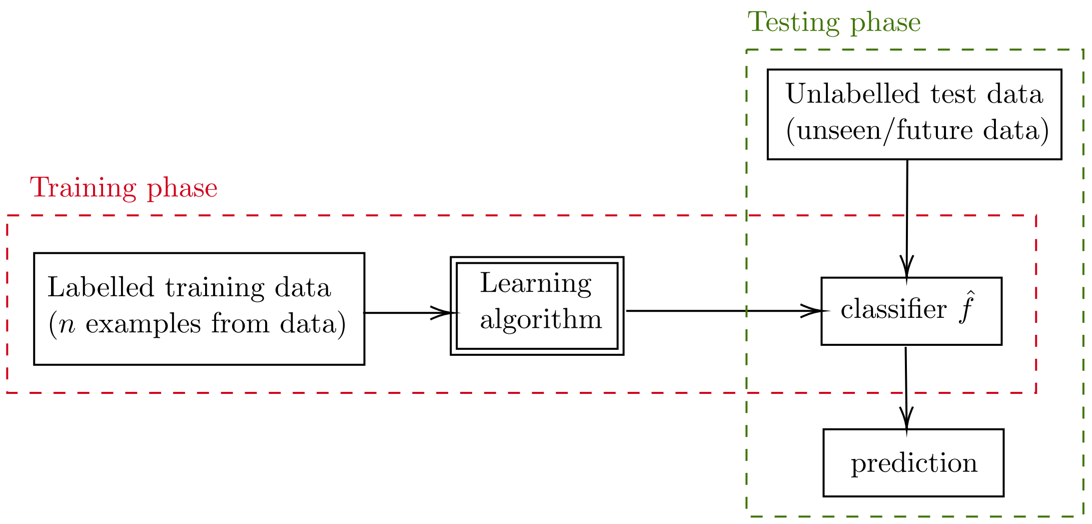
The problem of classification is parsimonously reducible to that of selecting some sufficiently good f

[1]The lecture notes were a bit supplemented, as lectures were often not as detailed and favoured heavy interaction as opposed to depth. Particular credit goes to James for his complementary contributions.

from a (possibly infinite) set of functions $\mathcal{F}$.

§1.1.1 Supervised learning

The goal of supervised learning is to find an underlying input-output relationship in a data set. Given data points $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, we will assume that there is a relatively simple function $f^*\colon \mathcal{X} \to \mathcal{Y}$ such that $y_i = f^*(\mathbf{x}_i)$ for most i. The learning task is then: given n examples from the data, find an approximation $\hat{f}$ to f^* that is good on new/unseen examples.


Testing phase
Unlabelled test data (unseen/future data)
Training phase
Labelled training data (n examples from data)
Learning algorithm
classifier $\hat{f}$
prediction

Example 1.1. Consider the problem of predicting the price of a house given its size. We are given a dataset of houses and their sizes and prices. We want to learn a function that takes the size of a house and returns its price. This is a regression problem, and the input space is $\mathcal{X} = \mathbb{R}$ and the output space is $\mathcal{Y} = \mathbb{R}$. The function f^* that we want to learn is the true price of a house given its size.

Example 1.2. Suppose we wish to find a function $f\colon \mathcal{X} \to \mathcal{Y}$, and we have a data set $\{(2,3),(4,5)\}$. These training examples are denoted $\left\{\left(\mathbf{x}^{(n)}, y^{(n)}\right)\right\}_{n=1}^{2}$, and our goal is to train a model that can then predict y for new $\mathbf{x}$.
Suppose our model gets $\mathbf{x} = 4$. Then we would expect our model to predict $y = 5$. This is a feature of learning: remembering the training data. Now suppose our model gets $\mathbf{x} = 5$. We would expect our model to predict $y = 6$. This is a feature of generalization: predicting for new $\mathbf{x}$.

§1.1.2 Unsupervised learning

In unsupervised learning, we are given data points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$, and we want to find some interesting structure in the data. For example, we might want to find clusters of similar data points, or find a low-dimensional representation of the data. In unsupervised learning, we are not given any labels $y_1, \ldots, y_n \in \mathcal{Y}$ for the data points. The goal is to find some structure in the data that can be useful for some other task. For example, clustering can be useful for finding groups of similar data points, and dimensionality reduction can be useful for visualizing high-dimensional data.

§1.1.3 Reinforcement learning

In reinforcement learning, an agent interacts with an environment over time. At each time step t, the agent receives an observation $x_t \in \mathcal{X}$, takes an action $a_t \in \mathcal{A}$, and receives a reward $r_t \in \mathbb{R}$. The goal of the agent is to learn a policy $\pi \colon \mathcal{X} \to \mathcal{A}$ that maps observations to actions in a way that maximizes the expected sum of rewards over time. Reinforcement learning is concerned with learning a good policy for an agent to follow in an environment.

§1.1.4 Other types of learning

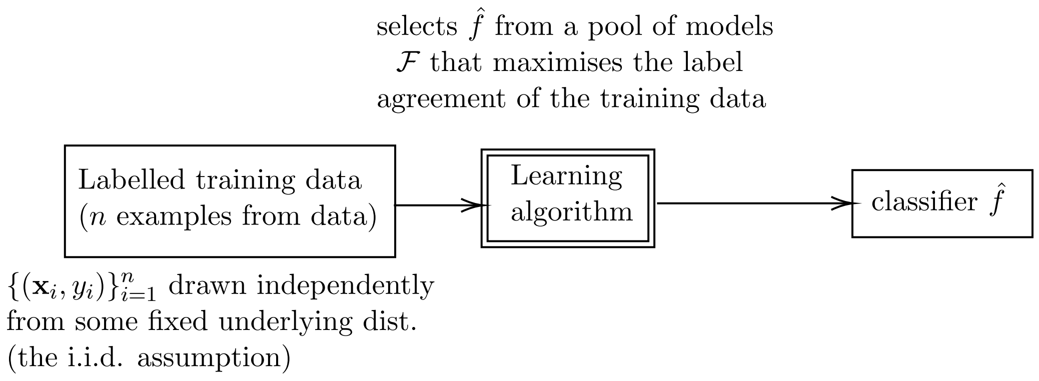
There are many other types of learning that we will not cover in this course. For example, semi-supervised learning is a type of learning where we have a small amount of labelled data and a large amount of unlabelled data, and we want to use both types of data to learn a good model. Active learning is a type of learning where the learner can interactively query the user (or some other information source) to obtain the labels of some unlabelled data points. Online learning is a type of learning where the learner receives data points one at a time and must make predictions for each data point before receiving the next one.

§2 Lecture 02—18th January, 2023

§2.1 Supervised machine learning

Last time, we introduced supervised learning and the problem of classification. We discussed the problem of learning a function that maps input data to output labels. We also discussed the problem of learning a function that maps input data to output real numbers.

Note that the distribution of the input data (over a continuous space) is not uniform; if it were, then the distribution function does not converge because it is over the real line, and it is hard to learn—the uniform distribution is the hard the distribution for proper learning.


selects $\hat{f}$ from a pool of models $\mathcal{F}$ that maximises the label agreement of the training data
Labelled training data (n examples from data)
Learning algorithm
classifier $\hat{f}$
$\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ drawn independently from some fixed underlying dist. (the i.i.d. assumption)

This semester, we will always assume that the data is drawn i.i.d. from whatever underlying process there is.

The natural question is then the following: how might we select the function $\hat{f} \in \mathcal{F}$? There are a variety of techniques, which we will cover as the semester goes on:

- Maximum likelihood techniques, which are based on the idea of finding the function that best fits the data.
- Maximum a posteriori techniques, which are based on the idea of finding the function that best fits the data, given some prior knowledge about the function.
- Optimising a "loss" criterion, which is based on the idea of finding the function that best discriminates the labels, given the data.
- Regularisation techniques, which are based on the idea of finding the function that best fits the data, but with some constraints on the function.
- Other methods...

§2.1.1 The classifier

So at this point we know that given a joint input-output space $\mathcal{X} \times \mathcal{Y}$ where the data is distributed according to some distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. Then a classifier is simply a measurable function of the type $\hat{f} : \mathcal{X} \to \mathcal{Y}$. As we know, there are many such functions; which do we pick? Here are some examples:

1. **The constant classifier**: This is the classifier that always outputs the same label, regardless of the input. That is, for all inputs $\mathbf{x} \in \mathcal{X}$ and for some fixed $y \in \mathcal{Y}$, we have $\hat{f}(\mathbf{x}) = y$. This is a very simple classifier, but it is not very useful in practice.
2. **The threshold classifier**: This is the classifier that outputs one label if the input is above a certain threshold, and another label if the input is below the threshold. That is, for all inputs $\mathbf{x} \in \mathcal{X}$ and for some fixed $t \in \mathbb{R}$, we have

$$\hat{f}(\mathbf{x}) = \begin{cases} y_1 & \text{if } \mathbf{x} > t \\ y_2 & \text{if } \mathbf{x} \leq t, \end{cases}$$

where $\mathcal{X} = \mathbb{R}$ and $\mathcal{Y} = \{y_1, y_2\}$. In some sense, we can think of this classifier as separating a projection of the space on $\mathbb{R}^2$ into two regions, and then assigning a label to each region.
3. **The majority class classifier**: This is the classifier that outputs the label that appears most frequently in the training data. That is, for all inputs $\mathbf{x} \in \mathcal{X}$, we have

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \sum_{i=1}^{n} \mathbb{1}\{y_i = y\},$$

where $\mathbb{1}\{\cdot\}$ is the indicator function. In terms of probabilities, this can also be written as

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \Pr[Y = y],$$

where Y is the random variable representing the label.

4. **The Bayes classifier**: This is the classifier that outputs the label that maximises the conditional probability of the label given the input. That is, for all inputs $\mathbf{x} \in \mathcal{X}$, we have

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \Pr[Y = y \mid X = \mathbf{x}],$$

   where Y is the random variable representing the label, and X is the random variable representing the input. This is the best possible classifier, in the sense that it minimises the probability of error. However, it is not always possible to compute this classifier in practice, because it requires knowledge of the distribution $\mathcal{D}$. In practice, we often approximate this classifier using the training data.

Now what might be a reasonable/good choice for the function $f \colon \mathcal{X} \to \mathcal{Y}$? We want to do prediction, so ideally we want to select a function that is actually good at prediction; in particular, on any example pair $(\mathbf{x}, y)$ sampled from $\mathcal{D}$, we want the function f to be such that $f(\mathbf{x}) = y$. This motivates the following definition:

Definition 2.1 (Accuracy)**.** *The* accuracy *of a classifier f, denoted* $\mathrm{acc}(f)$*, for some distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, is defined as*

$$\mathrm{acc}(f) = \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[f(\mathbf{x}) = y] = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}\{f(\mathbf{x}) = y\}],$$

where $\mathbb{1}\{\cdot\}$ is the indicator function. The error *of a classifier f, denoted* $\mathrm{err}(f)$*, is defined as* $\mathrm{err}(f) = 1 - \mathrm{acc}(f)$*.*

So from our discussion above, we want a classifier f with highest possible accuracy $\mathrm{acc}(f)$.

§2.1.2 The Bayes classifier

Let us now study the Bayes classifier in more detail. The Bayes classifier is the classifier that maximises the conditional probability of the label given the input:

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[Y = y \mid X = \mathbf{x}].$$

This is a very good classifier; in fact, it is the best possible classifier, in the sense that it minimises the probability of error, as we will soon see.

Before that, we first try to understand the classifier itself. Immediately, we can see that it is both input- and distribution-dependent; our proof must take this into account and work regardless of the input and distribution. We have already seen that the accuracy of a particular classifier f is

$$\mathrm{acc}(f) = \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[f(\mathbf{x}) = y] = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}\{f(\mathbf{x}) = y\}].$$

Now assume without loss of generality that the classification is binary, that is, $\mathcal{Y} = \{0, 1\}$ (if not, we can always map the labels to $\{0, 1\}$). Furthermore, take

$$f(\mathbf{x}) = \arg\max_{y \in \{0,1\}} \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[Y = y \mid X = \mathbf{x}]$$

to be the Bayes classifier, and $g \colon \mathcal{X} \to \{0, 1\}$ to be any other classifier. Then the Bayes classifier is optimal, as the following theorem shows.

Theorem 2.1. *For any classifier $g\colon \mathcal{X} \to \{0,1\}$, we have*

$$\mathrm{acc}(g) \le \mathrm{acc}(f),$$

where f is the Bayes classifier.

Proof. Fix any $\mathbf{x} \in \mathcal{X}$. Then for any classifier $h\colon \mathcal{X} \to \{0,1\}$, we have

$$\begin{aligned}
\Pr_{(\mathbf{x},y)\sim\mathcal{D}}[h(\mathbf{x}) = y] &= \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[h(\mathbf{x}) = y \mid X = \mathbf{x}] \\
&= \Pr\left[h(\mathbf{x}) = 1, Y = 1 \mid X = \mathbf{x}\right] + \Pr\left[h(\mathbf{x}) = 0, Y = 0 \mid X = \mathbf{x}\right] \\
&= \mathbb{1}\{h(\mathbf{x}) = 1\} \Pr\left[Y = 1 \mid X = \mathbf{x}\right] + \mathbb{1}\{h(\mathbf{x}) = 0\} \Pr\left[Y = 0 \mid X = \mathbf{x}\right] \\
&= \mathbb{1}\{h(\mathbf{x}) = 1\}\eta(\mathbf{x}) + \mathbb{1}\{h(\mathbf{x}) = 0\}(1 - \eta(\mathbf{x})),
\end{aligned}$$

where the randomness is over the true label y and $\eta(\mathbf{x}) = \Pr[Y = 1 \mid X = \mathbf{x}]$. Setting $\Delta := \Pr[f(\mathbf{x}) = y \mid X = \mathbf{x}] - \Pr[g(\mathbf{x}) = y \mid X = \mathbf{x}]$, we get

$$\begin{aligned}
\Delta &= \eta(\mathbf{x})\left(\mathbb{1}\{f(\mathbf{x}) = 1\} - \mathbb{1}\{g(\mathbf{x}) = 1\}\right) + (1 - \eta(\mathbf{x}))\left(\mathbb{1}\{f(\mathbf{x}) = 0\} - \mathbb{1}\{g(\mathbf{x}) = 0\}\right) \\
&= (2\eta(\mathbf{x}) - 1) \cdot \left(\mathbb{1}\{f(\mathbf{x}) = 1\} - \mathbb{1}\{g(\mathbf{x}) = 1\}\right) \\
&\ge 0,
\end{aligned}$$

by the choice of f and by the law of total probability. The result follows after integrating over all the $\mathbf{x}$ to remove the conditional. $\square$

§3 Lecture 03—23rd January, 2023

Last time we saw the optimality of the Bayes classifier:

Theorem 3.1. *For any classifier $g\colon \mathcal{X} \to \{0,1\}$, we have*

$$\mathrm{acc}(g) \le \mathrm{acc}(\hat{f}),$$

where $\hat{f}$ is the Bayes classifier.

Note that the assumption we made that $\mathcal{Y} \in \{0,1\}$ is useful because we assume that the output space is a finite element space (we will come to see infinite element spaces in situations like regression, etc.).

Now, if we have the best classifier, can't we simply solve all the classification problems in the world? No—on any input $\mathbf{x}$, we need to compute

$$\hat{f}(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \Pr[Y = y \mid X = \mathbf{x}]$$

But we don't know what the true data distribution is, so we don't really know what $\Pr[Y = y \mid X = \mathbf{x}]$ is! We need to estimate it from the observed data samples $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, and try to improve upon the quality of the estimate as we get more data.

The issue though is that there are practically infinitely many $\mathbf{x} \in \mathcal{X}$, and so there are also infinitely many densities $\Pr[Y = y \mid X = \mathbf{x}]$ to estimate. This is a problem because we can't possibly estimate infinitely many things from finitely many samples. But observe that, given the Bayes classifier

$$\hat{f}(\mathbf{x}) = \arg\max_{y\in\mathcal{Y}} \Pr[Y = y \mid X = \mathbf{x}],$$

we can instead estimate the *conditional class probabilities* $\Pr[Y = y \mid X = \mathbf{x}]$ for each $y \in \mathcal{Y}$ in the following way:

$$\begin{aligned} f(\mathbf{x}) &= \arg\max_{y\in\mathcal{Y}} \Pr[Y = y \mid X = \mathbf{x}] \\ &= \arg\max_{y\in\mathcal{Y}} \frac{\Pr[Y = y, X = \mathbf{x}]}{\Pr[X = \mathbf{x}]} \\ &= \arg\max_{y\in\mathcal{Y}} \frac{\Pr[X = \mathbf{x} \mid Y = y]\Pr[Y = y]}{\Pr[X = \mathbf{x}]} \\ &= \arg\max_{y\in\mathcal{Y}} \underbrace{\Pr[X = \mathbf{x} \mid Y = y]}_{\text{class cond. prob. model}} \underbrace{\Pr[Y = y]}_{\text{class prior}}, \end{aligned}$$

by Bayes' rule and where the last step is due to the fact that $\Pr[X = \mathbf{x}]$ is the same for all $y \in \mathcal{Y}$ and so doesn't affect the maximization. Now this is an improvement, because $|\mathcal{Y}|$ is finite and usually small, so the number of distributions to estimate is now finite and small.

So how do we estimate these conditional densities (like $\Pr[X = \mathbf{x} \mid Y = y]$) or class prior densities (like $\Pr[Y = y]$)? There are multiple ways of estimating these densities, each with its own trade-offs (indeed, *ceteris paribus*, the number of samples required for the second is exponentially larger than the first).

- If the particular form of the data density/distribution is known/assumed, we can assume some sort of parametric density estimation; that is, we assume that there is a functional form of the densities (which may not be true in general but might be reasonable for certain types of data) controlled by some parameters (usually the mean and the covariance structure). Techniques in this area include likelihood estimation, maximum a posteriori estimation, mixture models, etc.
- If it is inappropriate to assume a specific distribution, then we can simply use non-parametric density estimation methods like kernel density estimation, nearest neighbor estimation, parzen windows, histograms, etc.

§3.1 Maximum likelihood estimation

Recall that in maximum likelihood estimation, we are given independent and identically distributed data samples $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{X}$, and we have a model class $\mathcal{P} := \{p_\theta : \theta \in \Theta\}$ of models p which are described by a set of parameters θ. The goal is then to find the paraneter settings $\hat{\theta}$ that best fit—i.e. maximize the likelihood of—the observed data. TIf each model p is a probability model, then we can find the best fitting model by maximising the likelihood function defined as

$$\mathcal{L}(\theta \mid X) := \Pr[X = \mathbf{x}_1, \dots, X = \mathbf{x}_n \mid \theta] = \prod_{i=1}^{n} \Pr[X = \mathbf{x}_i \mid \theta].$$

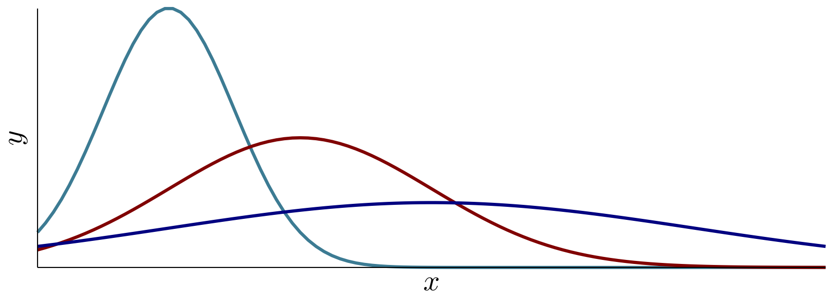
In other words, we want to find how probable (or how likely) the observed data is, given the model together with its parameters, p_θ. The maximum likelihood estimate $\hat{\theta}$ is then defined as

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta \mid X) = \arg\max_{\theta \in \Theta} \prod_{i=1}^{n} \Pr[X = \mathbf{x}_i \mid \theta].$$

We will examine more of this in the following example.

§3.1.1 Example for maximum likelihood estimation

Suppose we wanted to fit a statistical probability model to the heights of adult students, and we have height data $60, 62, 53, 58, \ldots \in \mathbb{R}$, more generally $x_1, \ldots, x_n \in \mathcal{X} = \mathbb{R}$. We can assume that the heights are normally distributed as a Gaussian distribution in $\mathbb{R}$,



and we want to tell which of the above three distributions best fits the data. Our model class is then $\mathcal{P} = \{p_{\mu,\sigma^2} : \mu \in \mathbb{R}, \sigma^2 > 0\}$, where p_{μ,σ^2} is the Gaussian distribution

$$p_\theta = p_{\{\mu,\sigma^2\}} := \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

Here $\theta = \{\mu, \sigma^2\}$ are the parameters of the model (the mean and the variance). We are interested in the most likely setting of the mean and the variance given the observed data. That is, by maximum likelihood estimation, we want

$$\arg\max_{\theta \in \Theta} \mathcal{L}(\theta \mid X) = \arg\max_{\mu \in \mathbb{R}, \sigma^2 > 0} \prod_{i=1}^{n} p_{\mu,\sigma^2}(x_i).$$

To maximise $\mathcal{L}$, we may want to take the derivative with respect to the controlling variable θ, which is the 2-dimensional vector $\{\mu, \sigma^2\}$. However, the product of the densities is not easy to differentiate, so we take the logarithm of the likelihood function to make it easier to differentiate (since the logarithm is a monotonic function, the maximum of the log-likelihood is the same as the maximum of the likelihood function). We then have

$$\begin{aligned} \arg\max_{\theta \in \Theta} \mathcal{L}(\theta \mid X) &= \arg\max_{\mu \in \mathbb{R}, \sigma^2 > 0} \log \mathcal{L}(\theta \mid X) \\ &= \arg\max_{\mu \in \mathbb{R}, \sigma^2 > 0} \log\left(\prod_{i=1}^{n} p_{\mu,\sigma^2}(x_i)\right) \end{aligned}$$

$$
\begin{aligned}
&= \arg\max_{\mu\in\mathbb{R},\sigma^2>0} \sum_{i=1}^{n} \log p_{\mu,\sigma^2}(x_i) \\
&= \arg\max_{\mu\in\mathbb{R},\sigma^2>0} \sum_{i=1}^{n} \log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x_i-\mu)^2}{2\sigma^2}\right)\right) \\
&= \arg\max_{\mu\in\mathbb{R},\sigma^2>0} \sum_{i=1}^{n} \left(\log\left(\frac{1}{\sqrt{2\pi\sigma^2}}\right) - \frac{(x_i-\mu)^2}{2\sigma^2}\right) \\
&= \arg\max_{\mu\in\mathbb{R},\sigma^2>0} \sum_{i=1}^{n} \left(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x_i-\mu)^2}{2\sigma^2}\right).
\end{aligned}
$$

Now let $g_i(\mu,\sigma^2) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x_i-\mu)^2}{2\sigma^2}$, so that the derivative of g_i with respect to μ is

$$\frac{\partial g_i}{\partial \mu} = \frac{x_i-\mu}{\sigma^2},$$

and the derivative of g_i with respect to σ^2 is

$$\frac{\partial g_i}{\partial \sigma^2} = -\frac{1}{2\sigma^2} + \frac{(x_i-\mu)^2}{2\sigma^4}.$$

We then have

$$
\begin{aligned}
\frac{\partial}{\partial\mu}\sum_{i=1}^{n} g_i(\mu,\sigma^2) &= \sum_{i=1}^{n}\frac{\partial g_i}{\partial\mu} = \sum_{i=1}^{n}\frac{x_i-\mu}{\sigma^2} = 0, \\
\frac{\partial}{\partial\sigma^2}\sum_{i=1}^{n} g_i(\mu,\sigma^2) &= \sum_{i=1}^{n}\frac{\partial g_i}{\partial\sigma^2} = \sum_{i=1}^{n}\left(-\frac{1}{2\sigma^2} + \frac{(x_i-\mu)^2}{2\sigma^4}\right) = 0.
\end{aligned}
$$

Solving the above equations, we get

$$
\begin{aligned}
\mu &= \frac{1}{n}\sum_{i=1}^{n} x_i, \\
\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2,
\end{aligned}
$$

Therefore the maximum likelihood estimate of the mean and the variance of the Gaussian distribution is the sample mean and the sample variance of the observed data, respectively. This is a very important result, and it is quite common result in statistics.

Let's now go back to our quandary with the Bayes classifier estimates; we saw that:

$$f(\mathbf{x}) = \arg\max_{y\in\mathcal{Y}} \Pr[X=\mathbf{x} \mid Y=y]\Pr[Y=y].$$

We will estimate $\Pr[X=\mathbf{x} \mid Y=y]$ and $\Pr[Y=y]$ using maximum likelihood estimation with two functional distribution forms; for $\Pr[X=\mathbf{x} \mid Y=y]$, we will use the high-dimensional Gaussian distribution, and for $\Pr[Y=y]$, we will use the multinomial distribution. In particular, when Y takes value 1, we can assume a (univariate) Gaussian distribution on $\mathcal{X}$, and when Y takes value 0, we can assume another (univariate) Gaussian distribution on $\mathcal{X}$, etc.

To clarify things moving forward, let's review some of the basic probability models:

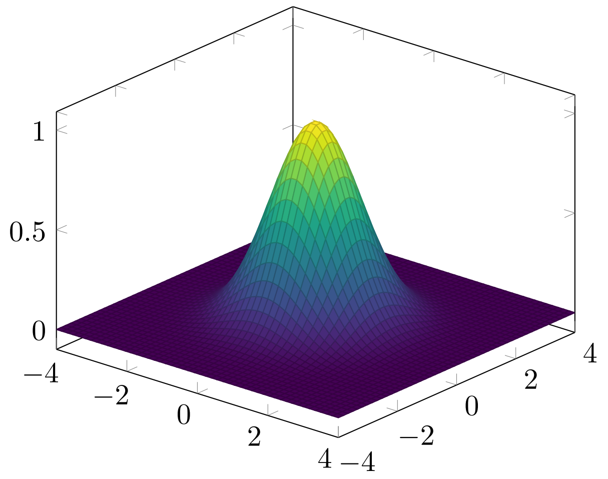
- **Bernoulli distribution**: This is a distribution over a binary random variable $X \in \{0, 1\}$, and is parameterised by a single parameter $\theta \in [0, 1]$. The probability mass function of the Bernoulli distribution is
$$\Pr[X = x \mid \theta] = \theta^x (1 - \theta)^{1-x},$$
for $x \in \{0, 1\}$.

- **Multinomial distribution**: This is a distribution over a categorical random variable (used to model the outcome of a multi-class event, like rolling a die), and is parameterised by the number of classes $k \in \mathbb{N}$ (so that the outcome space is $X\{1, \dots, k\}$), and is parameterized by a vector of k parameters $\theta = (\theta_1, \dots, \theta_k)$ such that $\theta_i \geq 0$ and $\sum_{i=1}^{k} \theta_i = 1$. The probability mass function of the multinomial distribution is
$$\Pr[X = x \mid \theta] = \prod_{i=1}^{k} \theta_i^{x_i},$$
for $x \in \{0, 1\}^k$ such that $\sum_{i=1}^{k} x_i = 1$.

- **Poisson model**: This is a distribution over the non-negative integers (used to model rare counting events), and is parameterised by a single parameter $\lambda > 0$. The probability mass function of the Poisson distribution is
$$\Pr[X = x \mid \lambda] = \frac{e^{-\lambda} \lambda^x}{x!},$$
for $x \in \{0, 1, 2, \dots\}$.

- **Gaussian distribution**: This is a distribution over the real numbers, and is parameterised by a mean $\mu \in \mathbb{R}$ and a variance $\sigma^2 > 0$. The probability density function of the Gaussian distribution is
$$\Pr[X = x \mid \mu, \sigma^2] = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right).$$

- **Multivariate Gaussian distribution**: This is a distribution over the real vectors, and is parameterised by a mean vector $\mu \in \mathbb{R}^d$ and a covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$ such that $\Sigma \succeq 0$. The probability density function of the multivariate Gaussian distribution is
$$\Pr[X = x \mid \mu, \Sigma] = \frac{1}{(2\pi)^{d/2} \det(\Sigma)^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^\top \Sigma^{-1} (x - \mu)\right).$$



Most machine learning data is vector-valued, and so the multivariate Gaussian distribution is a very important distribution for us.

§4 Lecture 04—25th January, 2023

§4.1 Classification via maximum likelihood estimation—an example

Suppose now that our task is to learn a classifier that can correctly (or at least most of the time) identify an individual as male or female, based on collecting, say, just their height and weight measurements. In this case,

$$\begin{aligned}\mathcal{X} &= \mathbb{R}^2 = \{(\text{height}, \text{weight})\},\\ \mathcal{Y} &= \{\text{male}, \text{female}\}.\end{aligned}$$

For some population of interest, we can assume that the data is distributed over $\mathcal{X} \times \mathcal{Y}$ according to some joint distribution $\mathcal{D}$ which we don't know, but is fixed (if the distribution is variable, then we have a non-stationary learning problem, which is harder). Now we randomly draw points of labelled data $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n) \sim_{\text{i.i.d.}} \mathcal{D}$. For this class, the Bayes classifier is

$$f(\mathbf{x}) = \arg \max_{y \in \{\text{male,female}\}} \Pr_{\mathcal{D}|_{X|Y=y}} [X = \mathbf{x} \mid Y = y] \Pr_{\mathcal{D}|_Y} [Y = y],$$

where $\Pr_{\mathcal{D}|_{X|Y=y}}[X = \mathbf{x} \mid Y = y]$ is the conditional density of the data given the class over the conditional distribution $\mathcal{D}|_{X|Y=y}$, and $\Pr_{\mathcal{D}|_Y}[Y = y]$ is the class prior density over the marginal distribution $\mathcal{D}|_Y$. Now using labelled training data, we want to learn all the class parameters. We can estimate the class priors as follows; $\hat{\Pr}[Y = \text{ male}]$ is the fraction of the training data labelled as male, and $\hat{\Pr}[Y = \text{ female}]$ is the fraction of the training data labelled as female (note that we are using a binomial model here to estimate the class priors via maximum likelihood estimation). To estimate the class conditional densities, we take

$$\begin{aligned}\hat{\Pr}[X = \mathbf{x} \mid Y = \text{ male}] &= p_{\theta(\text{male})}(\mathbf{x}),\\ \hat{\Pr}[X = \mathbf{x} \mid Y = \text{ female}] &= p_{\theta(\text{female})}(\mathbf{x}),\end{aligned}$$

where $\theta(\text{male})$ and $\theta(\text{female})$ can be estimated via maximum likelihood estimation of male and female data respectively, modelled using a bivariate Gaussian distribution. We can then use these estimates to construct our Bayes classifier.

By so doing, we have created our first predictor $\hat{f}$! It's important to keep in mind that there were many assumptions made in the process of creating this predictor, and that the predictor is only as good as the strength of the assumptions made. For example, we're assuming that the data is distributed according to a bivariate Gaussian distribution, and that the class conditional densities are Gaussian. We're also assuming that the class priors are estimated correctly, and that the training data is representative of the population of interest. If any of these assumptions are violated, then our predictor will not perform well. This is why it's important to always keep in mind the assumptions made when creating a predictor, and to always be sceptical of the results of a predictor.

A problem For the Bayes classifier, we can show that if p is the true class conditional density, and $\hat{p}$ is the estimated class conditional density, then the inaccuracy/discrepancy of the predictor $\hat{f}$ is given by the sum squared error

$$\int_{\mathcal{X}} (p(\mathbf{x}) - \hat{p}(\mathbf{x}))^2 \, \mathrm{d}\mathbf{x} \approx n^{-2/(2+d)} \approx \frac{1}{\sqrt[d]{n}},$$

where d is the dimension of the input space, and n is the number of training examples. Now as $n \to \infty$, the inaccuracy of the predictor $\hat{f}$ goes to zero regardless of d, and the predictor $\hat{f}$ becomes consistent—it gives the right answer in the limit.

However, as a function of d, the inaccuracy of the predictor $\hat{f}$ goes to zero at a rate of $1/\sqrt[d]{n}$, which is very slow; in fact, for it to cross the halfway point, we need $n \geq 2^d$ samples—the number of samples required (based on the feature space) must be at least exponential in the dimensionality–this is the curse of dimensionality, that we cannot get away with few samples. So we may want to relax the severity of our initial assumptions, and consider a more flexible class of predictors. This is where the naïve Bayes classifier comes in.

§4.2 The naïve Bayes classifier

The naïve Bayes classifier is a simple and effective classifier that is based on the assumption that the input features are conditionally independent given the class. This is a very strong assumption, but it is also a very useful one, as it allows us to estimate the class conditional densities using univariate models, and then combine them to form a multivariate model. In particular, the classifier is

$$\begin{aligned}\hat{f}(\mathbf{x}) &= \arg\max_{y \in \mathcal{Y}} \Pr\left[X = \mathbf{x} \mid Y = y\right] \cdot \Pr\left[Y = y\right] \\ &= \arg\max_{y \in \mathcal{Y}} \Pr\left[Y = y\right] \prod_{j=1}^{d} \Pr\left[X^{(j)} = x^{(j)} \mid Y = y\right] \cdot \Pr\left[Y = y\right],\end{aligned}$$

where the crux of the classifier is the assumption that the individual input features are conditionally independent given the class label.

We are okay with this assumption in practice (e.g. for spam detection and filtering, etc.), because it is gives us an acceptable solution without having to estimate a very high-dimensional class conditional density. The naïve Bayes classifier is a very simple and effective classifier, and it is often used as a baseline for more complex classifiers. It is also very simple, fast to train and to use, and it is very robust to overfitting. However, it is also very sensitive to the strong assumption of conditional independence, and it is not very good at capturing the structure of potentially interdependent data (and can give bad estimates as a result).

§4.3 Evaluating the quality of a classifier

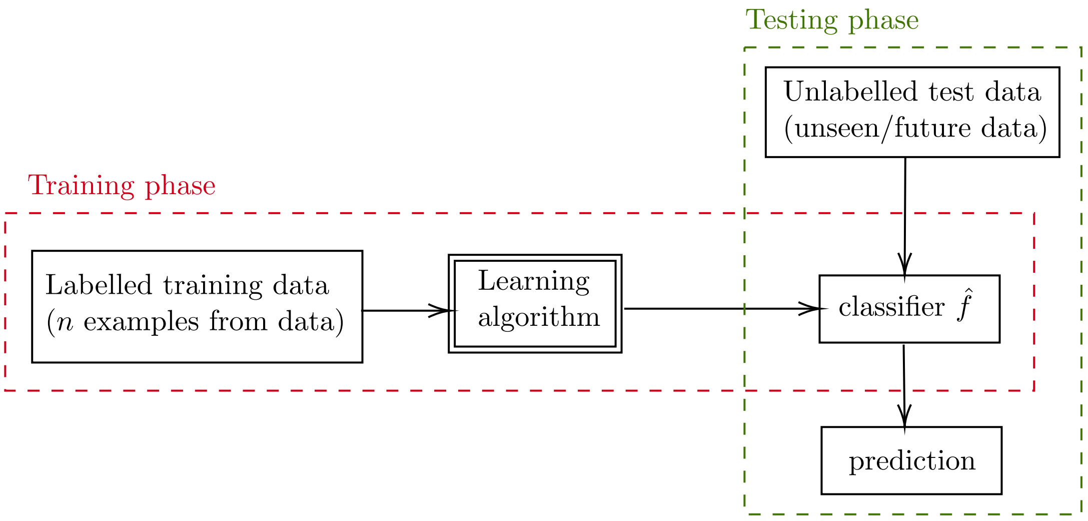
We have seen how to construct a classifier, but how do we know if it's any good, especially compared to other classifiers? We already have the accuracy as a gold standard:

$$\mathrm{acc}(f) = \Pr_{(\mathbf{x},y)\sim\mathcal{D}}[f(\mathbf{x}) = y] = \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}[\mathbb{1}\{f(\mathbf{x}) = y\}],$$

and so we just need to compute $\mathrm{acc}(\hat{f})$ for our classifier $\hat{f}$, and compare it to the accuracy of other classifiers. But once again we run into the same issue: we don't know the distribution $\mathcal{D}$, and so we can't compute the accuracy of our classifier. We can only estimate the accuracy of our classifier using the training data, and this sample estimate for the accuracy of our classifier is called the *training accuracy*:

$$\widehat{\mathrm{acc}}(\hat{f}) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{\hat{f}(\mathbf{x}_i) = y_i\}.$$

However, this severely overestimates the accuracy of our classifier, because the classifier was trained on the training data which was already used to construct $\hat{f}$, so it is not an unbiased estimator. Instead, what we might want to do first is to hold out a portion of the training data, called the *test set*, use only the rest of the training data to learn $\hat{f}$, and use the test data set to unbiasedly estimate the accuracy of our classifier. This is called *cross-validation*, and it is a very important technique in machine learning.


Testing phase
Unlabelled test data (unseen/future data)
Training phase
Labelled training data (n examples from data)
Learning algorithm
classifier $\hat{f}$
prediction

We can guarantee that with a sufficiently large test sample, across all test samples, the deviation of the estimate from the ground truth $\mathrm{acc}(\hat{f})$ is small.

Now we have seen the Bayes classifier and some of its conceptual relations that will be useful for us across machine learning. Because we almost never have access to the true underlying joint distribution, let us now take a look at a fundamental approach to classification that seems distribution-free, namely, the method of nearest neighbours (NN).

§4.4 Nearest neighbours

This approach is akin to taking a view opposite to the Bayes classifier: whereas Bayes assumes full knowledge and takes advantage of the data distribution, Nearest neighbors (NN) completely ignores the underlying probability distribution. At a high level, the NN method is based on the belief that features that are used to describe the data are relevant to the labelling in a way that makes "close" points likely to have the same label.

NN is one of the simplest possible classifiers, where the training process is essentially to memorize the training data. Then during testing, for a given point x, it finds the k training points that are closest to x, and predicts the weighted majority label among these k points. More precisely, given training data $S = \{(\mathbf{x}_i, y_i) : 1 \leq i \leq N\}$, the nearest neighbour prediction rule is

$$\mathrm{NN}_k(\mathbf{x}) \coloneqq \{j : \mathbf{x}_j \text{ is within the } k \text{ closest to } \mathbf{x} \text{ for } 1 \leq j \leq k\}.$$

We can measure the closeness between two examples $\mathbf{x}_1$ and $\mathbf{x}_2$ in many ways:

1. *Compute some sort of distance.* In this case, the smaller the distance, the closer the examples. In particular, if $\mathcal{X} = \mathbb{R}^d$, then there are a few natural ways of computing the distance:
   - The Euclidean distance:
   $$\begin{aligned}\rho(\mathbf{x}_1, \mathbf{x}_2) &= \left[\left(\mathbf{x}_1^{(1)} - \mathbf{x}_2^{(1)}\right)^2 + \cdots + \left(\mathbf{x}_1^{(d)} - \mathbf{x}_2^{(d)}\right)^2\right]^{1/2} \\ &= \left[(\mathbf{x}_1 - \mathbf{x}_2)^\top (\mathbf{x}_1 - \mathbf{x}_2)\right]^{1/2} = \|\mathbf{x}_1 - \mathbf{x}_2\|_2.\end{aligned}$$
   - Other normed distances:
   $$\rho(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\|_p = \left[\left(\mathbf{x}_1^{(1)} - \mathbf{x}_2^{(1)}\right)^p + \cdots + \left(\mathbf{x}_1^{(d)} - \mathbf{x}_2^{(d)}\right)^p\right]^{1/p}.$$
   When $p = 1$, we have the Manhattan distance, and when $p = \infty$, we have the Chebyshev distance (i.e. the maximum difference between the coordinates). When $p = 2$, we have the Euclidean distance, and when $p = 0$, we have the count-non-zero distance.
2. *Compute some sort of similarity.* In this case, the larger the similarity, the closer the examples. In particular, if $\mathcal{X} = \mathbb{R}^d$, then there are a few natural ways of computing the similarity:
   - The cosine similarity:
   $$\rho(\mathbf{x}_1, \mathbf{x}_2) = \frac{\mathbf{x}_1^T \mathbf{x}_2}{\|\mathbf{x}_1\|_2 \|\mathbf{x}_2\|_2}.$$
   - The Pearson correlation coefficient:
   $$\rho(\mathbf{x}_1, \mathbf{x}_2) = \frac{\sum_{i=1}^d (\mathbf{x}_1^{(i)} - \bar{\mathbf{x}}_1)(\mathbf{x}_2^{(i)} - \bar{\mathbf{x}}_2)}{\sqrt{\sum_{i=1}^d (\mathbf{x}_1^{(i)} - \bar{\mathbf{x}}_1)^2} \sqrt{\sum_{i=1}^d (\mathbf{x}_2^{(i)} - \bar{\mathbf{x}}_2)^2}},$$
   where $\bar{\mathbf{x}}_1$ and $\bar{\mathbf{x}}_2$ are the means of $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively.
   - Other similarity measures:
   $$\rho(\mathbf{x}_1, \mathbf{x}_2) = \frac{1}{1 + \|\mathbf{x}_1 - \mathbf{x}_2\|_p},$$
   where p is a normed distance, and
   $$\rho(\mathbf{x}_1, \mathbf{x}_2) = \sum_{i=1}^d \frac{1}{1 + \left|\mathbf{x}_1^{(i)} - \mathbf{x}_2^{(i)}\right|^2}.$$
3. *Compute closeness using domain expertise.* In this case, the closeness is determined by some domain-specific knowledge. Clearly the closeness differs across domains:

- The edit distance is a measure of similarity between two strings, and is the minimum number of operations (insertions, deletions, and mutations) required to transform one string into the other. In DNA analysis, the edit distance is used to measure the similarity between two DNA sequences.
- The Kendell-Tau distance is a measure of similarity between two rankings, and is the number of pairwise disagreements between two rankings. That is, given a ranking $\mathbf{x}_1 = (a_1, a_3, a_5, a_2, a_4)$ and a ranking $\mathbf{x}_2 = (a_3, a_1, a_2, a_5, a_4)$, the Kendell-Tau distance is 3—it is the minumum number of adjacent swaps required to transform one ranking into the other (i.e. the bubble sort distance).

§5 Lecture 05—30th January, 2023

Last time, we started talking about k-nearest neighbour classification. Note that there are some issues with this method of classification: for one, it is sensitive to noise in data, so labelling is quite unstable. However, we can make it stable by taking a majority over the k nearest neighbours. We also noted that the choice of k is important, and that we can use cross-validation to choose the best k.

§5.1 Optimality of the k-nearest neighbour classifier

Despite its simplicity, the k-nearest neighbour classifier is actually quite capable of learning complex nonlinear classifiers, as the following statement shows.

Theorem 5.1. *For fixed $k \in \mathbb{N}$, as $n \to \infty$, the k-nearest neighbour classifier converges to no more than twice the Bayes classifier error.*

Proof. Consider the error in the limit for a fixed test point $\mathbf{x}_t$. Let the nearest neighbour error rate be $\Pr[e]$ (the ratio of the number of correct classifications to the number of total classifications), let $D_n = (\mathbf{X}_n, Y_n)$ be the size n labelled training data, and let x_n be the nearest neighbour of $\mathbf{x}_t$ in D_n with label y_n. Then, we have

$$\begin{aligned}
\lim_{n\to\infty} \Pr_{y_t, D_n}[e \mid \mathbf{x}_t] &= \lim_{n\to\infty} \int \Pr_{y_t, Y_n}[e \mid \mathbf{x}_t, X_n] \Pr[X_n \mid \mathbf{x}_t] \, \mathrm{d}\mathbf{X}_n \\
&= \lim_{n\to\infty} \int \Pr_{y_t, Y_n}[e \mid \mathbf{x}_t, \mathbf{x}_n] \Pr[\mathbf{x}_n \mid \mathbf{x}_t] \, \mathrm{d}\mathbf{x}_n \\
&= \lim_{n\to\infty} \int \Pr_{y_t, Y_n}\left(1 - \sum_{y\in\mathcal{Y}} \Pr[y_t = y, y_n = y \mid \mathbf{x}_t, \mathbf{x}_n]\right) \Pr[\mathbf{x}_n \mid \mathbf{x}_t] \, \mathrm{d}\mathbf{x}_n \\
&= \lim_{n\to\infty} \int \left(1 - \sum_{y\in\mathcal{Y}} \Pr[y_t = y \mid \mathbf{x}_t] \Pr[y_n = y \mid \mathbf{x}_n]\right) \Pr[\mathbf{x}_n \mid \mathbf{x}_t] \, \mathrm{d}\mathbf{x}_n \\
&= 1 - \sum_{y\in\mathcal{Y}} \left(\Pr[y_t = y \mid \mathbf{x}_t]\right)^2,
\end{aligned}$$

where the last equality follows from the dominating convergence theorem and the fact that $\mathbf{x}_n$ is the nearest neighbour of $\mathbf{x}_t$ in the limit, the inequality before that follows from the i.i.d. assumption, and the equality before that follows from the property of the 1-nearest neighbour classifier. So we have obtained that in the case of a 1-nearest neighbour, for a fixed test point $\mathbf{x}_t$,

$$\lim_{n\to\infty} \Pr_{y_t, D_n}[e \mid \mathbf{x}_t] = 1 - \sum_{y\in\mathcal{Y}} \left(\Pr[y_t = y \mid \mathbf{x}_t]\right)^2 .$$

Now let $\Pr^*[e]$ be the Bayes classifier error rate, and let $\Pr^*[e \mid \mathbf{x}_t]$ be the Bayes classifier error rate for a fixed test point $\mathbf{x}_t$. Then, if the Bayes classifier returns y^* at some point $\mathbf{x}_t$, we have

$$\begin{aligned} 1 - \sum_{y\in\mathcal{Y}} \left(\Pr[y_t = y \mid \mathbf{x}_t]\right)^2 &\leq 1 - \left(\Pr[y_t = y^* \mid \mathbf{x}_t]\right)^2 \\ &\leq 2 \cdot (1 - \Pr[y_t = y^* \mid \mathbf{x}_t]) \\ &= 2 \cdot \Pr^*[e \mid \mathbf{x}_t], \end{aligned}$$

where the first inequality follows from the fact that the sum of squares is at most the square of the sum, and the second inequality follows from the fact that $1 - x \leq 2(1 - x^2)$ for $x \in [0, 1]$. Finally, we integrate over all test points to obtain

$$\lim_{n\to\infty} \Pr_{y_t, D_n}[e] \leq 2 \cdot \Pr^*[e],$$

which completes the proof. □

A "rate version" of the above results is as follows:

Corollary 5.1. *If $k \to \infty$ and $k/n \to 0$ as $n \to \infty$, then the k-nearest neighbour classifier converges to the Bayes classifier error rate.*

We don't prove this statement; both results are themselves simplifications of a rather technical result due to Stone (1977).

Theorem 5.2. *For any integrable function $f \colon \mathcal{X} \to \mathbb{R}$, any n, and any $k \leq n$, we have*

$$\sum_{i=1}^{k} \mathop{\mathbb{E}}_{\mathbf{x}_i \sim \mathcal{D}} \left[|f(\mathbf{x}_i)|\right] \leq k \cdot \gamma_d \mathbb{E}_{\mathbf{x}_0}\left[|f(\mathbf{x}_0)|\right],$$

where the constant $\gamma_d \leq \left(1 + 2/\sqrt{3 - \sqrt{3}}\right)^d$ depends only on the dimension d of the input space.

§5.2 Approaches to classification

There are two distinct paradigms for classification within supervised learning:

1. **Generative models** These models try to model the joint distribution of the features and the labels, i.e. $p(\mathbf{x}, y)$. They then use Bayes' rule to compute the conditional distribution $p(y \mid \mathbf{x})$, and use this to classify new data points. Examples of generative models include the Naive Bayes classifier and the Gaussian Mixture Model. The advantage of generative models is that they can be used to generate new data points that are similar to the training data. The disadvantage is that they can be less flexible than discriminative models, and can be more sensitive to model misspecification.

2. **Discriminative models** These models directly model the conditional distribution $p(y \mid \mathbf{x})$, and use this to classify new data points. Examples of discriminative models include logistic regression, support vector machines, and neural networks. The advantage of discriminative models is that they can be more flexible than generative models, and can be less sensitive to model misspecification. The disadvantage is that they cannot be used to generate new data points that are similar to the training data, as they give no information about the joint distribution of the features and the labels.

§5.3 Problems with the k-nearest neighbour classifier

There are generally three issues with the k-nearest neighbour classifier:

1. Finding the k nearest neighbours can be computationally expensive, especially in high dimensions.
2. The k-nearest neighbour classifier is sensitive to the choice of distance metric, and most times the 'closeness' in raw measurement space is not good enough
3. We pay a lot in terms of storage, as we need to store all the training data during test time.

We now address these issues, starting from the first.

§5.3.1 Finding the nearest neighbour quickly

Given some test example $\mathbf{x}_t$, as we have already seen, the computational cost of finding the nearest neighbour is $O(nd)$, where n is the number of training examples and d is the dimension of the input space. This is because we need to compute the distance between $\mathbf{x}_t$ and each of the n training examples, and each distance computation takes $O(d)$ time. This is not good enough for large n and d, and this is particularly problematic because most real-world data sets do have large n and d. We can address this issue by using data structures that allow us to find the nearest neighbour more quickly; in particular, we will see how we can obtain later gains after spending time to preprocess the data (using a data structure inspired by the binary search tree).

Suppose that our data set is in $\mathbb{R}$ and is equipped with a total order. Then, we can use a binary search tree to store the data set, and then use the tree to find the nearest neighbour of a test point x_t from a training pool $x_1, \ldots, x_n$ of examples—this naïve approach takes time $O(n)$, but we can improve this running time via a tree constructed as follows:

1. Start with the root node, which contains the median of the data set.
2. Recursively construct the left and right subtrees, which contain the medians of the left and right halves of the data set, respectively.

The search time of this tree is $O(\log n)$, and the processing overhead is $O(n \log n)$. This approach works nicely when we're answering the yes/no question of whether a point is in the data set, but it doesn't work so well when we're trying to find the nearest neighbour of a point for a test point that does not necessarily (and really doesn't usually) lie in the data set. We can address this by using a cut based on the median of the data set, and then recursively searching the left and right subtrees. This approach, generalised to d dimensions, is called the *k-d tree*, and we have search time $O(\log n)$ and processing overhead $O(n \log n)$.

Even though this procedure is quite efficient, it only gives us a near neighbour, and not the exact nearest neighbour (and you can convince yourself of this). In many machine learning contexts, the near neighbour is good enough for us, and we can give up on the exact nearest neighbour in exchange for a faster search time. However, the problem remains, and there are other algorithms that can address this issue, such as locality-sensitive hashing (LSH), vector quantisation, and clustering, as well as other tree-based methods such as the ball tree, navigation nets, and the metric tree.

§5.3.2 Improving the measurement space and bypassing storage issues

Sometimes, we don't know a priori which exact measurements are helpful for classification, so to rectify this we measure as much as we can for a particular observation. For example, suppose we wanted to learn a classfier to distinguish male goats from female goats, but we don't know which measurements would be helpful, so we measure a lot of things: the weight of the goat, the length of the goat, the height of the goat, the width of the goat, the colour of the goat, the eye colour of the goat, the number of legs of the goat, the number of horns of the goat, the number of ears of the goat, etc.

As we keep on making more and more measurements, it becomes likely that some of these measurements are not helpful for classification, and that some of these measurements are redundant, so that the k-nearest neighbour classifier becomes inflated; if we happen to have highly correlated features, then the k-nearest neighbour classifier will be redundant.

How then can we make our distance measurement robust? We can use the following approach: we can re-weight the contribution of each feature to the distance computation; in particular, we can use the following distance metric:

$$\begin{aligned}\rho(\mathbf{x}_1, \mathbf{x}_2; \mathbf{w}) &= \left(w_1 \cdot \left(x_1^{(1)} - x_2^{(1)} \right)^2 + \cdots + w_d \cdot \left(x_1^{(d)} - x_2^{(d)} \right)^2 \right)^{1/2}, \\ &= \left((\mathbf{x}_1 - \mathbf{x}_2)^\top \mathbf{W} (\mathbf{x}_1 - \mathbf{x}_2) \right)^{1/2},\end{aligned}$$

where $\mathbf{W} = \operatorname{diag}(w_1, \ldots, w_d)$ is a diagonal matrix of weights. How can we figure out what these optimal weights are? Essentially we want a distance metric $\rho(\mathbf{x}_1, \mathbf{x}_2; \mathbf{w})$ such that data samples from the same class are close to each other, and data samples from different classes are far from each other. One way to get this is to create two sets, a similarity set $S = \{(\mathbf{x}_i, \mathbf{x}_j) : y_i = y_j, i, j \in [n]\}$ and a dissimilarity set $D = \{(\mathbf{x}_i, \mathbf{x}_j) : y_i \neq y_j, i, j \in [n]\}$, and then use the following cost function:

$$\Psi(\mathbf{w}) = \lambda \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in S} \rho(\mathbf{x}_i, \mathbf{x}_j; \mathbf{w}) - (1 - \lambda) \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in D} \rho(\mathbf{x}_i, \mathbf{x}_j; \mathbf{w}),$$

where $\lambda \in [0, 1]$ is a trade-off parameter. We can then use Lagrange multipliers (or any other optimisation strategy) to solve the following optimisation problem:

$$\min_{\mathbf{w}} \Psi(\mathbf{w}) = \min_{\mathbf{w}} \left\{ \lambda \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in S} \rho(\mathbf{x}_i, \mathbf{x}_j; \mathbf{w})^2 - (1 - \lambda) \sum_{(\mathbf{x}_i, \mathbf{x}_j) \in D} \rho(\mathbf{x}_i, \mathbf{x}_j; \mathbf{w})^2 \right\}.$$

Now we focus our attention on mitigating the storage issue. Although it seems like we need to keep all the training data around test time, we can actually bypass this issue by using a method called *lazy learning*. The idea is that we don't actually need to store the training data at all; we can just store the tree structure that we used to find the nearest neighbour, label each of the width-r cells of the k-d tree, and then use this tree structure to find the nearest neighbour of a test point. This is a very nice idea, and it is used in many machine learning algorithms, such as the k-nearest neighbour classifier, the kernel density estimator, and the locally weighted regression algorithm. The space requirement here is then at most $\min\{n, 1/r^d\}$, where n is the number of training examples and d is the dimension of the input space.

§5.4 Decision tree learning

If tree-based structures are so useful for k-nearest neighbour classification, then why not use them for classification directly? This is the idea behind decision tree learning, which is a supervised learning algorithm that is used for classification and regression.

Definition 5.2 (Decision trees)**.** *A* decision tree *is a tree where each internal node represents a feature (or attribute), each branch represents a decision rule, and each leaf node represents the outcome. The topmost node in the tree is the root node, and the tree is grown from the root node. The tree is grown by recursively splitting the data set into subsets based on the value of a feature, and then recursively splitting the subsets on the value of another feature, and so on, until the data set is split into subsets that are all of the same class.*

Definition 5.3. *A* decision tree learning algorithm *is an algorithm that takes as input a data set and outputs a decision tree that is consistent with the data set.*

Definition 5.4 (Random forests)**.** *A* random forest *is an ensemble learning method that constructs a multitude of decision trees at training time and outputs the class that is the mode of the classes of the individual trees. Random forests correct for decision trees' habit of overfitting to their training data.*

Observe right away the advantage that decision trees have over k-d trees; although k-d trees are useful for obtaining nearest neighbours, they are not optimised for classification accuracy, as they are not designed to split the data set into subsets that are all of the same class. Decision trees are useful because rather than selecting an artbitrary feature and splitting at the median, we select the feature and threshold that maximally reduces the label uncertainty—and so we can use decision trees to obtain good-enough classifiers. Here's the general structure for a decision tree:

1. Given a rooted binary tree T
2. A non-leaf node is associated with a predicate involving a single feature.
3. A leaf node is associated with a label from the label set Y.
4. To compute $f_T(x)$, the prediction of tree T at x, we start at the root node, and then recurse on the left child if the predicate at x is true, and recurse on the right child otherwise.

Decision trees are similar to nearest neighbours in that they both try to exploit local regularity, but they differ in that nearest neighbours memorise the training data, while decision trees use the training data to carve the input space into regions, so that for each region there is a good constant prediction.

Now, let's see how to measure the uncertainty of a label. We can use the following measures:

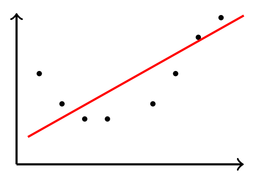
1. Computing the classification error per cell C, which is the fraction of training examples for a particular label that are misclassified:
$$u(C) = 1 - \max_{y \in \mathcal{Y}} \Pr[y \mid \mathbf{x}].$$
2. Computing the Gini impurity per cell C, which is the probability of misclassifying a randomly chosen label:
$$u(C) = 1 - \sum_{y \in \mathcal{Y}} \left(\Pr[y \mid \mathbf{x}]\right)^2 .$$
3. Entropy, which is some measure of the expected value of the information content of a label:
$$u(C) = -\sum_{y \in \mathcal{Y}} \Pr[y \mid \mathbf{x}] \log \Pr[y \mid \mathbf{x}].$$

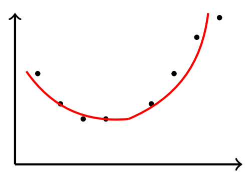
Now we just find the feature F and threshold T that maximally reduces the uncertainty of the label Y:
$$\arg\max_{F,T} \left\{ u(C) - (u(C_L) \cdot p_L + u(C_R) \cdot p_R) \right\},$$
where C_L is the left cell and C_R is the right cell, p_L is the fraction of training examples that go to the left cell, and p_R is the fraction of training examples that go to the right cell, until we reach a stopping criterion, such as a maximum depth, a minimum number of examples per cell, or a minimum uncertainty.

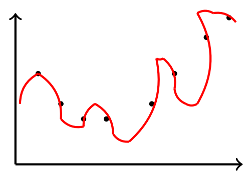
Observations for decision trees Observe that

1. Decision trees are very interpretable, and can be used to understand the importance of different features.
2. Decision trees are constructed in a greedy manner, and so they are not guaranteed to be optimal. In fact, finding the optimal decision tree is NP-hard, and so we have to use heuristics to find a good decision tree in practice.
3. We quickly run out of data as we go down the tree, so uncertainty estimates become less reliable as we go down the tree.
4. Tree complexity is highly dependent on the data geometry in the feature space, and so decision trees can be sensitive to small perturbations in the data.

§5.5 Overfitting and underfitting


underfitting


appropriate fit


overfitting

When we overfit, we end up with a model that is too complex and goes so far as to fit the noise in the data, and so it doesn't generalise well to new data. When we underfit, we end up with a model that is too simple and doesn't capture the complexity of the data, and so it doesn't generalise well to new data. The goal is to find a model that is just right, and this is called the *bias-variance tradeoff*. The bias-variance tradeoff is the tradeoff between the bias of the model and the variance of the model. The bias of the model is the error that is introduced by approximating a real-world problem, which may be extremely complicated, by a much simpler model. The variance of the model is the error that is introduced by the model's sensitivity to small fluctuations in the training data. The bias-variance tradeoff is a fundamental problem in supervised learning, and it is a key reason why we may need to use cross-validation to choose the best model.

§6 Lecture 06—1st February, 2023

Last class, we saw how the nearest neigbour classifier can be accomplished by building a tree or a distribution, approximating it, and so on. Our goal now is to directly figure out the decision boundary, instead of all the other stuff. We will start with linear classification, and then move on to non-linear classification in expanded feature spaces.

§6.1 Linear classification

We will start with the simplest case of linear classification, where we have two classes, and we want to find a line (or really, a hyperplane) that separates the two classes. We will start with the simplest case of two dimensions, and then move on to higher dimensions.

A dataset S from $\mathbb{R}^d \times \{-1, 1\}$ is linearly separable if there exists $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that $y(\mathbf{w}^\top \mathbf{x} + b) > 0$ for all $(\mathbf{x}, y) \in S$, i.e there is a linear decision boundary that perfectly separates the two classes. We can also write this as $y(\mathbf{w}^\top \mathbf{x} + b) \geq 0$ for all $(\mathbf{x}, y) \in S$. We use the output space $Y = \{-1, 1\}$ (instead of $\{0, 1\}$) for notational convenience. The linear classifier determined by this weight vector $\mathbf{w}$ and intercept parameter b is called a linear separator for the dataset S. The decision boundary is the set of points $\mathbf{x}$ such that $\mathbf{w}^\top \mathbf{x} + b = 0$. This is a hyperplane in $\mathbb{R}^d$.

So if we want to learn an affine function (that is, the composition of a linear function and a translation) that separates the two classes, we can define a decision boundary g as

$$g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0 = \underbrace{\begin{bmatrix} \mathbf{w} \\ w_0 \end{bmatrix}}_{\mathbf{w}'} \cdot \underbrace{\begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}}_{\mathbf{x}'} = \mathbf{w}' \cdot \mathbf{x}',$$

where $\mathbf{w}$ is the weight vector and w_0 is the bias. We can then classify a point $\mathbf{x}$ via the linear classifier f:

$$\begin{aligned} f(\mathbf{x}) &= \begin{cases} 1 & \text{if } g(\mathbf{x}) \geq 0 \\ -1 & \text{otherwise} \end{cases} \\ &= \operatorname{sign}(\mathbf{w} \cdot \mathbf{x} + w_0). \end{aligned}$$

Note that we need to learn $d+1$ parameters: w_0 and the d weights in $\mathbf{w}$. We can also write this as

$$f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^\top \mathbf{x} + w_0).$$

§6.2 Logistic regression

A logistic regression model is a linear classifier that uses the logistic function to model the probability that a given input $\mathbf{x}$ belongs to a particular binary class. The logistic function is defined as

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

One nice property of this function is that $1 - \sigma(t) = \sigma(-t)$, and this motivates the use of the log odds ratio (or logit) as the linear function. The logit is defined as

$$\ln\left(\frac{\Pr(Y=1 \mid X=\mathbf{x})}{\Pr(Y=0 \mid X=\mathbf{x})}\right) = \mathbf{w}^\top \mathbf{x} + w_0,$$

which is a linear function in $\mathbf{x}$. Given $X = \mathbf{x}$ for instance, a label $Y = 1$ is more likely than $Y = 0$ if and only if $\mathbf{w}^\top \mathbf{x} + w_0 > 0$. So the classifier with the smallest error rate under this distribution is

$$f^*(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^\top\mathbf{x} + w_0 > 0 \\ 0 & \text{otherwise} \end{cases} = \mathbb{1}\{\mathbf{w}^\top\mathbf{x} + w_0 > 0\}.$$

Such a classifier is calle da linear classifier.

§6.2.1 Maximum likelihood estimation from logistic regression

We can estimate the parameters $\mathbf{w}$ and w_0 by maximum likelihood estimation. The likelihood of the parameters given the data is

$$\begin{aligned} L(\mathbf{w}) &= \prod_{(\mathbf{x},y)\in S} \begin{cases} \sigma(\mathbf{w}^\top\mathbf{x} + w_0) & \text{if } y = 1 \\ 1 - \sigma(\mathbf{w}^\top\mathbf{x} + w_0) & \text{if } y = 0 \end{cases} \\ &= \prod_{(\mathbf{x},y)\in S} \frac{1}{1 + e^{-y(\mathbf{w}^\top\mathbf{x}+w_0)}}. \end{aligned}$$

In this case the log likelihood is

$$\ln L(\mathbf{w}) = \sum_{(\mathbf{x},y)\in S} \ln\left(\frac{1}{1+e^{-y(\mathbf{w}^\top\mathbf{x}+w_0)}}\right) = \sum_{(\mathbf{x},y)\in S}\left(-\ln(1+e^{-y(\mathbf{w}^\top\mathbf{x}+w_0)})\right).$$

The maximiser in this case is not characterised by a system of linear equations, but can be approximated by iterative methods, such as gradient descent:

```
Algorithm: ITERATIVE IMPROVEMENT FOR LOGISTIC REGRESSION

• Start with $\mathbf{w}^{(0)} \leftarrow \mathbf{0}$.
• Repeat $T$ times, until convergence:
    – For each $(\mathbf{x}, y) \in S$, update $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \eta y \mathbf{x} \sigma(-y\mathbf{w}^{(t-1)\top}\mathbf{x})$, where $\eta$ is the learning rate.
• Output $\mathbf{w}^{(T)}$.
```

Logarithmic loss The negative log-likelihood objective in logistic regression may be written as

$$-\ln L(\mathbf{w}) = \frac{1}{|S|} \sum_{(\mathbf{x},y)\in S} y \ln\left(\frac{1}{\hat{p}_{\mathbf{w}}(\mathbf{x})}\right) + (1-y)\ln\left(\frac{1}{1-\hat{p}_{\mathbf{w}}(\mathbf{x})}\right),$$

where $\hat{p}_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + w_0)$. This objective is proportional to the empirical risk of the logistic regression model, and is known as the *logarithmic loss* or *cross-entropy loss*.

§6.2.2 Approximate maximum likelihood estimation from logistic regression

How can we find a linear separator for a linearly separable dataset S? One approach is to find an approximate maximiser of the log-likelihood from the logistic regression model. Any algorithm that can find $(\mathbf{w}, w_0)$ with log-likelihood arbitrarily close to the maximum log-likelihood is sufficient. We have already seen that the log-likelihood of $(\mathbf{w}, w_0)$ given S in the logistic regression model is

$$\ln L(\mathbf{w}) = \sum_{(\mathbf{x},y)\in S} \ln\left(\frac{1}{1+e^{-y(\mathbf{w}^\top \mathbf{x}+w_0)}}\right).$$

Notice that, in each term from the summation, the argument to the logarithm is strictly between 0 and 1, and hence the logarithm is strictly negative. Thus, the log-likelihood is strictly negative, regardless of the choice of $\mathbf{w}$ and w_0. However, if S is linearly separable, then it s possible to achieve log-likelihood arbitrarily close to 0. Suppose $(\mathbf{w}, w_0)$ determines a linear separator for S. Then for any $c > 0$, $(c\mathbf{w}, cw_0)$ also determines a linear separator for S. This is because $y(c\mathbf{w}^\top \mathbf{x} + cw_0) = cy(\mathbf{w}^\top \mathbf{x} + w_0) > 0$ for all $(\mathbf{x}, y) \in S$. Moreover, by choosing a sufficiently large c, we can make the positive number $y(c\mathbf{w}^\top \mathbf{x} + cw_0)$ arbitrarily large for all $(\mathbf{x}, y) \in S$, which in turn makes

$$\frac{1}{1+e^{-y(c\mathbf{w}^\top \mathbf{x}+cw_0)}}$$

arbitrarily close to 1. Therefore each term in the log-likelihood of $(c\mathbf{w}, cw_0)$ given S is arbitrarily close to 0, and hence the log-likelihood is arbitrarily close to 0. This means that

$$\max_{(\mathbf{w},w_0)\in\mathbb{R}^d\times\mathbb{R}} \ln L(\mathbf{w}) = 0,$$

that is, the maximum log-likelihood is 0. It remains then to show that any $(\mathbf{w}, w_0)$ that achieves log-likelihood sufficiently close to the maximum log-likelihood (i.e., close to 0) also determines a linear separator for S. Suppose $\ln L(\mathbf{w}) \geq -\ln 2$. Then we can readily show that

$$\frac{1}{1+e^{-y(\mathbf{w}^\top \mathbf{x}+w_0)}} \geq \frac{1}{2},$$

which is the same as saying that $y(\mathbf{w}^\top \mathbf{x} + w_0) \geq 0$. Therefore, any $(\mathbf{w}, w_0)$ that achieves log-likelihood sufficiently close to the maximum log-likelihood also determines a linear separator for S. This means that we can use the logistic regression model to find a linear separator for a linearly separable dataset S.

§6.3 Perceptron

Now recall our discussion from the section on linear separability. How do we learn the weights? Consider the following problem:

Problem 6.1. *Given a training data set S from $\mathbb{R}^d \times \{0,1\}$, find the $\mathbf{w} \in \mathbb{R}^d$ that minimises the training error rate; that is, for $f_{\mathbf{w}}(\mathbf{x}) = \mathbb{1}(\mathbf{w}^\top \mathbf{x} \geq 0)$, we want to find*

$$\begin{aligned}\arg\min_{\mathbf{w}} \frac{1}{|S|} \sum_{(x,y)\in S} \mathrm{loss}_{0/1}(f_{\mathbf{w}}(x), y) &= \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{\mathrm{sign}(\mathbf{w}\cdot\mathbf{x}_i) \neq y_i\} \\ &= \arg\min_{\mathbf{w}} \frac{1}{n} \sum_{\substack{x_i \\ s.t.\ y_i-+1}} \mathbb{1}\{\mathbf{x}_i \cdot \mathbf{w} < 0\} + \sum_{\substack{x_i \\ s.t.\ y_i=-1}} \mathbb{1}\{\mathbf{x}_i \cdot \mathbf{w} \geq 0\}.\end{aligned}$$

But how then do we solve the minimisation problem? We cannot use the standard technique of taking the gradient and examining the critical points, because the loss function is not differentiable. Indeed this problem is computationally intractable in general (assuming $\mathsf{P} \neq \mathsf{NP}$). However, we can relax the problem to one that is more amenable to our desires by specifying that the training data set is linearly separable:

Problem 6.2 (Linear separability)**.** *Given a training data set S from $\mathbb{R}^d \times \{0,1\}$, is there a $\mathbf{w} \in \mathbb{R}^d$ such that the training error rate of $f_{\mathbf{w}}$ is zero? That is,*

$$\exists?\ w \in \mathbb{R}^d \quad \textit{such that} \quad \frac{1}{|S|} \sum_{(x,y)\in S} \mathrm{loss}_{0/1}(f_{\mathbf{w}}(x), y) = 0.$$

If there is, find such a $\mathbf{w}$.

This is also known as the linear feasibility problem. We can solve this problem using the perceptron algorithm.

Algorithm: Perceptron

- Start with $\mathbf{w}^{(0)} \leftarrow \mathbf{0}$.
- For $t = 1, 2, \ldots, T$:
  - If there exists $(\mathbf{x}, y) \in S$ such that $\mathrm{sign}\left(\mathbf{w}^{(t-1)} \cdot \mathbf{x}\right) \neq y$, then update

$$\mathbf{w}^{(t)} \leftarrow \begin{cases} \mathbf{w}^{(t-1)} + \mathbf{x} & \text{if } y = 1 \\ \mathbf{w}^{(t-1)} - \mathbf{x} & \text{if } y = -1 \end{cases} = \mathbf{w}^{(t-1)} + y\mathbf{x}.$$

- Output $\mathbf{w}^{(T)}$.

We have the following mistake bound for the perceptron algorithm:

Theorem 6.1 (Perceptron mistake bound)**.** *Suppose there is a $\mathbf{w}^* \in \mathbb{R}^d$ such that $\|\mathbf{w}^*\| \leq R$, where $R = \max_{(\mathbf{x},y)\in S} \|\mathbf{x}\|$, and the training data set S is linearly separable with margin γ. Then the* *perceptron algorithm with $T = \frac{R^2}{\gamma^2}$ mistakes/iterations (in the worst case, over all possible orderings of the training data set S) outputs a $\mathbf{w}^{(T)}$ such that $\|\mathbf{w}^{(T)}\| \leq R$ and*

$$\frac{1}{|S|} \sum_{(x,y)\in S} \mathbb{1}\{\text{sign}(\mathbf{w}^{(T)} \cdot \mathbf{x}) \neq y\} = 0.$$

Right away, we remark that the number of data points n does not appear in the mistake bound. This is a good thing, because it means that the perceptron algorithm is not sensitive to the size of the training data set. Also, we can assume without loss of generality that $R = 1$, because we can always scale the data set to make this so. In that case the mistake bound becomes $T = \frac{1}{\gamma^2}$.

The existence of $\mathbf{w}^*$ is a strong assumption, and it is not always the case that the training data set is linearly separable. However, we can always find a $\mathbf{w}$ such that the training error rate is small, and this notion is closely related to the notion of margin γ.

Definition 6.3 (Margin)**.** *The margin of a linear classifier $f_\mathbf{w}$ for a training data set S is*

$$\gamma = \min_{(\mathbf{x},y)\in S} y(\mathbf{w}^\top \mathbf{x}).$$

The margin is a measure of how well the linear classifier separates the two classes. The larger the margin, the better the separation. The margin is also a measure of the robustness of the linear classifier to perturbations in the data.

Proof of Theorem 6.1. The key quantity to consider is the angle (or distance) between the gold standard $\mathbf{w}^*$ and the current iterant $\mathbf{w}^{(t)}$. Suppose that on the tth iteration, the perceptron algorithm makes a mistake on the point $(\mathbf{x}, y)$. Then we have

$$\mathbf{w}^{(t)} \cdot \mathbf{w}^* = \mathbf{w}^{(t-1)} \cdot \mathbf{w}^* + y\mathbf{w}^* \cdot \mathbf{x} = \left(\mathbf{w}^{(t-1)} + y\mathbf{x}\right) \cdot \mathbf{w}^* \geq \mathbf{w}^{(t-1)} \cdot \mathbf{w}^* + \gamma,$$

since $\mathbf{w}^* \cdot \mathbf{x} \geq \gamma$. Now we want the length of $\mathbf{w}^{(t)}$ to not grow too much, so we want to bound $\|\mathbf{w}^{(t)}\|^2$. We have

$$\|\mathbf{w}^{(t)}\|^2 = \|\mathbf{w}^{(t-1)} + y\mathbf{x}\|^2 = \|\mathbf{w}^{(t-1)}\|^2 + 2y\mathbf{w}^{(t-1)} \cdot \mathbf{x} + \|y\mathbf{x}\|^2 \leq \|\mathbf{w}^{(t-1)}\|^2 + R^2,$$

since $\|y\mathbf{x}\| \leq R$ (remember $y \in \{-1, 1\}$). Thus we have that for all iterations t in which the perceptron algorithm makes a mistake, we have

$$\begin{aligned}\mathbf{w}^{(t)} \cdot \mathbf{w}^* &\geq \mathbf{w}^{(t-1)} \cdot \mathbf{w}^* + \gamma \\ \|\mathbf{w}^{(t)}\|^2 &\leq \|\mathbf{w}^{(t-1)}\|^2 + R^2.\end{aligned}$$

So after T rounds,

$$T\gamma \leq \mathbf{w}^{(T)} \cdot \mathbf{w}^* \leq \|\mathbf{w}^{(T)}\|\|\mathbf{w}^*\| \leq R\sqrt{T},$$

where the last inequality follows from the Cauchy-Schwarz inequality and by summing the inequalities over all $t = 1, 2, \ldots, T$ since $\|\mathbf{w}^{(0)}\| = 0$ and

$$\|\mathbf{w}^{(T)}\|^2 \leq TR^2,$$

by summing the inequalities over all $t = 1, 2, \ldots, T$. Thus we have

$$\gamma^2 \leq \frac{R^2}{T},$$

and the result follows. $\square$

Note that the perceptron algorithm examines the sequence in arbitrary order in an online fashion—there is no insistence on an i.i.d. type sampling procedure or something similar. This is a good thing, because it means that the perceptron mistake bound presents an upper bound on the mistakes made by itself (an online algorithm) in the worst case, over all possible orderings of the training data set S. This is particularly useful in the case of streaming data, where we typically resort to online algorithms of this sort.

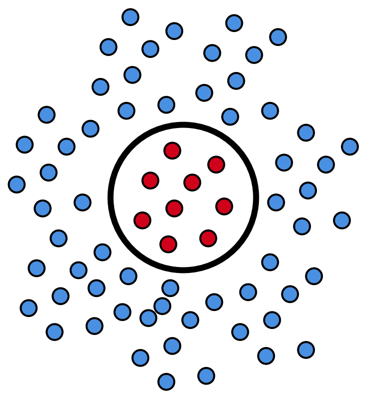
Definition 6.4 (Online algorithm)**.** *An algorithm is said to be* online *if it examines the sequence in arbitrary order.*

Online algorithms with small mistake bonds—i.e. small values for the number of mistakes made by an online algorithm on an arbitrary sequence of examples—can be used to develop classifiers with good generalisation properties, which we may see more about later.

§7 Lecture 07—6th February, 2023

§7.1 Generalising linear classification: feature transformations and kernels

Suppose we are trying to do classification, but the data is not linearly separable:



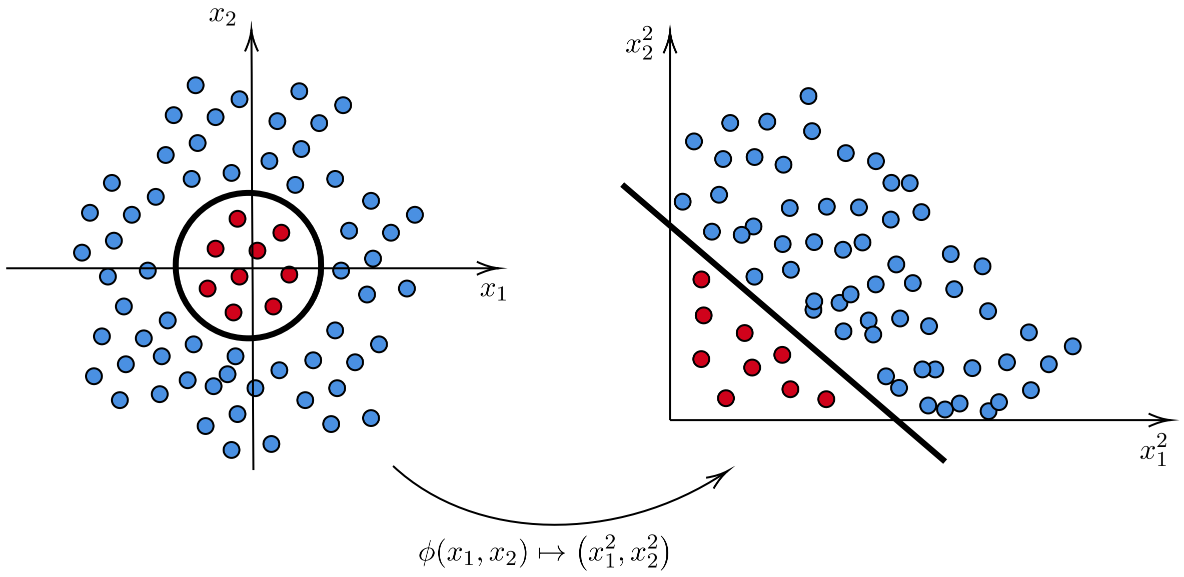
This data set is not linearly separable, however there exists a circular boundary that can adequately separate the data. In the linear case, we parameterised the (affine) form of the decision boundary as $w^\top x + b = 0$. So maybe we can do something similar here. We can use a non-linear transformation (perhaps a "circular" transformation) of the input space, perhaps for $d = 2$:

$$g(\mathbf{x}) = w_1 x_1^2 + w_2 x_2^2 + w_0,$$

which for $w_1 = w_2 = 1$ and $w_0 = -r^2$ gives us the equation of a circle of radius r centred at the origin. However, this classifier does not accommodate every single quadratic boundary. Fortunately, g is linear in some higher-dimensional space, so we can still use the same linear classification techniques if we transform the data appropriately:

$$\begin{aligned} g(\mathbf{x}) &= w_1 x_1^2 + w_2 x_2^2 + w_0 \\ &= w_1 \chi_1 + w_2 \chi_2 + w_0, \end{aligned}$$

And so if we apply the feature transformation $\phi(x_1, x_2) \mapsto (x_1^2, x_2^2)$, g is linear in the new ϕ-transformed space. This is the idea behind the kernel trick.



So for the generic quadratic boundary (i.e. the $\mathbb{R}^2 \to \mathbb{R}$ case), we have

$$g(\mathbf{x}) = w_1 x_1^2 + w_2 x_2^2 + w_3 x_1 x_2 + w_4 x_1 + w_5 x_2 + w_0 = \sum_{\substack{i,j \in \mathbb{N} \\ i+j \leq 2}} w_{i,j} x_1^i x_2^j + w_0$$

where $\mathbf{x} = (x_1, x_2)$, and the feature transformation is $\phi(\mathbf{x}) = (x_1^2, x_2^2, x_1 x_2, x_1, x_2, 1)$. In general, for the d-dimensional quadratic boundary, we have

$$g(\mathbf{x}) = \sum_{i,j=1}^{d} \sum_{p,q \in \mathbb{N}, p+q \leq 2} w_{i,j,p,q} x_i^p x_j^q,$$

which captures all pairwise interactions between the d features. The feature transformation (which we will also call the kernel transformation) is

$$\phi(\mathbf{x}) = (x_1^2, x_2^2, \ldots, x_d^2, x_1 x_2, x_1 x_3, \ldots, x_{d-1} x_d, x_1, x_2, \ldots, x_d, 1),$$

which is a mapping from $\mathbb{R}^d \to \mathbb{R}^{d^2+d+1}$.

The moral lesson here is that the data is not always linearly separable in the original space, but it is linearly separable in *some* sufficiently-high-dimensional space:

Theorem 7.1. *Given n distinct points $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, there exists a feature transform such that any labelling of S is linearly separable in the transformed space.*

Proof. Given n points, consider the following mapping into $\mathbb{R}^n$:

$$\phi(\mathbf{x}_i) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

where the 1 is in the ith position. Then the decision boundary induced by the linear weighting $w = \begin{bmatrix} y_1 & \cdots & y_n \end{bmatrix}^\top$ perfectly separates the input data. □

The kernel trick Explicitly working in a generic kernel space $\phi(\mathbf{x}_i)$ takes time $\Omega(n^2 d)$, which is not feasible for large n or d. However, the inner product $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$ between two data points in the kernel space can be computed relatively quickly.

Example 7.1. 1. For the quadratic kernel, we have the:

- *explicit transform*

$$\phi\colon \mathbf{x} \to \left(x_1^2, \ldots, x_d^2, \sqrt{2}x_1x_2, \ldots, \sqrt{2}x_{d-1}x_d, \sqrt{2}x_1, \ldots, \sqrt{2}x_d, 1\right),$$

which takes time $O(d^2)$ (here the $\sqrt{2}$ is for normalisation), and the

- *kernel trick via inner product*

$$\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = \left(1 + \mathbf{x}_i^\top \mathbf{x}_j\right)^2,$$

which takes time $O(d)$.

2. For the radial basis function (RBF) kernel, we have the:

- *explicit transform*

$$\mathbf{x} \mapsto \left(\left(\frac{2}{\pi}\right)^{d/4} \cdot \exp\left(-\|\mathbf{x} - \alpha\|^2\right)\right)_{\alpha \in \mathbb{R}^d},$$

which takes time corresponding to infinite dimensions, and the

- *kernel trick via inner product*

$$\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = \exp\left(-\|\mathbf{x}_i - \mathbf{x}_j\|^2\right),$$

which takes time $O(d)$.

The spirit of the kernel trick is to perform classification in such a way that it only accesses the data in terms of dot products (so that it can be done quicker).

Example 7.2 (Kernel perceptron)**.** Recall the update rule for the perceptron algorithm:

$$w^{(t)} \leftarrow w^{(t-1)} + y\mathbf{x} \text{ if } y \cdot w^{(t-1)} \cdot \mathbf{x} \leq 0.$$

Equivalently, we can write this as

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i,$$

where α_i is the number of times $\mathbf{x}_i$ was misclassified. The kernel trick allows us to do classification in the kernel space, that is,

$$f(\mathbf{x}) := \text{sign}(\mathbf{w} \cdot \mathbf{x}) = \text{sign}\left(\mathbf{x} \cdot \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\right) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x})\right),$$

and so we only need to access the data in terms of dot products.

The example above suggests that for classification in the original space, we should have

$$f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x})\right),$$

where α_i is the number of times $\mathbf{x}_i$ was misclassified. If we were working in the transformed kernel space, it would have been

$$f(\phi(\mathbf{x})) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i (\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}))\right).$$

So we can present the kernel perceptron algorithm as follows:

Algorithm: Kernelised perceptron

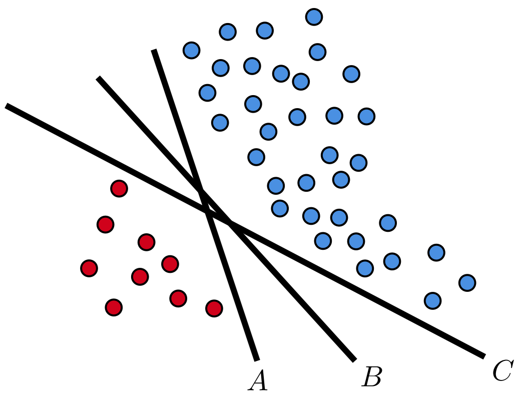
```
• Initialise $(\alpha_1, \ldots, \alpha_n) \leftarrow (0, \ldots, 0)$.
• For $t = 1, 2, \ldots, T$:
    – If there exists $(\mathbf{x}_i, y_i) \in S$ such that $\text{sign}\left(\sum_{k=1}^{n} \alpha_k y_k (\phi(\mathbf{x}_k) \cdot \phi(\mathbf{x}_i))\right) \neq y_i$, then
      update $\alpha_i \leftarrow \alpha_i + 1$.
• Output the final classifier $f(\mathbf{x}) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i (\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_T))\right)$.
```

Notice that we are not explicitly computing the feature transformation $\phi(\mathbf{x})$, but we are implicitly working in the kernel space. The kernel trick allows us to work in the kernel space without explicitly computing the feature transformation, with a view towards reducing the computational complexity of the algorithm and making it more efficient.

§8 Lecture 08—8th February, 2023

§8.1 Support vector machines

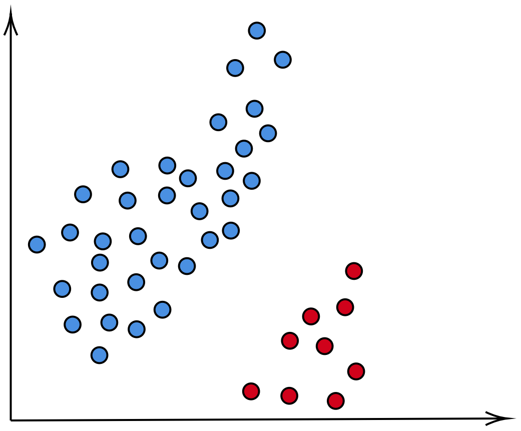
Motivation for SVMs Suppose that there is a linear decision boundary which can perfectly separate the training data. Which of the two linear separators below would we want the Perceptron algorithm to return?



Clearly the separator with the largest margin γ is better for generalisation, because it is less likely to overfit the training data, and we make fewer mistakes on the test data as a result. (Recall that the margin is the distance from the separator to the closest point in the training data.) Even then, however, perceptron cannot handle erroneous data points, and it is not guaranteed to converge to a solution if the data is not explicitly linearly separable.

The support vector machine helps us achieve a large margin stable hyperplane; it returns a linear classifier that is a stable solution by giving a maximum margin solution to the problem. Furthermore, a slight modification to the problem provides a way to deal with nonseparabl cases. In particular, support vector machines are kernelisable, so they give an implicit way of yielding non-linear classifiers.

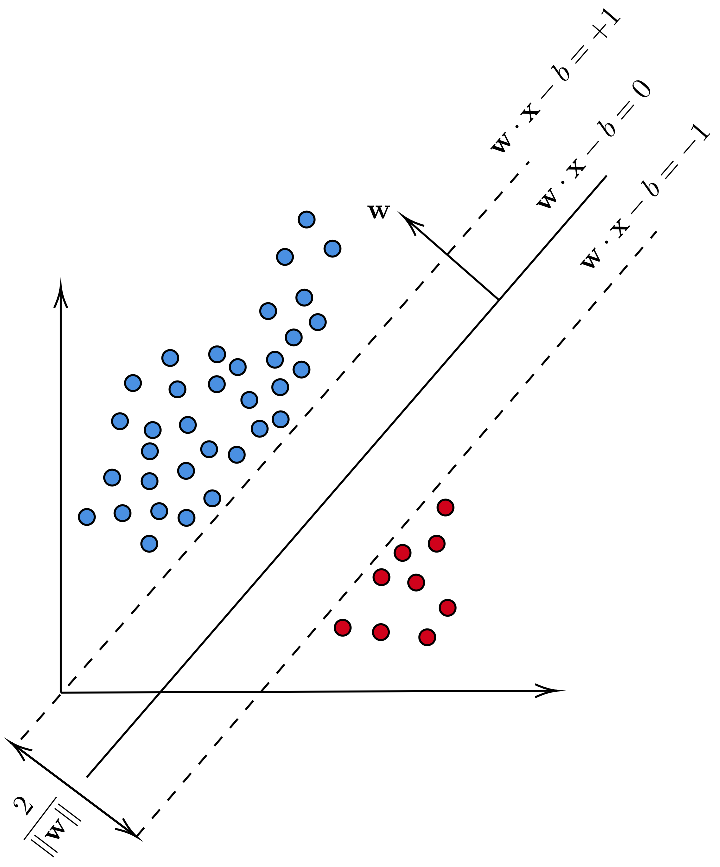
SVM formulation Suppose that the training data $S = (\mathbf{x}, y)$ is linearly separable by some margin (but the linear separator does not necessarily pass through the origin).



The decision boundary is $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b = 0$—where b is the bias and $\mathbf{w}$ is the weight vector—and the linear classifier is

$$f(\mathbf{x}) = \operatorname{sign}(g(\mathbf{x})) = \operatorname{sign}(\mathbf{w} \cdot \mathbf{x} - b).$$

Instead of learning one hyperplane, we can try to learn two *parallel* hyperplanes that classify *all* of the points, and maximise the distance between them!



$\mathbf{w} \cdot \mathbf{x} - b = +1$
$\mathbf{w} \cdot \mathbf{x} - b = 0$
$\mathbf{w} \cdot \mathbf{x} - b = -1$
$\mathbf{w}$
$\frac{2}{\|\mathbf{w}\|}$

Here the decision boundaries for the two parallel hyperplanes are $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b = +1$ and $g(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - b = -1$. Note that the distance between these two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, because the closest distance of the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ to the origin is $\frac{b}{\|\mathbf{w}\|}$, and so the distance between the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 1$ and the origin is $\frac{1-b}{\|\mathbf{w}\|}$, and the distance between the hyperplane $\mathbf{w} \cdot \mathbf{x} + b = -1$ and the origin is $\frac{-1-b}{\|\mathbf{w}\|}$, thus the sum of these two distances is $\frac{2}{\|\mathbf{w}\|}$.

Now in the diagram above, the training data is correctly classified if

$$\begin{aligned} &\mathbf{w} \cdot \mathbf{x}_i - b \geq 1 && \text{for } y_i = +1, \\ &\mathbf{w} \cdot \mathbf{x}_i - b \leq -1 && \text{for } y_i = -1. \end{aligned}$$

Altogether, this is just saying that $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq +1$ for all i. Now given this and the fact that the distance between the two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, our goal for the SVM is to maximize the margin $\frac{2}{\|\mathbf{w}\|}$, such that the training data is correctly classified:

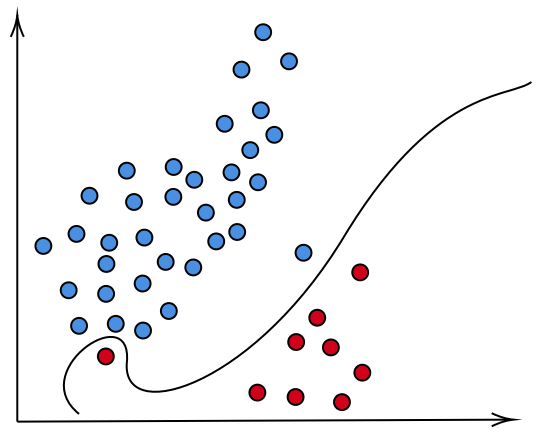
$$\begin{aligned} \text{maximise} \quad & \frac{2}{\|\mathbf{w}\|} \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i - b) \geq 1, \quad \text{for all } i \end{aligned}$$

The more typical optimisation problem for the SVM, the so-called *primal problem*, is

$$\begin{aligned} \text{minimise} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i - b) \geq 1, \quad \text{for all } i \end{aligned}$$

§8.2 SVMs with non-linear decision boundaries

All of the discussion we have given so far assumes that the decision boundary is linear. However, we often need work with data sets that are not linearly separable, such as this one:

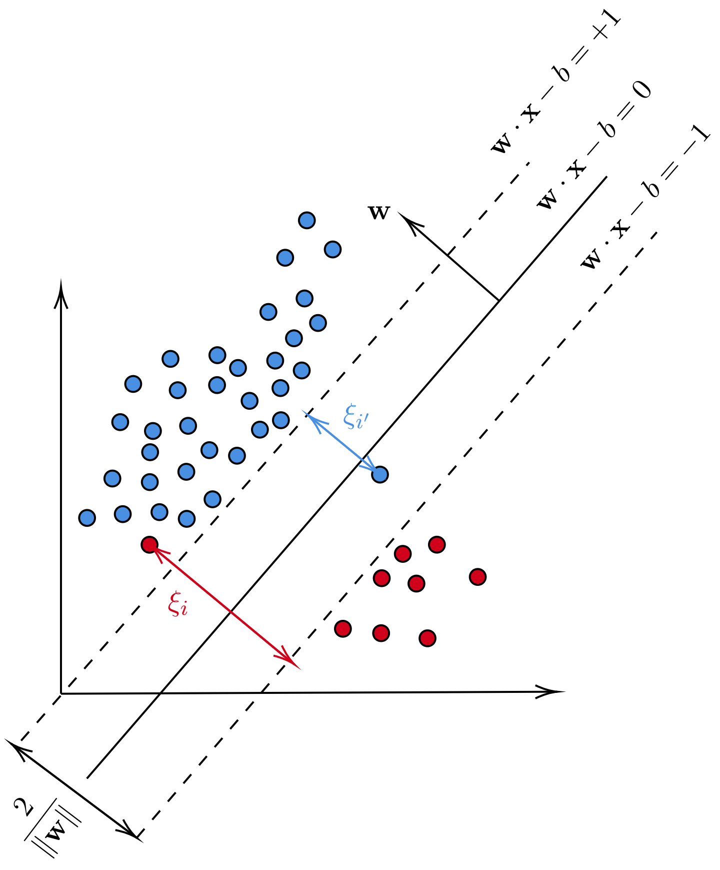


This is much too complex, and may be trading off too much complexity for too little error. We can try to simplify the model by essentially allowing some "slack", which makes us more or less "give up" on some of the data points, but not on too many. This is the idea behind the *soft margin SVM*.

Following from the discussion above, we introduce slack variables $\xi_i \geq 0$ for each data point $(\mathbf{x}_i, y_i)$, and we will seek to minimise the slack as well as the margin. We can write the optimisation problem for the soft margin SVM as

$$\begin{gathered} \text{minimise } \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i \\ \text{subject to } y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \xi_i \text{ for all } i = 1, \ldots, n \\ \xi_i \geq 0 \text{ for all } i = 1, \ldots, n. \end{gathered}$$

Here $C > 0$ is a hyperparameter that controls the tradeoff between the margin and the slack. If C is very small, then we are essentially back to the hard margin SVM, and if C is very large, then we are allowing a lot of slack. The optimisation problem above is a convex quadratic program, and can be solved efficiently via the *Lagrangian dual* method, which we will introduce soon.


$\mathbf{w}\cdot\mathbf{x}-b=+1$
$\mathbf{w}\cdot\mathbf{x}-b=0$
$\mathbf{w}\cdot\mathbf{x}-b=-1$
$\mathbf{w}$
$\xi_{i'}$
ξ_i
$\frac{2}{\|\mathbf{w}\|}$

Note that we cannot simply take the derivative of the objective function and set it to zero to find the optimal $\mathbf{w}, \xi$ and b, because the objective function is not differentiable at the points where $\mathbf{w} \cdot \mathbf{x}_i - b = \pm 1$; we are optimising a somewhat piecewise function.

§8.3 Detour: constrained optimisation

The typical problems we encounter in this subdomain of optimisation are of the form

$$\begin{aligned} \text{minimise}_{\mathbf{x}\in\mathbb{R}^d} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_i(\mathbf{x}) \le 0, \quad i = 1, \dots, n. \end{aligned}$$

We will always assume that the problems we discuss are feasible. There are at least two generic methods for solving problems of this sort:

- *Projection methods.* Here we: (1) start with a feasible solution x_0; (2) find the x_1 that has a slightly lower objective value, and in the case that x_1 has violated constraints, project it back onto the feasible set; (3) repeat this process until convergence.
- *Penalty (or barrier) methods.* Here we: (1) start with a feasible solution x_0; (2) find the x_1 that has a slightly lower objective value, and in the case that x_1 has violated constraints, penalise the objective function; (3) repeat this process until convergence.

The Lagrange penalty method Consider the augumented function

$$L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1}^{n} \lambda_i g_i(\mathbf{x}).$$

This function $L : \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}$ is called the Lagrange function. The following observations are a review of what we have seen in previous courses:

- For any feasible $\mathbf{x}$ and for all $\lambda_i \geq 0$, we have $L(\mathbf{x}, \boldsymbol{\lambda}) \leq f(\mathbf{x})$, and as a consequence $\max_{\lambda_i \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}) \leq f(\mathbf{x})$.
  If $\mathbf{x}$ is feasible, then $g_i(\mathbf{x}) \leq 0$ for all i, and so $\lambda_i g_i(\mathbf{x}) \leq 0$ for all i. Together with $\lambda_i \geq 0$, we have $f(\mathbf{x}) \geq f(\mathbf{x}) + \sum_{i=1}^{n} \lambda_i g_i(\mathbf{x}) = L(\mathbf{x}, \boldsymbol{\lambda})$.
- If $\mathbf{x}$ is infeasible, then $L(\mathbf{x}, \boldsymbol{\lambda}) = \infty$ for all $\boldsymbol{\lambda}$.
  If $\mathbf{x}$ is infeasible, then there exists an i such that $g_i(\mathbf{x}) > 0$. Since $\lambda_i \geq 0$, we have $\lambda_i g_i(\mathbf{x}) > 0$, and so $L(\mathbf{x}, \boldsymbol{\lambda}) = \infty$.
- If $\mathbf{x}$ is feasible, then $\max_{\lambda_i \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x})$.
  If $\mathbf{x}$ is feasible, then $g_i(\mathbf{x}) \leq 0$ for all i, and so $\max_{\lambda_i \geq 0} \lambda_i g_i(\mathbf{x}) = 0$. Therefore, the maximum $\max_{\lambda_i \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x})$.

From all of these, the solution to the original (primal) constrained optimisation problem is

$$p^* := \min_{\mathbf{x} \in \mathbb{R}^d} \max_{\boldsymbol{\lambda} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}),$$

and this problem is now a simple unconstrained optimisation problem in $\mathbf{x}$ and $\boldsymbol{\lambda}$, which can be solved using gradient descent or other methods.

Now let x^* be the minimum feasible solution (over the domain of f) to the primal problem. Then for all $\lambda_i \geq 0$, we have

$$\min_{\mathbf{x} \in \mathbb{R}^d} L(\mathbf{x}, \boldsymbol{\lambda}) \leq L(x^*, \boldsymbol{\lambda}) \leq f(x^*) = p^*.$$

Therefore,

$$\underbrace{\max_{\boldsymbol{\lambda} \geq 0} \min_{\mathbf{x} \in \mathbb{R}^d} L(\mathbf{x}, \boldsymbol{\lambda})}_{:=d^*} \leq p^*.$$

We call the problem $\max_{\boldsymbol{\lambda} \geq 0} \min_{\mathbf{x} \in \mathbb{R}^d} L(\mathbf{x}, \boldsymbol{\lambda})$ the dual problem, and the solution d^* the dual optimum. The difference $p^* - d^*$ is called the duality gap.

We have established one of the duality theorems:

Theorem 8.1 (Weak Lagrange duality)**.** *The duality gap $p^* - d^*$ is nonnegative, i.e.*

$$\min_{\mathbf{x} \in \mathbb{R}^d} \max_{\boldsymbol{\lambda} \geq 0} L(\mathbf{x}, \boldsymbol{\lambda}) \geq \max_{\boldsymbol{\lambda} \geq 0} \min_{\mathbf{x} \in \mathbb{R}^d} L(\mathbf{x}, \boldsymbol{\lambda}).$$

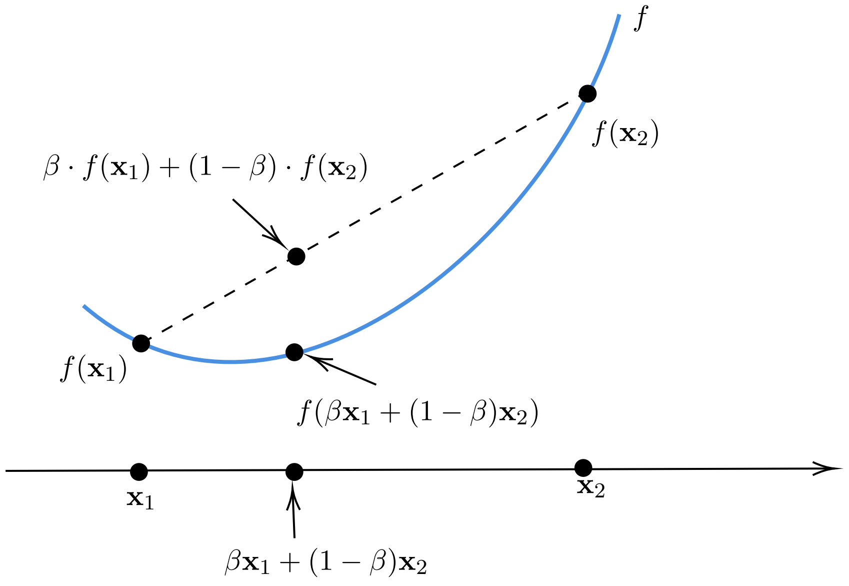
We will discuss the case where the duality gap is zero in the next lecture.

§9 Lecture 09—13th February, 2023

§9.1 Convexity and convex optimisation

A function $f\colon \mathbb{R}^d \to \mathbb{R}$ is called *convex* if for all $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^d$ and $\beta \in [0,1]$, we have

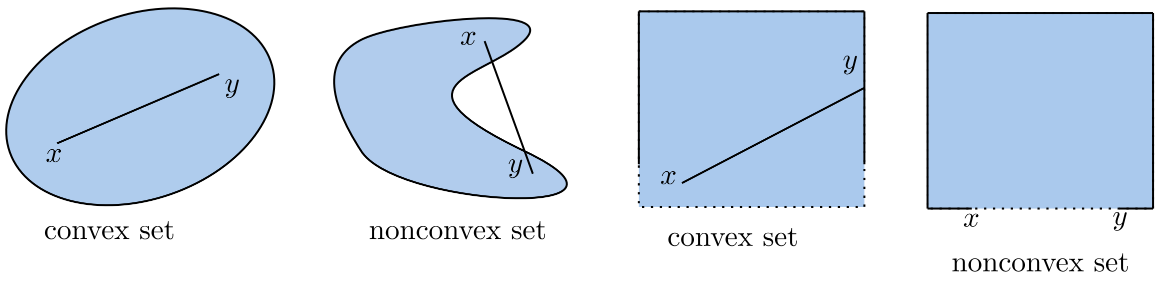
$$f(\beta \mathbf{x}_1 + (1-\beta)\mathbf{x}_2) \leq \beta f(\mathbf{x}_1) + (1-\beta) f(\mathbf{x}_2).$$



f
$f(\mathbf{x}_2)$
$\beta \cdot f(\mathbf{x}_1) + (1-\beta) \cdot f(\mathbf{x}_2)$
$f(\mathbf{x}_1)$
$f(\beta \mathbf{x}_1 + (1-\beta)\mathbf{x}_2)$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\beta \mathbf{x}_1 + (1-\beta)\mathbf{x}_2$

A set $S \subseteq \mathbb{R}^d$ is called *convex* if for all $\mathbf{x}_1, \mathbf{x}_2 \in S$ and $\beta \in [0,1]$, we have

$$\beta \mathbf{x}_1 + (1-\beta)\mathbf{x}_2 \in S.$$



A constrained optimisation problem

$$\begin{array}{rl} \text{minimise} & f(\mathbf{x}) \\ \text{subject to} & g_i(\mathbf{x}) \leq 0, \quad i = 1, \ldots, n. \end{array}$$

is called a convex optimisation problem if:

- the objective function f is a convex function,
- the feasible set $\{\mathbf{x} \in \mathbb{R}^d \mid g_i(\mathbf{x}) \leq 0, i = 1, \ldots, n\}$ induced by the constraints g_i is a convex set.

We will often care about convex optimisation problems because they are easier to solve than non-convex optimisation problems. In particular, for convex optimisation problems, we can often find the global minimum efficiently. There are also the following niceties for convex optimisation:

- Every local optimum is a global optimum in a convex optimisation problem. Some examples of convex optimisation problems include linear programming, quadratic programming, and support vector machines. Several solvers are available for convex optimisation problems, including CVXOPT, MOSEK, and SeDuMi.
- We can use a simple descent algorithm called *gradient descent* to solve convex optimisation problems.

For convex optimisation problems, the duality gap is zero:

Theorem 9.1 (Strong Lagrangian duality)**.** *For a convex optimisation problem, if there exists a feasible point x such that $g_i(\mathbf{x}) < 0$ for all i or $g_i(\mathbf{x}) \leq 0$ whenever g_i is affine (i.e. Slater's condition holds), then the primal and dual problems have the same optimal value, i.e. $p^* = d^*$.*

Gradient descent for finding local minima We will now briefly discuss gradient descent in this specific context of convex optimisation. The idea is to iteratively update the current point $\mathbf{x}_t$ by moving in the direction of the negative gradient of the objective function f at $\mathbf{x}_t$:

Theorem 9.2. *Given a smooth convex function $f\colon \mathbb{R}^d \to \mathbb{R}$, for any $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{x}' := \mathbf{x} - \eta\nabla_{\mathbf{x}} f(\mathbf{x})$, for sufficiently small $\eta > 0$, we have $f(\mathbf{x}') \leq f(\mathbf{x})$.*

We can derive this result by considering the Taylor expansion of f around $\mathbf{x}$. But more importantly for us right now, we present the projected gradient descent algorithm for constrained optimisation problems:

Algorithm: Projected gradient descent

```
• Initialise $\mathbf{x}_0$
• For $t = 0, 1, 2, \ldots$ do:
    – $\mathbf{x}^{(t)} := \mathbf{x}^{(t-1)} - \eta\nabla_{\mathbf{x}} f(\mathbf{x}^{(t-1)})$        // step in the gradient direction
    – $\mathbf{x}^{(t)} := \Pi_{g_i}(\mathbf{x}^{(t)})$        // project back to the feasible set
    – If $\left|f\left(\mathbf{x}^{(t)}\right) - f\left(\mathbf{x}^{(t-1)}\right)\right| < \varepsilon$, break        // convergence criterion
```

§9.2 Convex optimisation for SVMs

Recall the primal form of the hard-margin SVM optimisation problem:

$$\begin{aligned} \text{minimise} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{subject to} \quad & y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1, \quad \text{for all } i \end{aligned}$$

We can make the following observations right away:

- The objective function $\frac{1}{2}\|\mathbf{w}\|^2$ is a convex function.
- The constraints are affine, inducing a polyhedral feasible set, which is a convex set.

Thus, the support vector machine optimisation problem is a convex optimisation problem (indeed it is a quadratic programming problem), and moreover, the strong Lagrangian duality theorem holds.

Let us examine the dual. The Lagrangian is given by

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 + \sum_{i=1}^{n} \alpha_i \left(1 - y_i(\mathbf{w} \cdot \mathbf{x}_i - b)\right),$$

and the primal problem for this Lagrangian is

$$p^* = \min_{\mathbf{w},b} \max_{\boldsymbol{\alpha} \geq \mathbf{0}} L(\mathbf{w}, b, \boldsymbol{\alpha}),$$

while the dual problem is

$$d^* = \max_{\boldsymbol{\alpha} \geq \mathbf{0}} \min_{\mathbf{w},b} L(\mathbf{w}, b, \boldsymbol{\alpha}).$$

The problem d^* is unconstrained, so we can differentiate the Lagrangian with respect to $\mathbf{w}$ and b to obtain the dual problem:

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}, b, \boldsymbol{\alpha}) = \mathbf{w} - \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i = 0 \implies \mathbf{w} = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i.$$

(Note that when $\alpha_i > 0$, the corresponding $\mathbf{x}_i$ is a support vector.) Similarly, we have

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\alpha}) = -\sum_{i=1}^{n} \alpha_i y_i = 0 \implies \sum_{i=1}^{n} \alpha_i y_i = 0.$$

Therefore,

$$\begin{aligned}
\min_{\mathbf{w},b} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{1}{2}\left(\sum_i \alpha_i y_i \mathbf{x}_i\right)^\top \left(\sum_j \alpha_j y_j \mathbf{x}_j\right) + \sum_i \alpha_i - \sum_i \alpha_i y_i \left(\left(\sum_j \alpha_j y_j \mathbf{x}_j\right)^\top \mathbf{x}_i - b\right) \\
&= \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j + \sum_i \alpha_i - \sum_i \alpha_i y_i \left(\sum_j \alpha_j y_j \mathbf{x}_j^\top \mathbf{x}_i - b\right) \\
&= \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j + \sum_i \alpha_i - \sum_i \sum_j \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j + \sum_i \alpha_i y_i b \\
&= \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j.
\end{aligned}$$

We have derived the dual problem for the SVM optimisation problem:

$$\begin{aligned}
\text{maximise} \quad & \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^\top \mathbf{x}_j \\
\text{subject to} \quad & \sum_i \alpha_i y_i = 0,
\end{aligned}$$

$$\alpha_i \geq 0, \quad \text{for all } i.$$

The advantage of this formulation is that it is kernelised; we only need to access data in terms of inner products. The support vectors are the ones for which the dual variables (which are Lagrange multipliers) $\alpha_i > 0$:

Definition 9.1 (Support vector)**.** *Given a separating hyperplane $\mathbf{w} \cdot \mathbf{x}_i - b = 0$, the two supporting hyperplanes are given by $\mathbf{w} \cdot \mathbf{x}_i - b = \pm 1$. The points $\mathbf{x}_i$ for which $\mathbf{w} \cdot \mathbf{x}_i - b = \pm 1$ are called the* support vectors*.*

§10 Lecture 10—15th February, 2023

§10.1 Reproducing kernels

So far we have seen data that is perfectly linearly separable (i.e. in the hard-margin SVM case) and data that is "mostly" linearly separable (i.e. in the soft-margin SVM case). We now consider the case where the data is not linearly separable at all.

In the linear classifier, we used the affine function $\mathbf{w} \cdot \mathbf{x} + b$ to separate the data. In a similar spirit, we can define a quadratic classifier.

Definition 10.1 (Quadratic classifier)**.** *A* quadratic classifier *is a function $f \colon \mathbb{R}^d \to \mathbb{R}^d$ of the form $f(\mathbf{x}) = \langle \mathbf{x}, Q\mathbf{x} \rangle + \sqrt{2} \langle \mathbf{x}, \mathbf{p} \rangle + b$, where the weights Q is a $d \times d$ symmetric matrix, $\mathbf{p}$ is a d-dimensional vector, and the bias term b is a scalar.*

Recall from linear algebra that for all $A, B, C \in \mathbb{R}^{d \times d}$, we have $\langle AB, C \rangle = \left\langle B, A^\top C \right\rangle$ and $\langle A, BC \rangle = \left\langle AB^\top, C \right\rangle$.

Definition 10.2 (Matrix vectorisation)**.** *Given a matrix $A \in \mathbb{R}^{m \times n}$, let $\overrightarrow{A} \in \mathbb{R}^{mn}$ be its* vectorisation*, that is, the vector obtained by stacking the columns of A on top of each other. Thus, for*

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},$$

we have $\overrightarrow{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{m1} & \cdots & a_{1n} & \cdots & a_{mn} \end{bmatrix}^\top$.

Then we can write the quadratic classifier as

$$\begin{aligned} f(\mathbf{x}) &= \langle \mathbf{x}, Q\mathbf{x} \rangle + \sqrt{2} \langle \mathbf{x}, \mathbf{p} \rangle + b \\ &= \left\langle \mathbf{x}\mathbf{x}^\top, Q \right\rangle + \left\langle \sqrt{2}\mathbf{x}, \mathbf{p} \right\rangle + b \\ &= \left\langle \begin{bmatrix} \overrightarrow{\mathbf{x}\mathbf{x}^\top} \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix}, \begin{bmatrix} \overrightarrow{Q} \\ \mathbf{p} \\ b \end{bmatrix} \right\rangle. \end{aligned}$$

If we write $\phi(\mathbf{x}) = \begin{bmatrix} \overrightarrow{\mathbf{x}\mathbf{x}^\top} & \sqrt{2}\mathbf{x} & 1 \end{bmatrix}^\top$ and $\mathbf{w} = \begin{bmatrix} \overrightarrow{Q} & \mathbf{p} & b \end{bmatrix}^\top$, then we can write the quadratic classifier as

$$f(\mathbf{x}) = \langle \phi(\mathbf{x}), \mathbf{w} \rangle$$

but this really blows up the dimension to $d^2 + d + 1$. Recall that in the dual formulation of the SVM, we only needed to access data in terms of inner products, and if we have a map that reduces the dimension of the data, we can use the kernel trick for inner products to avoid computing the map explicitly; all we need to know is the inner product $\langle \phi(\mathbf{x}), \phi(\mathbf{w}) \rangle$. With this new ϕ, we get the kernel

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) := \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle &= \left\langle \begin{bmatrix} \overrightarrow{\mathbf{x}\mathbf{x}^\top} \\ \sqrt{2}\mathbf{x} \\ 1 \end{bmatrix}, \begin{bmatrix} \overrightarrow{\mathbf{z}\mathbf{z}^\top} \\ \sqrt{2}\mathbf{z} \\ 1 \end{bmatrix} \right\rangle \\ &= \left\langle \overrightarrow{\mathbf{x}\mathbf{x}^\top}, \overrightarrow{\mathbf{z}\mathbf{z}^\top} \right\rangle + \left\langle \sqrt{2}\mathbf{x}, \sqrt{2}\mathbf{z} \right\rangle + 1 \\ &= \left\langle \overrightarrow{\mathbf{x}\mathbf{x}^\top}, \overrightarrow{\mathbf{z}\mathbf{z}^\top} \right\rangle + 2\langle \mathbf{x}, \mathbf{z} \rangle + 1 \\ &= (\langle \mathbf{x}, \mathbf{z} \rangle + 1)^2 . \end{aligned}$$

This process is easily reproducible for a given ϕ. But what about the other direction?

Definition 10.3 (Reproducing kernel)**.** *A* reproducing kernel *is a symmetric function $k \colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ such that there exists a Hilbert space $\mathcal{H}$ and a map $\phi \colon \mathcal{X} \to \mathcal{H}$ such that for all $\mathbf{x}, \mathbf{z} \in \mathcal{X}$, we have $k(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$.*

Note that when such a kernel exists, it may not be unique. For example, the kernels $\phi = \begin{bmatrix} x_1^2 & \sqrt{2}x_1x_2 & x_2^2 \end{bmatrix} \in \mathbb{R}^3$ and $\varphi = \begin{bmatrix} x_1^2 & x_1x_2 & x_1x_2 & x_2^2 \end{bmatrix} \in \mathbb{R}^4$ both have the same inner product $\langle \phi(x), \phi(z) \rangle = \langle \varphi(x), \varphi(z) \rangle$.

Theorem 10.1 (Mercer's theorem)**.** *The function $k \colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a reproducing kernel if and only if for any $n \in \mathbb{N}$ and any $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathcal{X}$ and, the kernel matrix $K \in \mathbb{R}^{n \times n}$ with entries $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ is positive semi-definite and symmetric.*

Recall from linear algebra that a matrix A is positive semi-definite if for all vectors $\mathbf{v}$, we have $\mathbf{v}^\top A \mathbf{v} \geq 0$. In the context of kernels, this means that for all $\alpha \in \mathbb{R}^n$, we have $\alpha^\top K \alpha \geq 0$.

Example 10.4. Here are some more examples of kernels:

- *The polynomial kernel* $k(\mathbf{x}, \mathbf{z}) = (\langle \mathbf{x}, \mathbf{z} \rangle + 1)^p$ for some hyperparameter $p \in \mathbb{N}$.
- *The Gaussian kernel* $k(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{z}\|_2^2}{\sigma}\right)$ for some hyperparameter $\sigma > 0$.
- *The Laplace kernel* $k(\mathbf{x}, \mathbf{z}) = \exp\left(-\frac{\|\mathbf{x}-\mathbf{z}\|_1}{\sigma}\right)$ for some hyperparameter $\sigma > 0$.

We can now substitute our expression for the inner product into the primal formulation for the soft-margin SVM to get

$$\min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{i=1}^n \xi_i \quad \text{s.t.} \quad \forall i \in [n],\ 1 - y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle + b) \leq \xi_i,$$

and the dual formulation becomes

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j k(\mathbf{x}_i, \mathbf{x}_j) \quad \text{s.t.} \quad \sum_{i=1}^{n} \alpha_i y_i = 0.$$

So now we only need to compute the score function

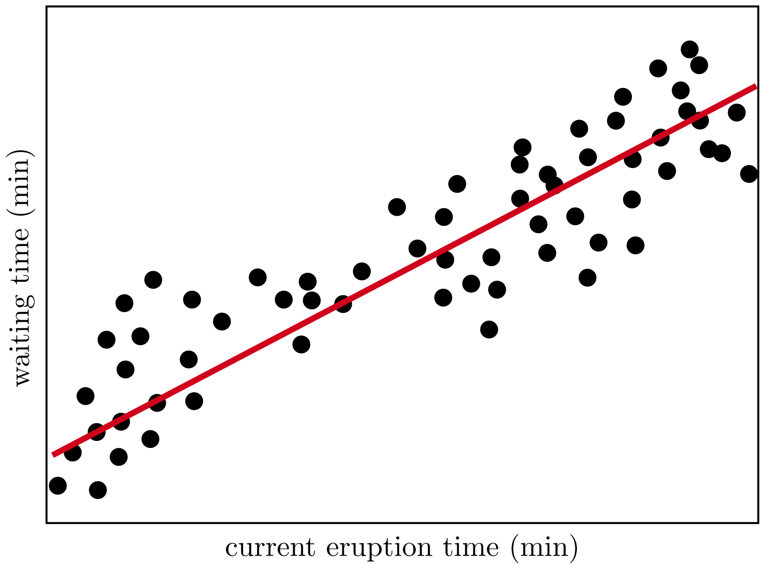
$$f(\mathbf{x}) := \langle \phi(\mathbf{x}), \mathbf{w}^* \rangle = \left\langle \phi(\mathbf{x}), \sum_{i=1}^{n} \alpha_i^* y_i \phi(\mathbf{x}_i) \right\rangle = \sum_{i=1}^{n} \alpha_i^* y_i k(\mathbf{x}, \mathbf{x}_i),$$

and so we can classify a new point $\mathbf{x}$ by computing $f(\mathbf{x})$ and checking the sign.

§10.2 Introducing linear regression

So far we have focused on classification, i.e. the problem of finding the map $f\colon \mathcal{X} \to \{1, \ldots, k\}$ that maps an input $\mathbf{x} \in \mathcal{X}$ to one of k classes. But what about other outputs? For instance we might want to predict the price of a house given its features, or the temperature tomorrow given today's weather. This is the problem of regression.

In regression, we are interested in predicting a real-valued output $y \in \mathbb{R}$ given an input $\mathbf{x} \in \mathcal{X}$, that is, we want to find a function $f\colon \mathcal{X} \to \mathbb{R}$ that maps inputs to real numbers. The simplest form of regression is linear regression, where we assume that the output is a linear function of the input. For instance, suppose we wanted to predict the next eruption time of the Old Faithful geyser in Yellowstone National Park given the duration of the current eruption. We might assume that the eruption time is a linear function of the duration of the current eruption, i.e. $y = \beta_0 + \beta_1 x$ for some $\beta_0, \beta_1 \in \mathbb{R}$, and we want to find the best values of β_0 and β_1 that fit the data we have.



So given a new data point $\mathbf{x} \in \mathcal{X}$, we want to predict an estimate $\hat{y}$ of y which minimises the discrepancy (which we will denote ℓ) between $\hat{y}$ and y. That is, we will seek to minimise either the absolute error

$$\ell(\hat{y}, y) = |\hat{y} - y|$$

or the squared error

$$\ell(\hat{y}, y) = (\hat{y} - y)^2$$

or even the 0-1 loss

$$\ell(\hat{y}, y) = \mathbb{1}\{\hat{y} \neq y\}.$$

Which of the error/loss measures we use is calibrated by the application. For example, in the case of the volcano, we might want to use the squared error loss, as we care more about the magnitude of the error than the direction.

A *linear predictor f* (shown in red in the figure) is defined as

$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0.$$

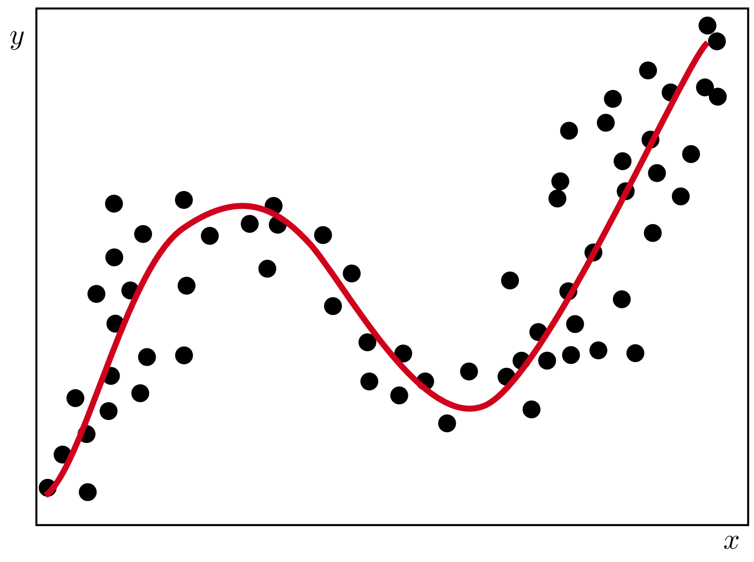
where the slope $\mathbf{w}$ and the intercept w_0 are chosen to minimise the expected prediction loss. So in this sense, the problem of regression may be thought of as a problem of finding the best linear predictor, that is, of finding the predictor $\hat{f}$ such that

$$\hat{f} = \arg\min_{f \in \mathcal{F}} \mathbb{E}_{\mathbf{x},y} \ell(f(\mathbf{x}), y).$$

where $\mathcal{F}$ is the set of all linear predictors.

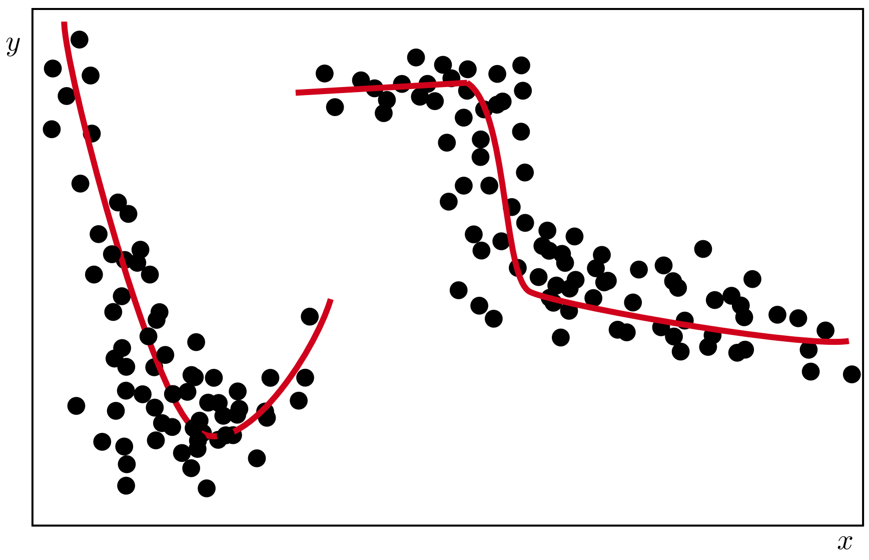
Parametric versus non-parametric methods for regression We can distinguish between parametric and non-parametric methods for regression.

Definition 10.5 (Parametric regression)**.** *A* parametric *method for regression is one in which the number of parameters is fixed as the number of training examples grows, i.e. we assume a particular form for the regressor f.*



In the case of parametric regression, the goal is to learn the parameters which yield the minimum error or loss; the problem of regression is tackled using parameters as a proxy, unlike the non-parametric case, where we are focused on learning the predictor that yields the minimu error/loss directly from the input data:

Definition 10.6 (Non-parametric regression)**.** Non-parametric *regression is a type of regression analysis in which the predictor does not take a predetermined form but is constructed based on the data.*



§11 Lecture 11—20th February, 2023

§11.1 Learning the parameters for linear regression

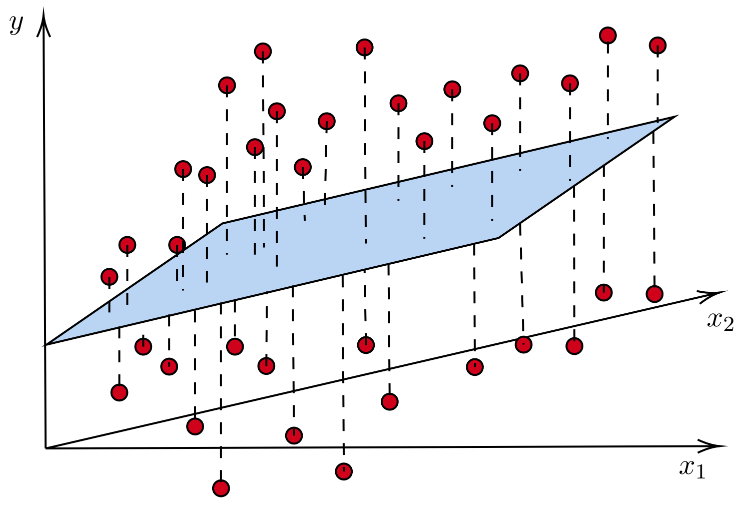
We start by discussing parametric methods for regression. Like we discussed before the goal is to find a linear predictor $\hat{f}(\mathbf{x}) := \mathbf{w} \cdot \mathbf{x}$ (after adding a bias term to the weights and a constant 1 feature to the input) that minimises the prediction loss over the prediction, i.e. appropriately solves

$$\min_{\mathbf{w}\in\mathbb{R}^n} \mathbb{E}_{\mathbf{x},y}\left[\ell\left(\hat{f}(\mathbf{x}), y\right)\right].$$

We can estimate the parameters of the regressor by minimising the corresponding loss on the training data. With the squared loss (i.e. in the ordinary least squares setting), we have

$$\begin{aligned}
\arg\min_{\mathbf{w}\in\mathbb{R}^n} \frac{1}{n}\sum_{i=1}^{n} \ell(\hat{f}(\mathbf{x}_i), y_i) &= \arg\min_{\mathbf{w}\in\mathbb{R}^n} \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w}\cdot\mathbf{x}_i, y_i) \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^n} \frac{1}{n}\sum_{i=1}^{n} (\mathbf{w}\cdot\mathbf{x}_i - y_i)^2 \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^n} \left\| \begin{bmatrix} \longleftarrow & \mathbf{x}_1 & \longrightarrow \\ \longleftarrow & \mathbf{x}_2 & \longrightarrow \\ & \vdots & \\ \longleftarrow & \mathbf{x}_n & \longrightarrow \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \right\|^2 \\
&= \arg\min_{\mathbf{w}\in\mathbb{R}^n} \|X\mathbf{w} - \mathbf{y}\|_2^2,
\end{aligned}$$

(Geometrically, we should have the following picture in mind:



This is a plot of the data $(\mathbf{x}_i, y_i)$, and the assumption is that there exists a linear relationship between the data as given by the plane $\hat{f} = \mathbf{w} \cdot \mathbf{x}$ shown. The mean squared error is given by the average of the squared differences between the predicted value and the actual value shown. The goal is to find the $\mathbf{w}$ that minimizes this error, and consequently tunes the plane to best fit the data.)

So our goal is to find the $\mathbf{w}$ that minimizes the mean squared error, and we know that such a $\mathbf{w}$ is given by the solution to the equation

$$\arg\min_{\mathbf{w}\in\mathbb{R}^n} \|X\mathbf{w} - \mathbf{y}\|_2^2 .$$

This is an unconstrained optimisation problem, and we can solve it by setting the gradient of the objective function to zero and examining the stationary points (we do not need to check the second order conditions since the objective function is convex). Therefore the best fitting $\mathbf{w}$ satisfies

$$\frac{\partial}{\partial \mathbf{w}} \|X\mathbf{w} - \mathbf{y}\|_2^2 = 2X^\top (X\mathbf{w} - \mathbf{y}) = 0,$$

and at the stationary points, we get that $X^\top X\mathbf{w} = X^\top \mathbf{y}$. As it turns out, this system is always consistent. To see why, recall that by the orthogonal decomposition of $\mathbf{y}$,

$$\mathbf{y} = \mathbf{y}_{\text{col(X)}} + \mathbf{y}_{\text{null}(X^\top)},$$

and so $X^\top \mathbf{y} = X^\top \mathbf{y}_{\text{col}(X)}$ since $X^\top \mathbf{y}_{\text{null}(X^\top)} = 0$. Now, since $\mathbf{y}_{\text{col}(X)} \in \text{col}(X)$, there exists a linear combination of the columns $\overline{\mathbf{x}_i}$ of X that equals $\mathbf{y}_{\text{col}(X)}$:

$$\mathbf{y}_{\text{col}(X)} = \sum_{i=1}^{n} w_i \overline{\mathbf{x}_i}.$$

Let $\mathbf{w} = \begin{bmatrix} w_1 & w_2 & \cdots & w_n \end{bmatrix}^\top$ be exactly the weights that achieve this linear combination. Then we have that

$$X^\top X\mathbf{w} = X^\top (X\mathbf{w}) = X^\top \left(\sum_{i=1}^{n} w_i \overline{\mathbf{x}_i} \right) = X^\top \mathbf{y}_{\text{col}(X)} = X^\top \mathbf{y},$$

and so we have exhibited a solution $\mathbf{w}$ to the system $X^\top X\mathbf{w} = X^\top \mathbf{y}$, and thus the system is always consistent. Indeed, we can always find this $\mathbf{w}$ via the Moore-Penrose pseudoinverse:

$$\mathbf{w} = (X^\top X)^\dagger X^\top \mathbf{y}.$$

§11.2 Linear regression: a geometric viewpoint

Consider the column space view of X:

$$X = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \overline{\mathbf{x}_1} & \cdots & \overline{\mathbf{x}_d} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}.$$

Here the column vectors $\overline{\mathbf{x}_i} \in \mathbb{R}^n$ span the column space of X. The goal of OLS regression as always is to find a $\mathbf{w}$ such that the linear combination of the columns of X given by $X\mathbf{w}$ is as close as possible to $\mathbf{y}$, i.e.

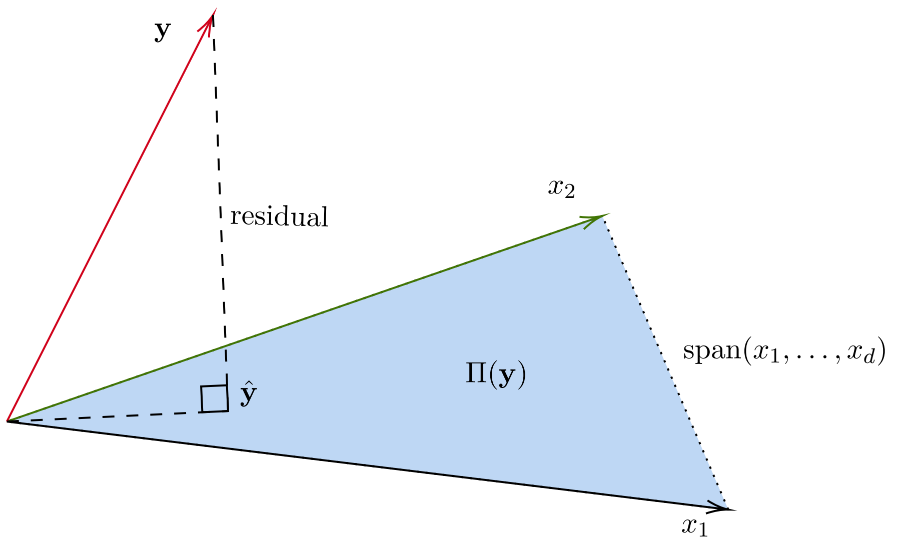
$$\frac{1}{n}\left\|\mathbf{y} - \sum_{i=1}^{d} w_i \overline{\mathbf{x}_i}\right\|_2^2,$$

which we will call the *residual*, is as small as possible. This is equivalent to finding the projection of $\mathbf{y}$ onto the subspace spanned by the columns of X.

Suppose $\hat{\mathbf{y}} \coloneqq X\mathbf{w}_{\text{ols}}$ is the OLS solution, i.e.

$$\hat{\mathbf{y}} \coloneqq X\mathbf{w}_{\text{ols}} = \sum_{i=1}^{d} w_{\text{ols},i}\overline{\mathbf{x}_i},$$

and this $\mathbf{y}$ is the orthogonal projection of $\mathbf{y}$ onto $\mathsf{span}(\overline{\mathbf{x}_1}, \ldots, \overline{\mathbf{x}_d})$. In this sense $\mathbf{w}_{\text{ols}}$ forms the coefficients of $\hat{\mathbf{y}}$ in the basis $\overline{\mathbf{x}_1}, \ldots, \overline{\mathbf{x}_d}$.



§11.3 Linear regression: a statistical modelling viewpoint

Suppose that the data is generated from the following process:

1. An example x_i is drawn independently from the data space $\mathcal{X}$, i.e. $\mathbf{x}_i \sim \mathcal{D}_{\mathcal{X}}$.
2. A label y_i^{clean} is computed as $\mathbf{w} \cdot \mathbf{x}_i$ from a fixed but unknown $\mathbf{w}$, i.e. $y_i^{\text{clean}} = \mathbf{w} \cdot \mathbf{x}_i$.

3. The label y_i is generated by adding noise to y_i^{clean}, i.e.

$$y_i = y_i^{\text{clean}} + \varepsilon_i = \mathbf{w} \cdot \mathbf{x}_i + \varepsilon_i,$$

where $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$ is a random variable that is independent of $\mathbf{x}_i$ and has mean zero.

4. The pair $(\mathbf{x}_i, y_i)$ is observed as the training data, i.e. $S = \{(x_i, y_i)\}_{i=1}^n$.

How can we determine $\mathbf{w}$ given S? Observe that

$$y_i = \mathbf{w} \cdot \mathbf{x}_i + \mathcal{N}(0, \sigma^2) = \mathcal{N}(\mathbf{w} \cdot \mathbf{x}_i, \sigma^2),$$

and so we can use maximum likelihood estimation to estimate the parameters of the Gaussian distribution that generated the data. Note that

$$\begin{aligned} \log \mathcal{L}(\mathbf{w} \mid S) &= \sum_{i=1}^n \log p(y_i \mid \mathbf{w}) \\ &\propto -\frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - \mathbf{w} \cdot \mathbf{x}_i)^2. \end{aligned}$$

Ignoring terms independent of $\mathbf{w}$, we find that the maximum likelihood estimate of $\mathbf{w}$ is the same as the OLS estimate of $\mathbf{w}$.

§11.4 Linear regression: ridge and LASSO

Let us go back to the OLS problem, i.e.

$$\arg\min_{\mathbf{w} \in \mathbb{R}^d} \|X\mathbf{w} - \mathbf{y}\|_2^2 .$$

We know that the OLS solution is given by

$$\mathbf{w}_{\text{ols}} = (X^\top X)^\dagger X^\top \mathbf{y}.$$

If we have limited data, then this solution is poorly behaved (due to overfitting). We may want to improve this somewhat, perhaps by incorporating prior knowledge. For example it could be the case that:

- only a few input features dictate or control the outcome, and so we want to encourage sparsity in the solution (this is exemplified in many applications in biology, where only a few genes control the outcome of a particular experiment—this will give rise to the LASSO regression method).
- multiple features are highly correlated and we need to find a stable relationship between the input and the response variables (this is exemplified in many applications in finance, where the prices of different stocks are highly correlated—this will give rise to the Ridge regression method).

Ridge regression In ridge regression, we add a penalty term to the OLS objective function:

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \underbrace{\|X\mathbf{w}-\mathbf{y}\|_2^2}_{\text{reconstruction error}} + \lambda \underbrace{\|\mathbf{w}\|_2^2}_{\text{penalty}},$$

where $\lambda > 0$ is a hyperparameter that controls the strength of the penalty term. Both of these terms are in competition with each other: the first term wants to fit the data as well as possible, while the second term wants to keep the weights small. If we over-optimise for the first term, we will get a solution that is very sensitive to (overfits) the data, and so the second term helps to regularise the solution. We want to balance these out so that we both have good prediction power and stably generalises to new data. At the minimum of the objective function f, we have that

$$\frac{\partial f}{\partial \mathbf{w}} = 2X^\top(X\mathbf{w}-\mathbf{y}) + 2\lambda\mathbf{w} = 0,$$

and so the solution is given by

$$\mathbf{w}_{\text{ridge}} = (X^\top X + \lambda I_{d\times d})^{-1}X^\top \mathbf{y}.$$

Can we always find a solution to this system? The answer is yes, since the matrix $X^\top X + \lambda I_{d\times d}$ is always invertible. To see why, observe that

$$\lambda I_{d\times d} \succeq 0,$$

and so $X^\top X + \lambda I_{d\times d} \succeq X^\top X$. Since $X^\top X$ is positive definite, we have that $X^\top X + \lambda I_{d\times d}$ is also positive definite, and so it is invertible.

§12 Lecture 12—22nd February, 2023

§12.1 More on ridge and LASSO regression

Continuing ridge regression Recall that ridge regression is a regularized version of linear regression:

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \underbrace{\|X\mathbf{w}-\mathbf{y}\|_2^2}_{\text{reconstruction error}} + \lambda \underbrace{\|\mathbf{w}\|_2^2}_{\text{penalty}},$$

where $\lambda > 0$ is a hyperparameter[2] that controls the trade-off between the two terms. We saw that the solution to this optimisation problem is given by $\mathbf{w}_{\text{ridge}} = (X^\top X + \lambda I_{d\times d})^{-1}X^\top \mathbf{y}$, and this solution is unique.

This problem is easily shown to be equivalent to the following optimisation problem:

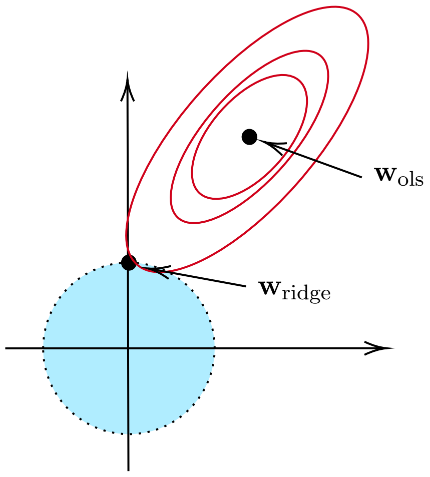
$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2^2 \quad \text{s.t.} \quad \|\mathbf{w}\|_2^2 \leq t,$$

where $t = O\left(\frac{1}{\lambda}\right)$. (To see this, note that the Lagrangian of the constrained optimisation problem is

$$\mathcal{L}(\mathbf{w},\alpha) = \|X\mathbf{w}-\mathbf{y}\|_2^2 + \alpha(\|\mathbf{w}\|_2^2 - t),$$

and the solution to the constrained optimisation problem is the same as the solution to the unconstrained optimisation problem with $\lambda = 2\alpha$.) This is a convex optimisation problem, and the solution is unique.

[2]We can choose λ using cross-validation.



LASSO regression LASSO regression is another regularized version of linear regression:

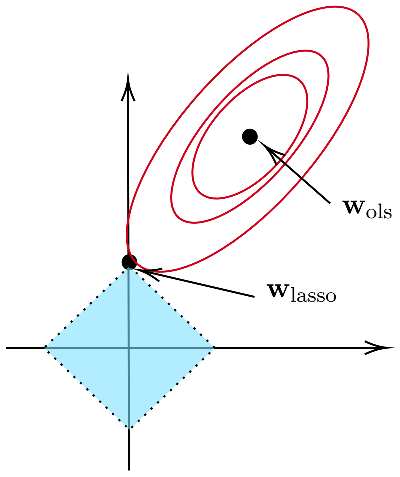
$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_1,$$

where $\lambda > 0$ is a hyperparameter that controls the trade-off between the two terms. This problem is a hard optimisation problem, and the solution is not unique. The solution encourages sparsity, i.e., many of the components of $\mathbf{w}$ are zero. This is useful for feature selection, as it can be used to identify the most important features in the data.

Just as before, the LASSO problem can be written as a constrained optimisation problem:

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2^2 \quad \text{s.t.} \quad \|\mathbf{w}\|_1 \le t,$$

where $t = O\left(\frac{1}{\lambda}\right)$. This is a convex optimisation problem.



Elastic net regression Elastic net regression is a combination of ridge and LASSO regression:

$$\arg\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2^2 + \lambda_1\|\mathbf{w}\|_1 + \lambda_2\|\mathbf{w}\|_2^2,$$

where $\lambda_1, \lambda_2 > 0$ are hyperparameters that control the trade-off between the three terms. This problem is also a hard optimisation problem, and the solution is not unique. The solution encourages sparsity and also encourages the weights to be small.

§12.2 Linear regression for classification

The gist here is that we want to derive a binary classifier from a linear regression model. We can do this by thresholding the output of the linear regression model. Specifically, given a dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, we can train a linear regression model to predict y_i from $\mathbf{x}_i$. The model is given by

$$\hat{y} = \langle \mathbf{w}, \mathbf{x} \rangle + b,$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ are the parameters of the model. We can then classify a new data point $\mathbf{x}$ as follows:

$$\hat{y} = \langle \mathbf{w}, \mathbf{x} \rangle + b \begin{cases} > 0 & \text{classify as } 1, \\ \leq 0 & \text{classify as } -1. \end{cases}$$

To do this we can fit a better model, namely the sigmoid:

$$\hat{y} = \sigma(\langle \mathbf{w}, \mathbf{x} \rangle + b),$$

where $\sigma(t) = \frac{1}{1+e^{-t}}$ is the sigmoid function. This function maps $\mathbb{R}$ to $(0,1)$, and can be interpreted as the probability of the output being 1. We can then perform the binary 0/1 classification by thresholding the output of the sigmoid function as $\operatorname{sign}(2\hat{y} - 1)$.

Here's a probabilistic interpretation of the sigmoid function. Given an input $\mathbf{x}$, how likely is it that it has label 1? Suppose the probability of $\mathbf{x}$ having label 1 is $p = \Pr(Y = 1 \mid X = \mathbf{x})$, which ranges from 0 to 1, and hence cannot be modelled appropriately via linear regression. However, consider the odds ratio $\frac{p}{1-p}$, which ranges from 0 to ∞. The log-odds ratio is then

$$\log\left(\operatorname{odds}(p)\right) \coloneqq \operatorname{logit}(p) = \log\left(\frac{p}{1-p}\right),$$

which is symmetric (i.e., $\operatorname{logit}(p) = -\operatorname{logit}(1-p)$) and ranges from $-\infty$ to ∞. This is the quantity that linear regression can model. The sigmoid function can then be obtained from the logit function.

Given an input $\mathbf{x}$,

$$\operatorname{logit}\left(\Pr(Y = 1 \mid X = \mathbf{x})\right) = \operatorname{logit}(p) = \log\left(\frac{p}{1-p}\right) = \langle \mathbf{w}, \mathbf{x} \rangle .$$

Solving for p, we get

$$p = \frac{1}{1 + e^{-\langle \mathbf{w}, \mathbf{x} \rangle}} = \sigma(\langle \mathbf{w}, \mathbf{x} \rangle).$$

This is the probabilistic interpretation of the sigmoid function. So now we have a probabilistic model for classification, and we can use the maximum likelihood principle to estimate the parameters $\mathbf{w}$ and b. Given samples $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ where $y_i \in \{0, 1\}$, the likelihood of the data is

$$\mathcal{L}(\mathbf{w} \mid S) = \prod_{i=1}^n \Pr((\mathbf{x}_i, y_i) \mid \mathbf{w})$$

$$= \prod_{i=1}^{n} \Pr(Y = y_i \mid X = \mathbf{x}_i, \mathbf{w}) \Pr(X = \mathbf{x}_i, \mathbf{w}) \propto \prod_{i=1}^{n} \Pr(Y = y_i \mid X = \mathbf{x}_i, \mathbf{w})$$

where the last step follows because the prior probability of the data is constant. Therefore the binomial likelihood is

$$\begin{aligned}
\mathcal{L}(w \mid S) &= \prod_{i=1}^{n} \Pr(y_i = 1 \mid \mathbf{x}_i, \mathbf{w})^{y_i} \Pr(y_i = 0 \mid \mathbf{x}_i, \mathbf{w})^{1-y_i} \\
&= \prod_{i=1}^{n} \Pr(y_i = 1 \mid \mathbf{x}_i, \mathbf{w})^{y_i} \left(1 - \Pr(y_i = 1 \mid \mathbf{x}_i, \mathbf{w})\right)^{1-y_i}.
\end{aligned}$$

Taking the log-likelihood, we get

$$\begin{aligned}
\log \mathcal{L}(\mathbf{w} \mid S) &= \sum_{i=1}^{n} y_i \log \Pr(y_i = 1 \mid \mathbf{x}_i, \mathbf{w}) + (1 - y_i) \log \left(1 - \Pr(y_i = 1 \mid \mathbf{x}_i, \mathbf{w})\right) \\
&= \sum_{i=1}^{n} y_i \log \frac{p_{x_i}}{1 - p_{x_i}} + (1 - y_i) \log \frac{1 - p_{x_i}}{p_{x_i}} \\
&= \sum_{i=1}^{n} y_i \langle \mathbf{w}, \mathbf{x}_i \rangle + \sum_{i=1}^{n} - \log \left(1 + e^{\langle \mathbf{w}, \mathbf{x}_i \rangle}\right).
\end{aligned}$$

This is the negative log-likelihood, which we want to minimise. We can do this using gradient ascent. The gradient of the negative log-likelihood is given by

$$\nabla \log \mathcal{L}(\mathbf{w} \mid S) = \sum_{i=1}^{n} \left(y_i - \sigma(\langle \mathbf{w}, \mathbf{x}_i \rangle)\right) \mathbf{x}_i.$$

Given a step size η, we can update the weights using the update rule

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \nabla \log \mathcal{L}(\mathbf{w} \mid S).$$

This is the gradient ascent algorithm for logistic regression. We can also use the Newton-Raphson method to update the weights. The update rule is

$$\mathbf{w} \leftarrow \mathbf{w} - \left(\nabla^2 \log \mathcal{L}(\mathbf{w} \mid S)\right)^{-1} \nabla \log \mathcal{L}(\mathbf{w} \mid S),$$

where $\nabla^2 \log \mathcal{L}(\mathbf{w} \mid S)$ is the Hessian of the negative log-likelihood. This is the Newton-Raphson algorithm for logistic regression.

§13 Lecture 13—27th February, 2023

§13.1 Optimality of the regressor

Linear regression (and its variants) are great, but what can we concretely say about the best possible estimate of the target variable? Can we construct an estimator for real outputs that parallels the optimality of the Bayes classifier for discrete outputs in classification?

Indeed we can. We claim that the best possible estimate of the target variable is the conditional expectation of the target variable given the input. That is, the best possible estimate of y given x is

$$f^*(\mathbf{x}) = \mathbb{E}[Y \mid X = \mathbf{x}] = \int y \cdot p(y \mid \mathbf{x}) \, \mathrm{d}y,$$

as the following theorem shows.

Theorem 13.1. *For any regression estimate $g(\mathbf{x})$, the expected squared error*

$$\mathbb{E}_{(\mathbf{x},y)}\left[|f^*(\mathbf{x}) - y|^2\right] \leq \mathbb{E}_{(\mathbf{x},y)}\left[|g(\mathbf{x}) - y|^2\right].$$

Proof. Pick your favourite $\mathbf{x} \in \mathbb{R}^d$ and take $\rho := \mathbb{E}[(g(\mathbf{x}) - y)^2 \mid X = \mathbf{x}]$. Then

$$\begin{aligned}
\rho &= \mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}) + f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \\
&= \mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}))^2 + 2(g(\mathbf{x}) - f^*(\mathbf{x}))(f^*(\mathbf{x}) - y) + (f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \\
&= \mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}))^2 \mid X = \mathbf{x}] + 2\mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}))(f^*(\mathbf{x}) - y) \mid X = \mathbf{x}] \\
&\qquad + \mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \\
&= \mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}))^2 \mid X = \mathbf{x}] + 2(f^*(\mathbf{x}) - \mathbb{E}[y \mid X = \mathbf{x}])\mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x})) \mid X = \mathbf{x}] \\
&\qquad + \mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \\
&= \mathbb{E}[(g(\mathbf{x}) - f^*(\mathbf{x}))^2 \mid X = \mathbf{x}] + \mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \\
&\geq \mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}].
\end{aligned}$$

Thus we only need to integrate over $\mathbf{x}$ to get the desired result. Assuming a continuous domain on which the joint distribution of (X, Y) is defined, we have

$$\mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \leq \mathbb{E}[(g(\mathbf{x}) - y)^2 \mid X = \mathbf{x}],$$

and integrating over $\mathbf{x}$ gives

$$\int \mathbb{E}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \, p(\mathbf{x}) \, \mathrm{d}\mathbf{x} \leq \int \mathbb{E}[(g(\mathbf{x}) - y)^2 \mid X = \mathbf{x}] \, p(\mathbf{x}) \, \mathrm{d}\mathbf{x},$$

which is an iterated expectation:

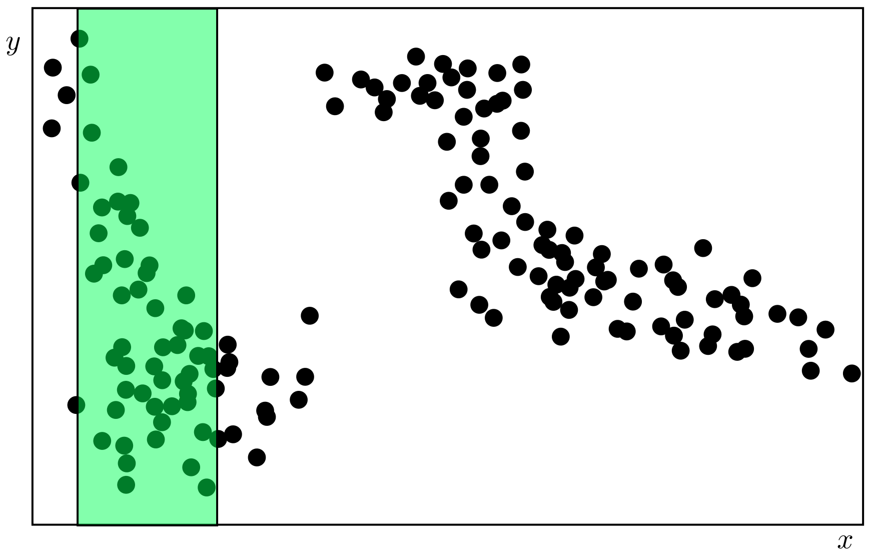
$$\mathbb{E}\left[\mathbb{E}_{(\mathbf{x},y)}[(f^*(\mathbf{x}) - y)^2 \mid X = \mathbf{x}]\right] \leq \mathbb{E}\left[\mathbb{E}_{(\mathbf{x},y)}[(g(\mathbf{x}) - y)^2 \mid X = \mathbf{x}]\right].$$

The result follows. □

§13.2 Non-parametric regression

Now, what if we don't know the parametric form of the relationship between the independent and dependent variables? How can we predict the value of a new test point $\mathbf{x}$ without any model assumptions? Here's an idea:

1. Find the k training points closest to $\mathbf{x}$.
2. Average the target values of these k points in the neighbourhood of $\mathbf{x}$.



For this we want $\hat{y}$ to be

$$\hat{y} = \hat{f}_n(\mathbf{x}) = \sum_{i=1}^{n} w_i(\mathbf{x}) y_i$$

where w_i are the weights constructed so as to emphasise local observations. There are a few "localisation functions" that can successfully accomplish this, including but not limited to:

- The Gaussian kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2h^2}\right)$$

  where h is the bandwidth parameter.

- The Epanechnikov kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} \frac{3}{4}\left(1 - \frac{\|\mathbf{x}-\mathbf{x}'\|^2}{h^2}\right) & \text{if } \|\mathbf{x} - \mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

- The Tri-cube kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} \frac{70}{81}\left(1 - \left(\frac{\|\mathbf{x}-\mathbf{x}'\|}{h}\right)^3\right)^3 & \text{if } \|\mathbf{x} - \mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

- The Quartic kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} \frac{15}{16}\left(1 - \left(\frac{\|\mathbf{x}-\mathbf{x}'\|}{h}\right)^2\right)^2 & \text{if } \|\mathbf{x}-\mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

- The Cosine kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} \frac{\pi}{4}\cos\left(\frac{\pi}{2}\frac{\|\mathbf{x}-\mathbf{x}'\|}{h}\right) & \text{if } \|\mathbf{x}-\mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

- The Truncated Gaussian kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} \exp\left(-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2h^2}\right) & \text{if } \|\mathbf{x}-\mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

- The triangle kernel:

$$K_h(\mathbf{x}, \mathbf{x}') = \begin{cases} 1 - \frac{\|\mathbf{x}-\mathbf{x}'\|}{h} & \text{if } \|\mathbf{x}-\mathbf{x}'\| < h \\ 0 & \text{otherwise} \end{cases}$$

With any of these kernels, we can define the localised regression estimate as

$$\hat{f}_n(\mathbf{x}) = \sum_{i=1}^{n} \underbrace{\frac{K_h(\mathbf{x}, \mathbf{x}_i)}{\sum_{j=1}^{n} K_h(\mathbf{x}, \mathbf{x}_j)}}_{:=w_i(\mathbf{x})} y_i$$

where h is the bandwidth parameter. The choice of h is crucial, as it determines the size of the neighbourhood around $\mathbf{x}$ that we consider when making the prediction. If h is too small, the estimate will be too sensitive to noise, while if h is too large, the estimate will be too smooth and will not capture the local structure of the data. The choice of h can be made using cross-validation.

Now recall that the best possible regression estimate at a point $\mathbf{x}$ is given by $f^*(\mathbf{x}) = \mathbb{E}[Y \mid X = \mathbf{x}]$.

Theorem 13.2 (Consistency theorem). *As $n \to \infty$ and $h \to 0$ such that $nh \to \infty$, the localised regression estimate $\hat{f}_{n,h}(\mathbf{x})$ obtained via the kernel regressor converges to the best possible regression estimate $f^*(\mathbf{x})$, that is,*

$$\mathbb{E}_{(\mathbf{x},y)}\left[\left|\hat{f}_{n,h}(\mathbf{x}) - f^*(\mathbf{x})\right|^2\right] \to 0$$

where $\hat{f}_{n,h} := \sum_{i=1}^{n} \frac{K_h(\mathbf{x},\mathbf{x}_i)}{\sum_{j=1}^{n} K_h(\mathbf{x},\mathbf{x}_j)} y_i$ and $f^*(\mathbf{x}) = \mathbb{E}[Y \mid X = \mathbf{x}]$.

Proof sketch. The proof is a bit tedious, so we only present a sketch. Fix your favourite $\mathbf{x}$ and integrate over the distribution of $(\mathbf{x}, y)$ to get the bias-variance decomposition of the mean squared error of the localised regression estimate at $\mathbf{x}$:

$$\mathbb{E}\left[\left|\hat{f}_{n,h}(\mathbf{x}) - f^*(\mathbf{x})\right|^2\right] = \underbrace{\mathbb{E}\left[\left|\mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x})\right] - f^*(\mathbf{x})\right|\right]^2}_{\text{squared bias of } \hat{f}_{n,h}} + \underbrace{\mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x}) - \mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x})\right]\right]^2}_{\text{variance of } \hat{f}_{n,h}}$$

It can be shown that

$$\begin{aligned}
\mathbb{E}\left[\left|\mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x})\right] - f^*(\mathbf{x})\right|\right]^2 &\approx c_1 h^2 \\
\mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x}) - \mathbb{E}\left[\hat{f}_{n,h}(\mathbf{x})\right]\right]^2 &\approx c_2 \cdot \frac{1}{nh^d}.
\end{aligned}$$

Pick $h \approx n^{-1/2+d}$. Then

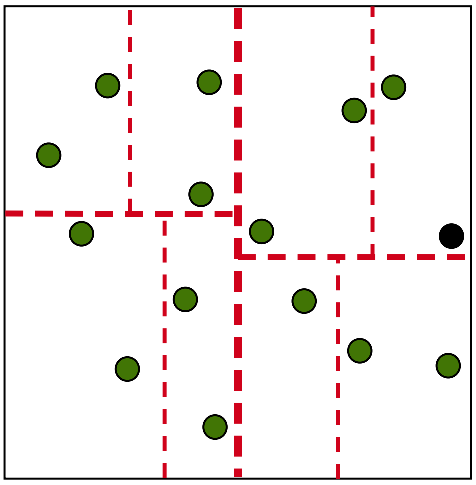
$$\mathbb{E}\left[\left|\hat{f}_{n,h}(\mathbf{x}) - f^*(\mathbf{x})\right|^2\right] \approx c_1 h^2 + c_2 \cdot \frac{1}{nh^d} \approx c_1 n^{-2+2d} + c_2 n^{-1+d} \to 0$$

as $n \to \infty$ and $h \to 0$ such that $nh \to \infty$. $\square$

Speeding up non-parametric regression The localised regression estimate $\hat{f}_{n,h}(\mathbf{x})$ requires $O(n)$ operations to compute. This is fine for small datasets, but can be prohibitively slow for large datasets. We can speed up the computation by using the *K-D tree* data structure, which allows us to find the k nearest neighbours of a point $\mathbf{x}$ in $O(\log n + k)$ operations. The K-D tree is a binary tree that recursively partitions the data into smaller and smaller regions. The tree is constructed as follows:

1. Start with the entire dataset.
2. Find the dimension with the largest spread in the data.
3. Find the median value of the data in that dimension.
4. Split the data into two parts based on the median value.
5. Recursively apply steps 2-4 to each of the two parts.

To find the k nearest neighbours of a point $\mathbf{x}$, we start at the root of the tree and recursively traverse the tree, keeping track of the k nearest neighbours. At each node, we check if the region corresponding to the node is closer to $\mathbf{x}$ than the k-th nearest neighbour. If it is, we recursively traverse the child nodes. If it is not, we prune the branch and move to the next node. The k-D tree can be used to speed up the computation of the localised regression estimate $\hat{f}_{n,h}(\mathbf{x})$ by replacing the $O(n)$ search for the k nearest neighbours with a $O(\log n + k)$ search using the k-D tree.



§14 Lecture 14—29th February, 2023

§14.1 Some more analysis of linear least-squares regression

Now let us revert to the linear least-squares regression problem. We have the following model:

$$y_i = \mathbf{w} \cdot \mathbf{x}_i + w_0.$$

Suppose we have performed the lifting transformation $\mathbf{x} \mapsto (\mathbf{x}, 1)$, so that the model becomes

$$y_i = \mathbf{w} \cdot \mathbf{x}_i.$$

Does solving least-squares regression work in this case? Consider an empirical observation model, in which we will analyse what happens as the number of data points increases. Suppose that the observations are perturbed by noise $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$. After estimating $\hat{\mathbf{w}}$ from the data, we make predictions $\hat{y}_i = \hat{\mathbf{w}} \cdot \mathbf{x}_i$. We can then write the prediction error as

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - \mathbb{E}[y_i])^2 = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{w}} \cdot \mathbf{x}_i - \mathbf{w} \cdot \mathbf{x}_i)^2 = \frac{1}{n}\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2,$$

where $\mathbb{E}[y_i] = \mathbb{E}[\langle \mathbf{w}, \mathbf{x}_i \rangle + \varepsilon_i] = \langle \mathbf{w}, \mathbf{x}_i \rangle$, and $\mathcal{X}$ is the matrix whose i-th row is $\mathbf{x}_i$. The following theorem shows that the prediction error converges to zero as the number of data points increases, which is a good property for a learning algorithm to have.

Theorem 14.1. *Let X be a matrix whose rows are $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, and let $\mathbf{w} \in \mathbb{R}^d$. Let $\hat{\mathbf{w}}$ be the solution to the least-squares regression problem. Then*

$$\lim_{n\to\infty} \mathbb{E}_{\boldsymbol{\varepsilon}}\left[\frac{1}{n}\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2\right] = 0.$$

Proof. Recall that $\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \|X\mathbf{w} - \mathbf{y}\|_2^2 = (X^\top X)^\dagger X^\top \mathbf{y}$. Thus

$$\begin{aligned}
\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2^2 &= \|X\hat{\mathbf{w}} - y + w_0\|_2^2 \\
&= \|X\hat{\mathbf{w}} - y\|_2^2 + \|w_0\|_2^2 + 2\left\langle X\hat{\mathbf{w}} - y, w_0 \right\rangle \\
&\le \|X\mathbf{w} - y\|_2^2 + \|w_0\|_2^2 + 2\left\langle X\hat{\mathbf{w}} - y, w_0 \right\rangle \\
&= \|X\mathbf{w} - y\|_2^2 + \|w_0\|_2^2 + 2\left\langle X\hat{\mathbf{w}} - X\mathbf{w} - w_0, w_0 \right\rangle \\
&= \|X\mathbf{w} - y\|_2^2 + \|w_0\|_2^2 + 2\left\langle X\hat{\mathbf{w}} - X\mathbf{w}, w_0 \right\rangle - 2\|w_0\|_2^2 \\
&= 2\left\langle X\hat{\mathbf{w}}, w_0 \right\rangle - 2\left\langle X\mathbf{w}, w_0 \right\rangle.
\end{aligned}$$

Now we take expectations over $\boldsymbol{\varepsilon}$, and use the fact that $\mathbb{E}[\boldsymbol{\varepsilon}] = 0$ and that $\boldsymbol{\varepsilon}$ is independent of X and $\mathbf{w}$ to get

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{\varepsilon}}\left[\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2^2\right] &\le 2\left\langle X\hat{\mathbf{w}}, w_0 \right\rangle - 2\left\langle X\mathbf{w}, w_0 \right\rangle \\
&= \left\langle \boldsymbol{\varepsilon}^\top X\hat{\mathbf{w}} \right\rangle \\
&= \left\langle \boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top \mathbf{y} \right\rangle \\
&= \left\langle \boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top (X\mathbf{w} + \boldsymbol{\varepsilon}) \right\rangle
\end{aligned}$$

$$
\begin{aligned}
&= 2\mathbb{E}\left[\boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top X\mathbf{w}\right] + \mathbb{E}\left[\boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top \boldsymbol{\varepsilon}\right] \\
&= 2\mathbb{E}[\boldsymbol{\varepsilon}^\top X\mathbf{w}] + \mathbb{E}[\boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top \boldsymbol{\varepsilon}] \\
&= 0 + \mathbb{E}[\boldsymbol{\varepsilon}^\top X(X^\top X)^\dagger X^\top \boldsymbol{\varepsilon}] \\
&= \mathbb{E}[\boldsymbol{\varepsilon}^\top \underbrace{X(X^\top X)^\dagger X^\top}_{:=Q} \boldsymbol{\varepsilon}].
\end{aligned}
$$

Observe that $\boldsymbol{\varepsilon}^\top Q\boldsymbol{\varepsilon}$ is a quadratic form in $\boldsymbol{\varepsilon}$; it is a scalar, and so it is equal to $\mathsf{trace}(\boldsymbol{\varepsilon}^\top Q\boldsymbol{\varepsilon})$.Then since

$$
\mathsf{trace}(ABC) = \mathsf{trace}(BCA) = \mathsf{trace}(CAB),
$$

we have that $\boldsymbol{\varepsilon}^\top Q\boldsymbol{\varepsilon} = \mathsf{trace}(Q\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top)$. Now we can write

$$
\begin{aligned}
\mathbb{E}_{\boldsymbol{\varepsilon}}\left[\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2^2\right] &\leq 2\mathbb{E}[\boldsymbol{\varepsilon}^\top Q\boldsymbol{\varepsilon}] \\
&= 2\mathbb{E}[\mathsf{trace}(Q\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top)] \\
&= 2\mathsf{trace}(Q\mathbb{E}[\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^\top]) \\
&= 2\mathsf{trace}(Q\sigma^2 I) \\
&= 2\sigma^2\mathsf{trace}(Q).
\end{aligned}
$$

Putting $Q = X(X^\top X)^\dagger X^\top$ into the above equation, we get

$$
\begin{aligned}
\mathbb{E}_{\boldsymbol{\varepsilon}}\left[\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2^2\right] &\leq 2\sigma^2\mathsf{trace}(X(X^\top X)^\dagger X^\top) \\
&= 2\sigma^2\mathsf{trace}(X^\top X(X^\top X)^\dagger) \\
&= 2\sigma^2\mathsf{trace}(I) \\
&= 2\sigma^2 d.
\end{aligned}
$$

Overall then,

$$
\lim_{n\to\infty} \mathbb{E}_{\boldsymbol{\varepsilon}}\left[\frac{1}{n}\|X\hat{\mathbf{w}} - X\mathbf{w}\|_2\right] = \frac{2\sigma^2 d}{n} \to 0,
$$

and the proof is complete. □

§14.2 Non-linear least-squares regression

With non-linear least-squares regression, we assume that

$$
\mathbb{E}[Y \mid X = \mathbf{x}] = \left\langle \mathbf{w}^\top, \phi(\mathbf{x}) \right\rangle + w_0,
$$

where $\phi\colon \mathbb{R}^d \to \mathbb{R}^m$ is a non-linear feature map. Then the *empirical risk minimisation* strategy with the square loss is

$$
\min_{\mathbf{w}\in\mathbb{R}^m, w_0\in\mathbb{R}} \mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^n (y_i - \langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - w_0)^2 = \|\Phi\mathbf{w} - \mathbf{y}\|_2^2,
$$

where Φ is the matrix whose i-th row is $\phi(\mathbf{x}_i)$. But what if $\phi(\mathbf{x})$ is very high-dimensional or is even infinite-dimensional? In this case, we cannot compute the matrix Φ, and so we cannot solve the least-squares regression problem. We can, however, use the kernel trick to solve this problem.

We now introduce *kernelised ridge regression*, in which we add a regularisation term which ensures that there are no issues if the kernel matrix K with $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ is not invertible. We define

$$\min_{\mathbf{w}\in\mathbb{R}^n, w_0\in\mathbb{R}} \mathcal{L}(\mathbf{w}) = \min_{\mathbf{w}\in\mathbb{R}^n, w_0\in\mathbb{R}} \frac{1}{n}\sum_{i=1}^{n} (y_i - \langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - w_0)^2 + \lambda \|\mathbf{w}\|_2^2,$$

where $\lambda > 0$ is the regularisation parameter. Taking the gradient with respect to $\mathbf{w}$ and setting it to 0, we get

$$\lambda \mathbf{w} + \Phi^\top \Phi \mathbf{w} = \Phi^\top \mathbf{y},$$

so that

$$\mathbf{w} = \sum_{i=1}^{n} \alpha_i \phi(\mathbf{x}_i),$$

where α_i are variables satisfying $n\lambda\alpha_i = y_i - \langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle$. With this representation of $\mathbf{w}$, we can calculate the prediction for a new point $\mathbf{x}$ as

$$\langle \mathbf{w}, \phi(\mathbf{x})\rangle = \sum_{i=1}^{n} \alpha_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x})\rangle = \sum_{i=1}^{n} \alpha_i k(\mathbf{x}_i, \mathbf{x}).$$

We still need to find the α_i that minimise the loss function. We have

$$n\lambda\alpha_i = y_i - \sum_{j=1}^{n} \alpha_j k(\mathbf{x}_j, \mathbf{x}_i),$$

and so we can write this as

$$\begin{bmatrix} n\lambda & k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_n) \\ k(\mathbf{x}_2, \mathbf{x}_1) & n\lambda & \cdots & k(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_n, \mathbf{x}_1) & k(\mathbf{x}_n, \mathbf{x}_2) & \cdots & n\lambda \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

This is a linear system of equations, and so we can solve it to get the α_i as

$$\boldsymbol{\alpha} = (K + n\lambda I)^{-1}\mathbf{y}.$$

§15 Lecture 15—19th March, 2023

§15.1 Formalising the concept of learnability

Until now, we haven't had a clear definition of what does it mean to be able to learn, and need to answer that. We also want to ask, what hypothesis class should we choose and what limitations do different hypothesis classes have. Further more, given a hypothesis class, we would like to discuss and determine what kinds of learning rules should we use, and how many data points do we need to learn a good model. With these questions in mind, we will start our discussion from simple settings and then try to generalise our conclusions.

Realisability We start our discussion with a simplifying assumption, realizability. Formally, realizability means that there exists an optimal hypothesis $h^* \in \mathcal{H}$ such that the true risk $L_{\mathbb{P}}(h^*) = 0$. This is a strong assumption and it implies that with probability 1, over random samples $S \sim \mathbb{P}$, $L_S(h^*) = 0$.

However, this strong assumption only implies the existence of such a hypothesis that can give 0 error, it is not guaranteed that the particular hypothesis h_S found by minimizing any ERM is the optimal hypothesis h^*. The realizability assumption makes sure our hypothesis class is "rich" or "capable" enough, so that we don't need to worry about under-fitting, but we can still be overfitting by only minimizing the empirical error. Thus, we want to further discuss that under this assumption, what is the risk of the ERM hypothesis h_S on the unseen data and can this risk be bounded such that we are guaranteed to find a good hypothesis?

ε-δ learning To quantitatively measure how good our hypothesis is, we introduce the ε and δ parameters for our discussion. The ε parameter is the *accuracy parameter* and is used to quantify the quality of the prediction. Concretely, we interpret the event $L_{\mathbb{P}}(h_S) > \varepsilon$ as a failure of the learner, while if $L_{\mathbb{P}}(h_S) \leq \varepsilon$, we view the output of the ERM as an approximately correct hypothesis.

However, this single parameter is not enough because h_S depends on the training set S, and the training set is picked by a random process so that there is randomness in the result of the ERM. It is not realistic to expect that with full certainty S will suffice to direct the learner toward a good hypothesis, as there is always some probability that the sampled training data happens to be very non-representative of the underlying distribution $\mathbb{P}$. We therefore denote the probability of getting a non-representative sample by δ, and call $(1 - \delta)$ the *confidence parameter* of our prediction.

To bound the error of the ERM hypothesis h_S, we further introduce some restrictions on the hypothesis class $\mathcal{H}$ so that we can prevent overfitting. The simplest type of restriction on a class is imposing an upper bound on its size, that is, the hypothesis class $\mathcal{H}$ has a finite cardinality. With this additional assumption, we can show that the ERM hypothesis will not overfit, i.e., have a bounded error on unseen data.

Theorem 15.1. *Let $\mathcal{H}$ be finite. Let $\delta \in (0,1)$, $\varepsilon > 0$ and $N \geq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$. Then, for any distribution $\mathbb{P}$ for which realizability holds, we have with probability at least $1 - \delta$ over the choice of dataset S of size N, every ERM hypothesis h_S satisfies $L_{\mathbb{P}} \leq \varepsilon$.*

Proof. Let $\mathcal{H}_B$ be the set of "failed" hypotheses, that is $\mathcal{H}_B = \{h \in \mathcal{H} : L_{\mathbb{P}}(h) > \varepsilon\}$. In addition, let M be the set of misleading samples, that is $M = \{S : \exists h \in \mathcal{H}_B, L_S(h) = 0\}$. Namely, for every $S \in M$, there is a 'failed' hypothesis, $h \in \mathcal{H}_B$, that looks like a 'good' hypothesis on S.

Now, recall that we would like to bound the probability of the event $L_{\mathbb{P}}(h_S) > \varepsilon$. Since the realizability implies that $L_S(h_S) = 0$, it follows that the event $L_{\mathbb{P}}(h_S) > \varepsilon$ can only happen if for some $h \in \mathcal{H}_B$, we have $L_S(h) = 0$.

In other words, the failure will only happen if our training data is in the set of misleading samples M. Formally, we have $\{S : L_{\mathbb{P}}(h_S) > \varepsilon\} \subseteq M$ as we can write M as $M = \bigcup_{h \in \mathcal{H}_B} \{S : L_S(h) = 0\}$. Hence,

$$\Pr(\{S : L_{\mathbb{P}}(h_S) > \varepsilon\}) \leq \Pr\left(\bigcup_{h \in \mathcal{H}_B} \{S : L_S(h) = 0\}\right)$$

Applying the union bound to the right-hand side yields

$$\Pr(\{S : L_{\mathbb{P}}(h_S) > \varepsilon\}) \leq \sum_{h \in \mathcal{H}_B} \Pr(\{S : L_S(h) = 0\})$$

Next, we can bound each summand of the right-hand side. Fix some 'failed' hypothesis $h \in \mathcal{H}_B$, the event $L_S(h) = 0$ is equivalent to the event that in the training set, $\forall i, h(x_i) = y_i$. Since the training data are i.i.d. sampled, we have

$$\Pr(\{S : L_S(h) = 0\}) = \prod_{i=1}^{N} \Pr(\{x_i : h(x_i) = y_i\}) = (1 - L_{\mathbb{P}}(h))^N \leq (1 - \varepsilon)^N$$

where the last inequality follows from the fact that $h \in \mathcal{H}_B$.

Using the inequality $1 - \varepsilon \leq e^{-\varepsilon}$, we have for every $h \in \mathcal{H}_B$,

$$\Pr(\{S : L_S(h) = 0\}) \leq (1 - \varepsilon)^N \leq e^{-\varepsilon N}$$

Therefore, we have

$$\Pr(\{S : L_{\mathbb{P}}(h_S) > \varepsilon\}) \leq |\mathcal{H}_B| e^{-\varepsilon N} \leq |\mathcal{H}| e^{-\varepsilon N}$$

Let $\delta = |\mathcal{H}| e^{-\varepsilon N}$, we reach the desired conclusion that with probability at least $1 - \delta$, and having $N \geq \frac{\log(|\mathcal{H}|/\delta)}{\varepsilon}$, $L_{\mathbb{P}}(h_S) \leq \varepsilon$. □

§15.2 PAC learning and Occam's razor

We see that the finite hypothesis class makes it possible to bound the unseen error of ERM hypothesis. In order to generalize this result, we first give a formal name of such hypothesis classes. As we are using the ε and δ parameters which implies the conclusion is both approximate and not determined, we use the name probably approximately correct learnability, also known as *PAC learnability*. A formal definition is as follows,

Definition 15.1 (PAC learnability)**.** *Assuming realizability, a hypothesis class $\mathcal{H}$ is* PAC-learnable *if there exists a function $N_{\mathcal{H}}(\varepsilon, \delta)$ and a learning algorithm with the following property: for every $\varepsilon, \delta \in (0, 1)$ and every distribution $\mathbb{P}$, training using $N \geq N_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d. samples generated from $\mathbb{P}$, the learning algorithm returns a hypothesis h such that $L_{\mathbb{P}} \leq \varepsilon$ with confidence $(1 - \delta)$ over choice of samples.*

Informally, PAC-learnability of class $\mathcal{H}$ means that enough number of random examples drawn from the data distribution will allow approximate risk minimization, i.e., ensure $L_{\mathbb{P}}(h) \leq \varepsilon$ with probability $\geq 1 - \delta$, where the number of samples needed depends on the desired tolerances (ε, δ).

Note here ε and δ are inevitable. δ arises due to the randomness of training data S drawn from $\mathbb{P}$ and ε arises due to the actual hypothesis picked by the learner on the finite data S.

With this formal concept of PAC-learnable defined, we can discuss the situations when our two assumptions on realizability and finite hypothesis class do not hold. Concretely, is the hypothesis class still learnable if realizability does not hold? And on the other hand, what about infinite hypothesis classes? Are they PAC-learnable?

Agnostic PAC-learnability We first release the realizability assumption. By No-Free-Lunch (NFL) theorem, we know that no learner is guaranteed to match the Bayes classifier in general, as there's always an adversarial distribution that can be constructed on which our learner fails while another may succeed. Thus, if the realizability does not hold, we don't have any hope of satisfying $L_{\mathbb{P}} \leq \varepsilon$.

We now can only weaken our aim, and see if we can at least come ε-close to the best possible classifier within our hypothesis class with high probability, i.e. $L_{\mathbb{P}}(h_S) \leq \inf_{h' \in \mathcal{H}} \mathbb{P}(h') + \varepsilon$. In this setting, the hypothesis class $\mathcal{H}$ may be bad, but we can still try to be approximately as good as the best possible hypothesis within this class. This weaker property is known as agnostic PAC-Learnability.

Definition 15.2 (Agnostic PAC learnability)**.** *A hypothesis class $\mathcal{H}$ is agnostic PAC learnable if there exists a function $N_{\mathcal{H}} \colon (0,1)^2 \to \mathbb{N}$ and a learning algorithm with the following property: For every $\varepsilon, \delta \in (0,1)$ and for every distribution $\mathbb{P}$ over $\mathcal{X} \times \mathcal{Y}$, when running the learning algorithm on $N > N_{\mathcal{H}}(\varepsilon, \delta)$ i.i.d. samples generated by $\mathbb{P}$, the algorithm returns a hypothesis h such that, with probability of at least $1 - \delta$ over the choice of the N training samples,*

$$L_{\mathbb{P}}(h) \leq \inf_{h' \in \mathcal{H}} \mathbb{P}(h') + \varepsilon.$$

Clearly, if the realizability assumption holds, Agnostic PAC-Learnability provides the same guarantee as PAC-Learnability. In that sense, Agnostic PAC-Learnability generalizes the definition of PAC-Learnability. When the realizability assumption does not hold, no learner can guarantee an arbitrarily small error. Nevertheless, under the definition of Agnostic PAC learning, a learner can still declare success if its error is not much larger than the best error achievable by a predictor from the class $\mathcal{H}$. This is in contrast to PAC Learning, in which the learner is required to achieve a small error in absolute terms and not relative to the best error achievable by the hypothesis class.

We can decompose the error into the approximation error term and the estimation error term, where

$$L_{\mathbb{P}}(h_S) = \varepsilon_{\text{apx}} + \varepsilon_{\text{est}}$$

where $\varepsilon_{\text{apx}} := \min_{h \in \mathcal{H}} L(h)$ and $\varepsilon_{\text{est}} := L_{\mathbb{P}}(h_S) - \varepsilon_{\text{apx}}$. As the approximation error depends on the fit of our prior knowledge via the inductive bias to the unknown underlying distribution, it won't be minimized further more after we've chosen the hypothesis class $\mathcal{H}$. The Agnostic PAC-Learnability loosens the bound on this term but bounds the estimation error uniformly over all distributions for a given hypothesis class.

Uniform convergence $\Longrightarrow$ PAC-learnability How can we make sure the ERM solution is close to the true risk? One strong assumption one can make is that $L_S(h)$ for all $h \in \mathcal{H}$ is close to the true risk $L_{\mathbb{P}}(h)$, then the ERM solution h_S will also have small true risk $L_{\mathbb{P}}(h_S)$. Hence, we introduce the notion of an ε-representative data sample.

Definition 15.3 (Representative sample)**.** *A sample S is* ε-representative *for a hypothesis class $\mathcal{H}$ with respect to a distribution $\mathbb{P}$ if for all $h \in \mathcal{H}$, we have*

$$|L_S(h) - L_{\mathbb{P}}(h)| \leq \varepsilon$$

The next simple conclusion we can make is that whenever the sample is $\varepsilon/2$-representative, the ERM learning rule is guaranteed to return a good hypothesis.

Theorem 15.2. *Assume S is $\varepsilon/2$-representative. Then, any ERM solution $h_S \in \arg\min_{h\in\mathcal{H}} L_S(h)$ satisfies*

$$L_{\mathbb{P}} \leq \min_{h\in\mathcal{H}} L_{\mathbb{P}} + \varepsilon.$$

Proof. For every $h \in \mathcal{H}$,

$$L_{\mathbb{P}} \leq L_S(h_S) + \varepsilon/2 \leq L_S(h) + \varepsilon/2 \leq L_{\mathbb{P}}(h) + \varepsilon/2 + \varepsilon/2 = L_{\mathbb{P}}(h) + \varepsilon. \qquad \square$$

The simple theorem implies that to ensure that the ERM rule is Agnostic PAC-Learnable, it suffices to show that with probability of at least $1-\delta$ over the random choice of a training set, it will be an ε-representative training set. The following uniform convergence condition formalizes this requirement.

Definition 15.4 (Uniform convergence property)**.** *A hypothesis class $\mathcal{H}$ has the* uniform convergence property *with respect to a domain $\mathcal{Z}$ and a loss function ℓ, if there exists a function $N_{\mathcal{H}}^{UC} : (0,1)^2 \to \mathbb{N}$ such that for every $\varepsilon, \delta \in (0,1)$ and for every probability distribution $\mathbb{P}$ over $\mathcal{Z}$, if S is a sample of $N \geq N_{\mathcal{H}}^{UC}(\varepsilon,\delta)$ i.i.d. examples drawn from $\mathbb{P}$, then, with probability of at least $1-\delta$, S is ε-representative.*

Similar to the definition of sample complexity for PAC learning, the function $N_{\mathcal{H}}^{UC}$ measures the minimal sample complexity of obtaining the uniform convergence property, namely, how many examples we need to ensure that with probability of at least $1-\delta$ the sample would be ε-representative. The term uniform here refers to having a fixed sample size that works for all members of $\mathcal{H}$ and over all possible probability distributions over the domain. The following corollary follows directly from the previous theorem and the definition of uniform convergence.

Corollary 15.5. *If a class $\mathcal{H}$ has the uniform convergence property with a function $N_{\mathcal{H}}^{UC}$, then the class is Agnostically PAC learnable with the sample complexity $N_{\mathcal{H}}(\varepsilon,\delta) \leq N_{\mathcal{H}}^{UC}(\varepsilon/2,\delta)$.*

§16 Lecture 16—21st March, 2023

§16.1 Concentration of measure

It is known from the central limit theorem (CLT) that given independent and identically sampled $X_1, X_2, \dots, X_n$ with $\mathbb{E}[X_i] = 0$ and $\mathrm{Var}(X_i) = \sigma^2$, we have

$$\frac{\sum_{i=1}^n X_i}{\sigma\sqrt{n}} \longrightarrow \mathcal{N}(0,1) \text{ in distribution.}$$

Write $S_n = \sum_{i=1}^n X_i$, $S_n \sim O(\sqrt{n})$.

Remark 16.1. *Notice that if $X_1, X_2, \dots, X_n$ are not mutually independent, then S_n does not necessarily scale like $O(\sqrt{n})$. For example, when $X_1 = X_2 = \cdots = X_n$, $S_n \sim O(n)$.*

While CLT ensures the convergence of S_n in distribution, it does not give information on the rate of convergence, which is the main topic of this lecture. In terms of S_n, we want a result of the form $\Pr[|S_n - \mathbb{E}[S_n]| > \alpha] \leq ?$, where $\alpha \sim O(\sqrt{n})$. More generally, given a function $F(X_1, X_2, \dots, X_n)$, we want to bound the probability

$$\Pr[|F(X_1, X_2, \dots, X_n) - \mathbb{E}[F(X_1, X_2, \dots, X_n)]| > \alpha] \leq ?$$

To motivate the study of concentration of measure, we give three applications:

1. *The Johnson-Lindenstrauss lemma.* This lemma is a mainstay in the area of dimensionality reduction.

   Lemma 16.2 (Johnson-Lindenstrauss)**.** *Given a set X of n points in $\mathbb{R}^D$, $0 < \varepsilon < 1$, and a number $d > \dfrac{8 \ln n}{\varepsilon^2}$, there exists a linear map $f\colon \mathbb{R}^D \to \mathbb{R}^d$ such that*

   $$(1-\varepsilon)\|u - v\| \leq \|f(u) - f(v)\| \leq (1+\varepsilon)\|u - v\|$$

   for all $u, v \in X$.

   Remark 16.3. *We have $f \cdot u = \sqrt{\frac{D}{d}} P \cdot u$, where P is a projection to a d-dimensional subspace and $u \in \mathbb{R}^D$.*

2. *Fast randomized SVD.* Let G be a $n \times k$ matrix where $k \ll n$, take $Gij \sim_{\text{i.i.d}} \mathcal{N}(0,1)$ and G is well-conditioned. Let u, v be different columns of the normalized matrix $\frac{G}{\sqrt{n}}$. We have that $\mathbb{E}\left[\|u\|^2\right] = 1$ and $\mathbb{E}[u^\top v] = 0$. Hence, $\frac{G}{\sqrt{n}}$ has "almost" orthonormal columns and singular values close to 1.

3. *Spectral norm of Wigner matrices.* Let W be a $n \times n$ symmetric matrix with $W_{ij} \sim_{\text{i.i.d}} \mathcal{N}(0,1)$ and $W_{ii} \sim_{\text{i.i.d}} \mathcal{N}(0,2)$. The spectral norm of W is $\lambda_1(W)$, the largest eigenvalue of W. It is known that $\lambda_1(W) \sim O(\sqrt{n})$.

We now introduce the concept of concentration of measure.

Definition 16.4. *Let $X_1, X_2, \dots, X_n$ be independent random variables. We say that a function $F(X_1, X_2, \dots, X_n)$ satisfies a* concentration of measure *inequality if there exist constants $c, C > 0$ such that for all $\alpha > 0$, $\Pr[|F(X_1, X_2, \dots, X_n) - \mathbb{E}[F(X_1, X_2, \dots, X_n)]| > \alpha] \leq Ce^{-c\alpha^2}$.*

Some important concentration of measure inequalities include:

Proposition 16.5 (Markov's inequality). *Let X be a non-negative random variable with $\mathbb{E}[X] < \infty$. Then, for $\alpha > 0$,*

$$\Pr[X > \alpha] \leq \frac{\mathbb{E}[X]}{\alpha}.$$

Proof. We have that $\mathbb{E}[X] = \mathbb{E}[X(\mathbb{1}_{X>\alpha} + \mathbb{1}_{X\leq\alpha})] \geq \mathbb{E}[X(\mathbb{1}_{X>\alpha})] \geq \mathbb{E}[\alpha\mathbb{1}_{X>\alpha}] = \alpha \Pr[X > \alpha]$. The conclusion follows. □

Proposition 16.6 (Chebyshev's inequality). *For any random variable X with $\mathbb{E}[X] = \mu$ and $\mathrm{Var}(X) = \sigma^2$, we have*

$$\Pr[|X - \mu| > \alpha] \leq \frac{\sigma^2}{\alpha^2}.$$

Proof. We have that $\Pr[|X - \mu| > \alpha] = \Pr[(X - \mu)^2 > \alpha^2] \leq \dfrac{\mathbb{E}[(X-\mu)^2]}{\alpha^2} = \dfrac{\mathrm{Var}(X)}{\alpha^2}$. □

Our goal is control the large deviation of S_n. Let us first consider the most basic case, where $X_1, X_2, \ldots, X_n$ are *i.i.d.* (Constraints on identical distribution or independence can be relaxed.)

For $t > 0$,

$$\Pr(S_n > \alpha) = \Pr\left(e^{tS_n} > e^{t\alpha}\right) \leq \frac{\mathbb{E}(\exp(tS_n))}{e^{t\alpha}}.$$

Hence it suffices to give an upper bound of $\mathbb{E}[\exp(tS_n)]$. By the *i.i.d* property of $\{X_i\}$,

$$\mathbb{E}[\exp(tS_n)] = \mathbb{E}\left[\exp\left(t\sum_{i=1}^n X_i\right)\right] = \prod_{i=1}^n \mathbb{E}\left[e^{tX_i}\right] = \left(\mathbb{E}\left[e^{tX_1}\right]\right)^n$$

If we assume $\mathbb{E}e^{tX_1} \leq e^{ct^2}$ for some constant $c > 0$ (which holds under rather general conditions), the upper bound is then obtained:

$$\Pr(S_n > \alpha) \leq \frac{\mathbb{E}(\exp(tS_n))}{e^{t\alpha}} \leq \frac{e^{ct^2 n}}{e^{\alpha t}} = e^{cnt^2 - \alpha t}$$

Since the above inequality holds for any $t > 0$, we can pick $t^* = \frac{\alpha}{2cn}$ so that it minimizes the expression on the right hand side of the inequality. Plugging in t^* gives

$$\mathbb{P}(S_n > \alpha) \leq e^{\frac{-\alpha}{4cn}}$$

Notice that we pick $\alpha \sim O(\sqrt{n})$ so that the above bound gives meaningful results.

This is the basic idea of concentration of measure. The key is to control the moment generating function $\mathbb{E}[\exp(tS_n)]$ of S_n, and by so doing, obtain similar results. One example is *Hoeffding's inequality.*

Theorem 16.1 (Hoeffding's inequality). *Suppose $X_1, X_2, \ldots, X_n$ are independent random variables with $a_i \leq X_i \leq b_i$. Write $S_n = \sum_{i=1}^n X_i$. Then, for $\alpha > 0$,*

$$P(|S_n - \mathbb{E}S_n| > \alpha) \leq 2\exp\left(\frac{-2\alpha^2}{\sum_{i=1}^n (b_i - a_i)^2}\right)$$

In order to prove the inequality, let us first introduce Hoeffding's lemma.

Lemma 16.7 (Hoeffding's lemma)**.** *Suppose X is a random variable such that $a \leq X \leq b$ for some constant a, b. For any $t > 0$, we have*

$$\mathbb{E}e^{tX} \leq \exp\left(\frac{t^2(b-a)^2}{8}\right).$$

Proof. By convexity, for any $x \in (0,1)$ we have $e^{tX} \leq xe^{tb} + (1-x)e^{ta}$. Hence,

$$e^{tX} \leq \frac{X-a}{b-a}e^{tb} + \frac{b-X}{b-a}e^{ta}.$$

Then

$$\mathbb{E}e^{tX} \leq \frac{\mathbb{E}X-a}{b-a}e^{tb} + \frac{b-\mathbb{E}X}{b-a}e^{ta} = \frac{-a}{b-a}e^{tb} + \frac{b}{b-a}e^{t}a.$$

Let $h = t(b-a), p = \dfrac{-a}{b-a}$ and $L(h) = -hp + \ln(1-p+pe^h)$. Then $\frac{-a}{b-a}e^{tb} + \frac{b}{b-a}e^{t}a = e^{L(h)}$.

Taking derivative of $L(h)$, $L(0) = L^{'}(0) = 0$ and

$$L^{''}(h) = \frac{(1-p)pe^h}{(1-p+pe^h)^2} \leq \frac{(1-p)pe^h}{4(1-p)pe^h} = \frac{1}{4}$$

for all h. By Taylor's expansion, $L(h) \leq \dfrac{1}{8}h^2 = \dfrac{1}{8}t^2(b-a)^2$. Hence, $\mathbb{E}e^{tX} \leq e^{\frac{1}{8}t^2(b-a)^2}$ ☐

Proof of Theorem 16.1. Without loss of generality we can consider the case where each X_i has $\mathbb{E}X_i = 0$. Otherwise we just replace X_i with $\bar{X}_i = X_i - \mathbb{E}X_i$ in later analysis. Notice that if $a_i \leq X_i \leq b_i$, then $a_i - \mathbb{E}X_i \leq \bar{X}_i \leq b_i - \mathbb{E}X_i$, and $(b_i - \mathbb{E}X_i) - (a_i - \mathbb{E}X_i) = b_i - a_i$. Hence the result remains the same.

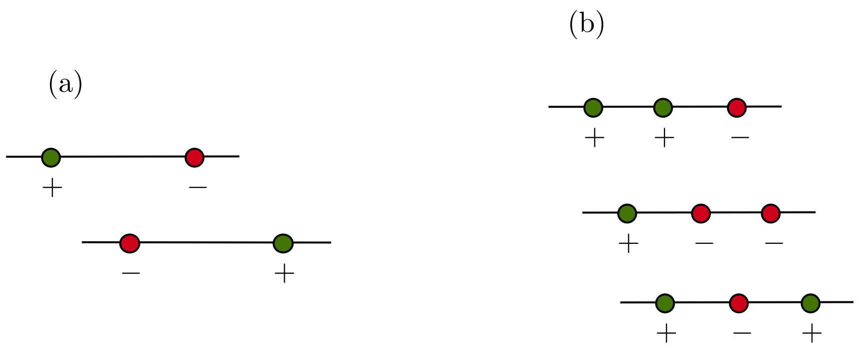
With the above being said, from now on we work with X_i that has $\mathbb{E}X_i = 0$.

$$\begin{aligned}\Pr(S_n > \alpha) = \Pr(e^{tS_n} > e^{t\alpha}) &\leq \frac{\mathbb{E}e^{tS_n}}{e^{t\alpha}} = \frac{\prod_{i=1}^n \mathbb{E}e^{tX_i}}{e^{t\alpha}} \\ &\leq e^{\frac{t^2\sum_{i=1}^n(b_i-a_i)^2}{8}}e^{-t\alpha}\end{aligned}$$

By picking $t = \frac{4\alpha}{\sum_{i=1}^n(b_i-a_i)^2}$ we can minimize the right hand side of the inequality, which gives $\Pr(S_n > \alpha) \leq \exp\left(\frac{-2\alpha^2}{\sum_{i=1}^n(b_i-a_i)^2}\right)$. Following the same steps, $\Pr(S_n < -\alpha) = \Pr(-S_n > \alpha) \leq \exp\left(\frac{-2\alpha^2}{\sum_{i=1}^n(b_i-a_i)^2}\right)$. Therefore, $\Pr(|S_n - \mathbb{E}S_n| > \alpha) \leq 2\exp\left(\frac{-2\alpha^2}{\sum_{i=1}^n(b_i-a_i)^2}\right)$. ☐

§16.2 The VC dimension and the fundamental theorem of learnability

Let us now resume our discussion of learning. Now, let's move to the situation of infinite hypothesis class Clearly, we don't have a measurement for the size of the hypothesis class any more, but it is still possible to quantitatively measure complexity of the model. For learnability in classification problems, what really matters is not the literal size of the hypothesis class, but the maximum number of data points that can be classified exactly.



Take the simple situation in Figure 1 for example; the hypothesis class of 1-dimensional linear classifier has an infinite size, but this doesn't mean this class is a very complex class. As shown in (a) above, two points with whatever labels can be classified correctly by a linear classifier, but in (b), we can see that this no longer holds for 3 points, as the last example in (b) cannot be classified correctly by any hypothesis in the linear classifier class. This inspires us that in order to measure the richness of our hypothesis class, we can try to construct a subset C of the data domain for which our classifier fails or succeeds. To understand the power of our hypothesis class, we just focus on its behavior on C and try to check how many different possible classification decisions on C can our hypothesis class capture. Then, if the hypothesis class can explain all decisions possible on C, then one can construct a 'misleading data distribution' so that we maintain realisability on C but can be totally wrong on the part outside of C and thus suffer large risk. This implies that to achieve learnability, we need to restrict the size of C. Linear classifiers in 1D can shatter 2 points as in (a), but cannot classifier the last case correctly in (b). Thus the VC-Dimension of 1-D linear classifiers is 2. To be more formal, here we introduce the definition of restriction of $\mathcal{H}$ to C and the VC-dimension.

Definition 16.8. *Let $\mathcal{H}$ be a class of functions from $\mathcal{X}$ to $\{0,1\}$ and let $\mathcal{C} = \{c_1, \ldots, c_m\} \subseteq \mathcal{X}$. The restriction of $\mathcal{H}$ to $\mathcal{C}$ is the set of functions from $\mathcal{C}$ to $\{0,1\}$ that can be derived from $\mathcal{H}$. That is,*

$$\mathcal{H}_\mathcal{C} = \{(h(c_1), \ldots, h(c_m)) : h \in \mathcal{H}\}$$

where we present each function from $\mathcal{C}$ to $\{0,1\}$ as a vector in $\{0,1\}^{|\mathcal{C}|}$.

If the restriction of H to C is the set of all functions from C to $\{0,1\}$, then we say H shatters the set C. Formally:

Definition 16.9 (Shattering)**.** *A hypothesis class $\mathcal{H}$* shatters *finite set $\mathcal{C} \subseteq \mathcal{X}$ if the restriction of $\mathcal{H}$ to $\mathcal{C}$ is the set of all functions from $\mathcal{C}$ to $\{0, 1\}$. That is, $|\mathcal{H}_\mathcal{C}| = 2^{|\mathcal{C}|}$.*

With the idea of shattering in, we are now ready to present the definition of VC dimension.

Definition 16.10 (VC-dimension)**.** *The* VC-dimension *of a hypothesis class $\mathcal{H}$, denoted* $\text{VCdim}(\mathcal{H})$*, is the maximal size of a set $\mathcal{C} \subseteq \mathcal{X}$ that can be shattered by $\mathcal{H}$. If $\mathcal{H}$ can shatter sets of arbitrarily large size, we say that $\mathcal{H}$ has infinite VC-dimension.*

To show that $\text{VCdim}(\mathcal{H}) = d$, we need to prove two things:

1. There exists a set $\mathcal{C}$ of size d that is shattered by $\mathcal{H}$, which proves $\text{VCdim}(\mathcal{H}) \geq d$;
2. No set of size $d+1$ is shattered by $\mathcal{H}$, which proves $\text{VCdim}(\mathcal{H}) < d+1$. Thus $\text{VCdim}(\mathcal{H}) = d$.

Though we showed in homework that the VC dimension of d-dimensional linear classifier is $d+1$, most of the time, we can only have lower/upper bound of VC dimension, but not an exact computable number. Thus, it is important to understand the meaning of the lower and upper bound of VC-Dimension.

This motivates the fundamental theorem of learning, which, as the name suggests, is quite important:

Theorem 16.2 (Fundamental theorem of statistical learning)**.** *Let $\mathcal{H}$ be a hypothesis class of functions from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the* 0-1 *loss. Then the following are equivalent:*

1. *$\mathcal{H}$ has the uniform convergence property.*
2. *Any ERM rule is a successful agnostic PAC learner for $\mathcal{H}$.*
3. *$\mathcal{H}$ is agnostic PAC learnable.*
4. *$\mathcal{H}$ is PAC learnable.*
5. *Any ERM rule is a successful PAC learner for $\mathcal{H}$.*
6. *$\mathcal{H}$ has a finite VC-dimension.*

In our previous discussion, we saw $1 \to 2$. $2 \to 3$, $3 \to 4$ and $2 \to 5$ are all trivial. For $4 \to 6$ and $5 \to 6$, there is detailed proof in [SSBD14] through the no-free-lunch theorem. Here, we take a closer look at $6 \to 1$, that a finite VC-dimension implies the uniform convergence property, and therefore is PAC-learnable. The detailed proof can be found in chapter 6 of [SSBD14]; here we provide a high level sketch of the proof. The two main parts of the proof are:

1. If $\text{VCdim}(\mathcal{H}) = d$, when restricting to a finite subset $\mathcal{C}$ of the data domain, its effective size $|\mathcal{H}_{\mathcal{C}}|$ is only $O(|\mathcal{C}|^d)$, instead of exponential in $|\mathcal{C}|$.
2. Finite hypothesis class can be proved to have the uniform convergence property by a direct application of Hoeffding inequality plus the union bound theorem. Similarly, the uniform convergence holds whenever the "effective size" is small.

To define the term "effective size", we introduce the definition of Growth Function:

Definition 16.11. *Let $\mathcal{H}$ be a hypothesis class. Then the growth function of $\mathcal{H}$, denoted $\tau_{\mathcal{H}} \colon \mathbb{N} \to \mathbb{N}$, is defined as*

$$\tau_{\mathcal{H}}(N) = \max_{\mathcal{C} \subseteq \mathcal{X} : |\mathcal{C}| = N} |\mathcal{H}_{\mathcal{C}}|$$

In words, $\tau_{\mathcal{H}}(N)$ is the number of different functions from a set $\mathcal{C}$ of size N to $\{0,1\}$ that can be obtained by restricting $\mathcal{H}$ to $\mathcal{C}$. We then can prove the Sauer's lemma that can bound this growth function.

Lemma 16.12. *Let $\mathcal{H}$ be a hypothesis class with* $\mathrm{VCdim}(\mathcal{H}) \leq d < \infty$. *Then for all N,*

$$\tau_{\mathcal{H}}(N) \leq \sum_{i=0}^{d} \binom{N}{i}.$$

In particular, if $N > d+1$ then $\tau_{\mathcal{H}}(N) \leq (eN)^d$.

Thus, finite VC-dimension implies polynomial growth, while infinite VC-dim means exponential growth. Intuitively, for any $\mathcal{C}$ as a subset of $\mathcal{X}$, let B be a subset of $\mathcal{C}$ such that $\mathcal{H}$ shatters B.

Then, $|\mathcal{H}_{\mathcal{C}}| \leq \#\{B \subseteq \mathcal{C} : \mathcal{H} \text{ shatters } B\}$. That is, if $\mathcal{C}$ is the collection of subsets of $\mathcal{C}$ that are shattered by $\mathcal{H}$, then $|\mathcal{H}_{\mathcal{C}}|$ is upper-bounded by the cardinality of $\mathcal{C}$. Then we can show the ERM error is bounded using the growth function.

Theorem 16.3. *Let $\mathcal{H}$ be a class and $\tau_{\mathcal{H}}$ its growth function. Then for every distribution $\mathbb{P}(X,Y)$ and every $\delta \in (0,1)$, with probability at least $1-\delta$ over the choices of $S \sim \mathbb{P}$, we have*

$$|L_S(h) - L_{\mathbb{P}}(h)| \leq \frac{4 + \sqrt{\log \tau_{\mathcal{H}}(2N)}}{\delta\sqrt{2N}}$$

And it follows from here that if $\mathrm{VCdim}(\mathcal{H})$ is finite, then the uniform convergence property holds, and indeed, $N_{\mathcal{H}}^{UC}(\varepsilon,\delta) \leq O\left(\frac{d}{(\delta\varepsilon)^2}\right)$ suffices for the uniform convergence property to hold. A more quantitative version of this theorem is as follows, and the proof can be found in Chapter 28 of [SSBD14].

Theorem 16.4. *Let $\mathcal{H}$ be a hypothesis class of functions from a domain $\mathcal{X}$ to $\{0,1\}$ and let the loss function be the 0-1-loss. Assume that* $\mathrm{VCdim}(\mathcal{H}) = d < \infty$. *Then, there are absolute constants C_1, C_2 such that:*

1. *$\mathcal{H}$ has uniform convergence with sample complexity* $C_1 \frac{d+\log(1/\delta)}{\varepsilon^2} \leq N_{\mathcal{H}}^{UC}(\varepsilon,\delta) \leq C_2 \frac{d+\log(1/\delta)}{\varepsilon^2}$,
2. *$\mathcal{H}$ is agnostic PAC learnable with sample complexity* $C_1 \frac{d+\log(1/\delta)}{\varepsilon^2} \leq N_{\mathcal{H}}(\varepsilon,\delta) \leq C_2 \frac{d+\log(1/\delta)}{\varepsilon^2}$,
3. *$\mathcal{H}$ is PAC learnable with sample complexity* $C_1 \frac{d+\log(1/\delta)}{\varepsilon} \leq N_{\mathcal{H}}(\varepsilon,\delta) \leq C_2 \frac{\log(1/\varepsilon)+\log(1/\delta)}{\varepsilon}$.

§17 Lecture 17—26th March, 2023

§17.1 Introducing unsupervised learning

In unsupervised learning, we are given a data set $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, where $x^{(i)} \in \mathbb{R}^d$ and m is the number of examples. The goal is to find some structure in the data. We are not given any labels, so we are not trying to predict anything. We are just trying to find some structure in the data that makes sense and is ideally useful. There are two aspects of unsupervised learning for us:

- *Clustering.* We want to group the data into meaningful structures. For example, in the case of a social network, we might want to group people into different communities and find out who is friends with whom.

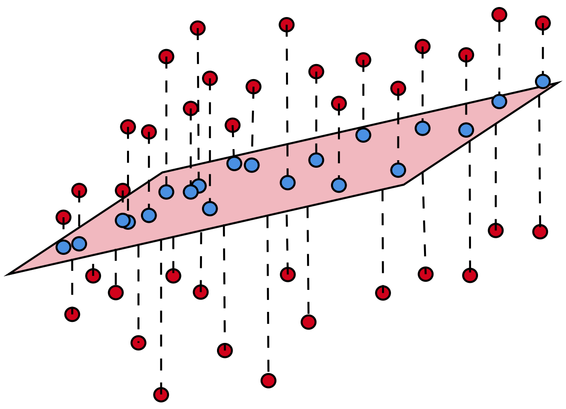
- *Dimensionality reduction.* We want to find a lower-dimensional representation of the data that retains relevant information and suppresses irrelevant/noisy information. For example, if we have a dataset with $n = 1000$, we might want to find a way to represent the data in a lower-dimensional space, say $n = 100$.

The classic example is the handwritten digit dataset, where we have $n = 784$ (since the images are 28×28) and $m = 60,000$. We can use unsupervised learning to find a lower-dimensional representation of the data that retains relevant information. This is useful because it can help us visualise the data, and it can also help us compress the data and make it easier to work with. (I will not go into the details of how this is done in this lecture because of how intuitive it is.)

We start our discussion of unsupervised learning with *dimensionality reduction*. We will discuss two methods: (1) *Principal Component Analysis (PCA).* This is a linear method that finds a lower-dimensional representation of the data that retains the most variance. (2) *t-Distributed Stochastic Neighbor Embedding (t-SNE).* This is a non-linear method that finds a lower-dimensional representation of the data that retains the most structure.

§17.2 Principal component analysis

In principal component analysis (PCA), we are given a dataset $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$, where $x^{(i)} \in \mathbb{R}^d$. The goal is to find a lower-dimensional representation of the data that retains the most variance, or more precisely stated, find the best linear transformation $\phi \colon \mathbb{R}^d \to \mathbb{R}^k$ that best maintains the reconstruction accuracy.



This problem is equivalent to that of minimising the aggregate residual error. We want to find the best subspace (depicted in light red) such that we can optimally reconstruct the data, that is, that minimises the aggregate residual error (depicted in dashed lines). Here the reconstruction error is

$$\frac{1}{n}\sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{p}_i\|^2$$

where $\mathbf{p}_i$ is the projection of $\mathbf{x}_i$ onto the subspace. We will use the orthogonal projection onto the subspace:

Definition 17.1. *The* k-dimensional linear orthogonal projection *of a vector* $\mathbf{x}$ *onto a subspace* $S \subset \mathbb{R}^d$ *is the map* $\Pi^k \colon \mathbb{R}^d \to \mathbb{R}^d$ *defined by*

$$\Pi^k(\mathbf{x}) = \sum_{i=1}^{k} \langle \mathbf{x}, \mathbf{v}_i \rangle \mathbf{v}_i$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_k$ *is an orthonormal basis for* S*.*

Here the problem is to find the best k-dimensional subspace such that the aggregate residual error is minimised, i.e.

$$\min_{\Pi^k \subset \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - \Pi^k(\mathbf{x}_i) \right\|^2 .$$

How do we optimise this?

Dimensionality reduction via projections Given the k-dimensional subspace, we can represent it in terms of its basis, really, the orthonormal basis $\mathbf{v}_1, \ldots, \mathbf{v}_k$. Then the projection of any $\mathbf{x}_i \in \mathbb{R}^d$ in $\mathsf{span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ is given by

$$\sum_{j=1}^{k} \langle \mathbf{x}_i, \mathbf{v}_j \rangle \mathbf{v}_j = \underbrace{\left(\sum_{j=1}^{k} \mathbf{v}_j \mathbf{v}_j^\top \right)}_{\Pi^k} \mathbf{x}_i .$$

Thus the representation in $\mathbb{R}^k$ via orthonormal basis $\mathbf{v}_1, \ldots, \mathbf{v}_k$ has coefficients $\langle \mathbf{x}_i, \mathbf{v}_1 \rangle, \ldots, \langle \mathbf{x}_i, \mathbf{v}_k \rangle$.

The case $k = 1$ In this case, we have a one-dimensional subspace, and we are looking for such a $\mathbf{v}$ that solves the optimisation problem

$$\min_{\mathbf{v} \in \mathbb{R}^d, \|\mathbf{v}\|=1} \frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - \left(\mathbf{v}\mathbf{v}^\top \right) \mathbf{x}_i \right\|^2 .$$

We can readily check that

$$\begin{aligned}
\frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - \left(\mathbf{v}\mathbf{v}^\top \right) \mathbf{x}_i \right\|^2 &= \left(\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i^\top \mathbf{x}_i \right) - \mathbf{v}^\top \left(\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top \right) \mathbf{v} \\
&\propto -\mathbf{v}^\top \left(\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top \right) \mathbf{v} \\
&= -\mathbf{v}^\top \left(\frac{1}{n} \mathbf{X}\mathbf{X}^\top \right) \mathbf{v},
\end{aligned}$$

where $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_n]$ and each $\mathbf{x}_i \in \mathbb{R}^d$. Therefore an equivalent formulation of the problem is

$$\max_{\mathbf{v} \in \mathbb{R}^d, \|\mathbf{v}\|=1} \mathbf{v}^\top \left(\frac{1}{n} \mathbf{X}\mathbf{X}^\top \right) \mathbf{v}.$$

Here, the matrix $M = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$ is the empirical covariance matrix of the data. We now investigate how to solve the problem $\max_{\mathbf{v} \in \mathbb{R}^d, \|\mathbf{v}\|=1} \mathbf{v}^\top M \mathbf{v}$ for even a generic matrix M. Recall the spectral decomposition of a symmetric matrix:

Theorem 17.1. *Let $M \in \mathbb{R}^{d\times d}$ be a symmetric matrix. Then there exists an orthonormal basis of eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_d$ of M such that*

$$M = \sum_{i=1}^{d} \lambda_i \mathbf{v}_i \mathbf{v}_i^\top = \mathbf{V}\boldsymbol{\Lambda}\mathbf{V}^\top,$$

where $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_d \geq 0$ are the eigenvalues of M.

Therefore for any unit length vector $\mathbf{v}$, we have, with $\sum_{i=1}^{d} \langle \mathbf{v}, \mathbf{x}_i\rangle^2 = 1$,

$$\mathbf{v}^\top M \mathbf{v} = \sum_{i=1}^{d} \lambda_i \langle \mathbf{v}, \mathbf{x}_i\rangle^2.$$

Let $\alpha_i = \langle \mathbf{v}, \mathbf{x}_i\rangle^2$. Then the problem becomes

$$\begin{aligned} \max_{\alpha_i \in \mathbb{R}} \quad & \sum_{i=1}^{d} \lambda_i \alpha_i \\ \text{s.t.} \quad & \sum_{i=1}^{d} \alpha_i = 1, \alpha_i \geq 0, \end{aligned}$$

and it is clear that the optimal solution is $\alpha_i = 1$ for the largest eigenvalue λ_1 and $\alpha_i = 0$ for all other eigenvalues; equivalently, $\mathbf{v}$ is the eigenvector corresponding to the largest eigenvalue. Therefore the optimal solution to the problem is the eigenvector corresponding to the largest eigenvalue of the empirical covariance matrix $\frac{1}{n}\mathbf{X}\mathbf{X}^\top$. So here's a basic recipe for finding the principal component for the case $k = 1$:

- Form the empirical covariance matrix $M = \frac{1}{n}\mathbf{X}\mathbf{X}^\top$.
- Find the eigenvector $\mathbf{v}$ corresponding to the largest eigenvalue of M.
- The principal component is $\mathbf{v}$.

Like we mentioned, for any $\mathbf{v}$, the quadratic form $\mathbf{v}^\top \left(\frac{1}{n}\mathbf{X}\mathbf{X}^\top\right) \mathbf{v}$ is the empirical variance of the data projected onto $\mathbf{v}$, i.e. of the data $\{\langle \mathbf{v}, \mathbf{x}_i\rangle\}_{i=1}^{n}$.

The general k-dimensional case In this case, we are looking for a k-dimensional subspace spanned by $\mathbf{v}_1, \dots, \mathbf{v}_k$ such that the aggregate residual error is minimised. We can readily check that

$$\begin{aligned} \arg \min_{\substack{V \in \mathbb{R}^{d\times k} \\ V^\top V = I_k}} \frac{1}{n} \sum_{i=1}^{n} \left\| \mathbf{x}_i - VV^\top \mathbf{x}_i \right\|^2 &= \arg \max_{\substack{V \in \mathbb{R}^{d\times k} \\ V^\top V = I_k}} \frac{1}{n} \sum_{i=1}^{n} \left\| X - VV^\top X \right\|_{\mathsf{F}}^2 \\ &\propto \arg \max_{\substack{V \in \mathbb{R}^{d\times k} \\ V^\top V = I_k}} \mathsf{trace}\left(V^\top \left(\frac{1}{n} XX^\top \right) V \right) \end{aligned}$$

where $\| \cdot \|_{\mathsf{F}}$ is the Frobenius norm. We can now apply the spectral decomposition of the empirical covariance matrix $M = \frac{1}{n}XX^\top$ to find the optimal k-dimensional subspace.

Therefore for any k-dimensional subspace spanned by $\mathbf{v}_1, \ldots, \mathbf{v}_k$, we have

$$\begin{aligned}\mathsf{trace}\left(V^\top \left(\frac{1}{n}XX^\top\right) V\right) = \mathsf{trace}\left(V^\top M V\right) &= \sum_{i=1}^{d} \lambda_i \mathsf{trace}\left(V^\top \mathbf{v}_i \mathbf{v}_i^\top V\right) \\ &= \sum_{i=1}^{d} \lambda_i \mathsf{trace}\left(\mathbf{v}_i^\top V V^\top \mathbf{v}_i\right) = \sum_{i=1}^{d} \lambda_i \mathsf{trace}\left(\mathbf{v}_i^\top \mathbf{v}_i\right) \\ &= \sum_{i=1}^{d} \lambda_i.\end{aligned}$$

Therefore the optimal k-dimensional subspace is spanned by the eigenvectors corresponding to the k largest eigenvalues of the empirical covariance matrix $\frac{1}{n}XX^\top$.

- Form the empirical covariance matrix $M = \frac{1}{n}XX^\top$.
- Find the k eigenvectors $\mathbf{v}_1, \ldots, \mathbf{v}_k$ corresponding to the k largest eigenvalues of M.
- The optimal k-dimensional subspace is spanned by $\mathbf{v}_1, \ldots, \mathbf{v}_k$.

In particular,

$$\sum_{i=1}^{k} \text{empirical variance of } \mathbf{v}_i^\top \mathbf{x} = \mathsf{trace}\left(V^\top \left(\frac{1}{n}XX^\top\right) V\right) = \sum_{i=1}^{k} \lambda_i.$$

(Went through bog-standard handwritten digits example, pretty dull.)

§18 Lecture 18—28th March, 2023

§18.1 Other popular dimensionality reduction techniques

We present a quick overview of some other popular dimensionality reduction techniques.

- *Multidimensional scaling (MDS)*: MDS is a technique that takes a set of pairwise distances between data points and tries to embed them in a lower-dimensional space such that the pairwise distances are preserved as much as possible. MDS can be used for visualization, but it is not as popular as PCA for dimensionality reduction.
- *Independent component analysis (ICA)*: ICA is a technique that tries to find a linear transformation of the data such that the components of the transformed data are statistically independent. ICA is often used in signal processing and is useful when the goal is to separate signals from noise.
- *Non-negative matrix factorization (NMF)*: NMF is a technique that factorizes a non-negative matrix into two non-negative matrices. NMF is often used in topic modeling and image processing.
- *Dictionary learning*: Dictionary learning is a technique that learns a dictionary of atoms that can be used to represent data. In dictionary learning, the goal is to find a dictionary that can represent the data well using a sparse representation. A dictionary is a set of basis vectors that can be used to represent data.
- *Autoencoders*: Autoencoders are a type of neural network that can be used for dimensionality reduction. An autoencoder consists of an encoder and a decoder. The encoder takes the input data and encodes it into a lower-dimensional representation, and the decoder takes the lower-dimensional representation and decodes it back into the original data. Autoencoders are often used for unsupervised learning and can be used for tasks such as denoising and anomaly detection.
- *t-Distributed Stochastic Neighbor Embedding (t-SNE)*: t-SNE is a technique that is often used for visualization. t-SNE takes a high-dimensional dataset and maps it to a lower-dimensional space such that the pairwise distances between data points are preserved as much as possible. t-SNE is often used for visualizing high-dimensional data in two or three dimensions.
- *Uniform Manifold Approximation and Projection (UMAP)*: UMAP is a technique that is similar to t-SNE and is often used for visualization. UMAP takes a high-dimensional dataset and maps it to a lower-dimensional space such that the local structure of the data is preserved. UMAP is often used for visualizing high-dimensional data in two or three dimensions.
- *Locally Linear Embedding (LLE)*: LLE is a technique that tries to preserve the local structure of the data. LLE works by finding a low-dimensional representation of the data such that the local relationships between data points are preserved. LLE is often used for visualization and can be used for tasks such as manifold learning.
- *Isomap*: Isomap is a technique that is often used for dimensionality reduction and visualization. Isomap works by finding a low-dimensional representation of the data such that the geodesic distances between data points are preserved. Isomap is often used for tasks such as manifold learning and can be used for visualizing high-dimensional data in two or three dimensions.

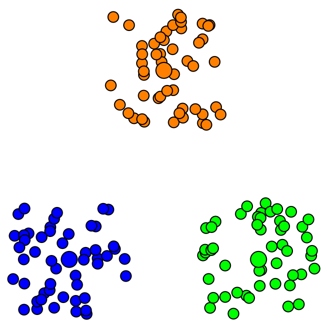
- *Random projection*: Random projection is a technique that works by projecting the data onto a random subspace of lower dimension. Random projection is often used for dimensionality reduction and can be used for tasks such as clustering and classification.

§18.2 Clustering

Clustering is an unsupervised learning technique that is used to group data points into clusters based on their similarity. Given data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathcal{X}$ as well as a target number of clusters k, the goal of clustering is to partition the data points into k clusters such that data points in the same cluster are similar to each other and data points in different clusters are dissimilar to each other. Clustering (also called unsupervised classification, also called quantisation) is often used for tasks such as customer segmentation, anomaly detection, and image segmentation.

There are many different clustering algorithms, each with its own strengths and weaknesses. Some popular clustering algorithms include k-means, hierarchical clustering, DBSCAN, and spectral clustering. In this lecture, we will focus on k-means clustering.

k-means clustering In k-means clustering, we have n data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and a target number of clusters k. The goal of k-means clustering is to find a candidate set of representative points $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k \in \mathbb{R}^d$ called centroids such that each data point is closest to the centroid of its cluster.



The main optimisation problem then for which k-means is a solution is

$$\min_{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k} \left(\sum_{i=1}^{n} \min_{j=1,2,\ldots,k} \|\mathbf{x}_i - \mathbf{c}_j\|^2 \right)$$

This problem is NP-hard, but the k-means algorithm is a simple and efficient heuristic that can be used to find a local minimum of the objective function.

Before we actually present the algorithm, first consider the case where $k = 1$. In this case, the minimum value of the objective function is achieved by setting the derivative of the objective function with respect to $\mathbf{c}$ to zero:

$$\nabla_{\mathbf{c}} \left(\sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{c}\|^2 \right) = 0 \implies 2 \sum_{i=1}^{n} (\mathbf{x}_i - \mathbf{c}) = 0 \implies \mathbf{c} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i.$$

Thus the optimal centroid for the single cluster is the mean of the data points. This motivates the k-means algorithm.

```
Algorithm: k-MEANS CLUSTERING

  • Given: data points $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ and a target number of clusters $k$.
  • Randomly initialise the centroids $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_k \in \mathbb{R}^d$.
  • Repeat until no more changes occur:
      – Assign each data point to the closest centroid.
      – Find the optimal centroid for each cluster by taking the mean of the data points
        in the cluster.
```

This is really Lloyd's algorithm, and it is guaranteed to converge to a local minimum of the objective function. However, the solution may not be unique and can even be arbitrarily bad, and the algorithm may get stuck in a local minimum. The quality of the solution depends on the initialisation of the centroids, and the algorithm may converge to different solutions depending on the initialisation.

How do we select the number of clusters k? One common approach is to use the elbow method, which involves plotting the value of the objective function as a function of k and selecting the value of k at the "elbow" of the curve, which is the point where the rate of decrease of the objective function starts to level off. Another approach is to use the silhouette score, which is a measure of how similar data points are to other data points in the same cluster compared to data points in other clusters. The silhouette score ranges from -1 to 1, with a higher score indicating better clustering.

Hierarchical clustering Hierarchical clustering is a technique that builds a hierarchy of clusters by recursively merging or splitting clusters. There are two main types of hierarchical clustering: agglomerative clustering and divisive clustering. In agglomerative clustering, we start with each data point as a separate cluster and then iteratively merge the closest clusters until we have a single cluster. In divisive clustering, we start with a single cluster containing all the data points and then iteratively split the cluster until we have n clusters, each containing a single data point.

In agglomerative clustering, we need to define a distance metric between clusters. One common distance metric is the average linkage distance, which is the average distance between data points in two clusters:

$$\text{average linkage distance}(C_i, C_j) = \frac{1}{|C_i||C_j|} \sum_{\mathbf{x}_i \in C_i} \sum_{\mathbf{x}_j \in C_j} \|\mathbf{x}_i - \mathbf{x}_j\| .$$

Another common distance metric is the complete linkage distance, which is the maximum distance between data points in two clusters:

$$\text{complete linkage distance}(C_i, C_j) = \max_{\mathbf{x}_i \in C_i, \mathbf{x}_j \in C_j} \|\mathbf{x}_i - \mathbf{x}_j\| .$$

Yet another common distance metric is the single linkage distance, which is the minimum distance between data points in two clusters:

$$\text{single linkage distance}(C_i, C_j) = \min_{\mathbf{x}_i \in C_i, \mathbf{x}_j \in C_j} \|\mathbf{x}_i - \mathbf{x}_j\|.$$

§19 Lecture 19—2nd April, 2023

§19.1 Clustering via probabilistic mixture modelling

Now we present an alternative way to cluster data via probabilistic generative models. Given $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, we want to cluster them into k clusters; it's the typical clustering problem.

Assume that there is a joint probability distribution $(\mathcal{X}, C)$ over the joint space $\mathbb{R}^d \times \{1, \ldots, k\}$. First we generate the cluster assignment C from a categorical distribution with probabilities $\pi_1, \ldots, \pi_k$:

$$C \sim \mathsf{Cat}(\pi_1, \ldots, \pi_k) = \sum_{j=1}^{k} \pi_j \delta_j,$$

where δ_j is the Dirac delta function at j. Then we generate $\mathcal{X}$ from a conditional distribution $p(\mathcal{X} \mid C = j)$ for each j. The joint distribution is

$$p(\mathcal{X}, C) = p(\mathcal{X} \mid C)p(C) = \sum_{j=1}^{k} \pi_j p(\mathcal{X} \mid C = j).$$

The marginal distribution of $\mathcal{X}$ is

$$p(\mathcal{X}) = \sum_{j=1}^{k} \pi_j p(\mathcal{X} \mid C = j).$$

Therefore the marginal distribution of $\mathcal{X}$ is a mixture of k distributions $p(\mathcal{X} \mid C = j)$.

The goal is to estimate the parameters $\pi_1, \ldots, \pi_k$ and the conditional distributions $p(\mathcal{X} \mid C = j)$. The EM algorithm is a popular method to estimate these parameters; but we will discuss it in a particular restricted case where the conditional distributions are Gaussian.

§19.1.1 Gaussian mixture models, the expectation maximisation algorithm

Assume that the conditional distribution $p(\mathcal{X} \mid C = j)$ is a Gaussian distribution with mean μ_j and covariance matrix Σ_j. The parameters are $\theta = \{(\pi_1, \mu_1, \Sigma_1), \ldots, (\pi_k, \mu_k, \Sigma_k)\}$. Then the modelling assumption is that $\mathcal{X}$ is generated by first choosing a cluster $C = j$ with probability π_j and then generating $\mathcal{X}$ from $\mathcal{N}(\mu_j, \Sigma_j)$. However, the cluster assignment C is not observed.

By the law of total probability, the joint distribution of $\mathcal{X}$ given the parameters θ is

$$p(\mathcal{X} \mid \theta) = \sum_{j=1}^{k} p(\mathcal{X}, C = j \mid \theta) = \sum_{j=1}^{k} \pi_j p(C = i \mid \theta) p(\mathcal{X} \mid C = j, \theta) = \sum_{j=1}^{k} \pi_j \mathcal{N}(\mathcal{X} \mid \mu_j, \Sigma_j).$$

Therefore,

$$p(\mathcal{X} \mid \theta) = \sum_{j=1}^{k} \underbrace{\pi_j}_{\text{mixing wt.}} \underbrace{\frac{1}{\sqrt{(2\pi)^d \det(\Sigma_j)}} \exp\left(-\frac{1}{2}(\mathcal{X} - \mu_j)^\top \Sigma_j^{-1} (\mathcal{X} - \mu_j)\right)}_{\text{mixing component}}.$$

Our goal is to estimate the parameters θ given the data $\mathcal{X}_1, \ldots, \mathcal{X}_n$. The likelihood of the data is $p(\mathcal{X}_1, \ldots, \mathcal{X}_n \mid \theta) = \prod_{i=1}^{n} p(\mathcal{X}_i \mid \theta)$, and the log-likelihood is $\log p(\mathcal{X}_1, \ldots, \mathcal{X}_n \mid \theta) = \sum_{i=1}^{n} \log p(\mathcal{X}_i \mid \theta)$. Thus we want to find

$$\begin{aligned}\theta_{\text{MLE}} &= \arg\max_{\theta} \log p(\mathcal{X}_1, \ldots, \mathcal{X}_n \mid \theta) \\ &= \arg\max_{\theta} \sum_{i=1}^{n} \log\left(\sum_{j=1}^{k} \pi_j \frac{1}{\sqrt{(2\pi)^d \det(\Sigma_j)}} \exp\left(-\frac{1}{2}(\mathcal{X}_i - \mu_j)^\top \Sigma_j^{-1} (\mathcal{X}_i - \mu_j)\right)\right).\end{aligned}$$

Unfortunately, this is a non-convex optimization problem that we cannot simplify easily. In fact, the maximum likelihood estimate for the parameters is not tractable to compute in general and is undesirable in general, because it is prone to overfitting.

The EM algorithm is a popular method to estimate the parameters of the Gaussian mixture model. It is an iterative algorithm that alternates between two steps: the E-step and the M-step. In the E-step, we compute the posterior distribution of the cluster assignment given the data and the current estimate of the parameters. In the M-step, we compute the maximum likelihood estimate of the parameters given the data and the current estimate of the cluster assignment. The EM algorithm is guaranteed to converge to a local maximum of the likelihood function.

Algorithm: Expectation maximisation for Gaussian mixture models

- **Input**: Data $\mathcal{X}_1, \ldots, \mathcal{X}_n$, number of clusters k, convergence threshold ϵ.
- **Output**: Parameters $\theta = \{(\pi_1, \mu_1, \Sigma_1), \ldots, (\pi_k, \mu_k, \Sigma_k)\}$.
- **Initialisation**: Randomly initialise the parameters θ.
- **Repeat**:
  - **E-step**: Compute the posterior distribution of the cluster assignment given the data and the current estimate of the parameters:

    $$\gamma_{ij} = \frac{\pi_j \mathcal{N}(\mathcal{X}_i \mid \mu_j, \Sigma_j)}{\sum_{j=1}^{k} \pi_j \mathcal{N}(\mathcal{X}_i \mid \mu_j, \Sigma_j)}.$$

  - **M-step**: Compute the maximum likelihood estimate of the parameters given the data and the current "soft" cluster assignment:

    $$\pi_j = \frac{1}{n}\sum_{i=1}^{n} \gamma_{ij}, \quad \mu_j = \frac{\sum_{i=1}^{n} \gamma_{ij} \mathcal{X}_i}{\sum_{i=1}^{n} \gamma_{ij}}, \quad \Sigma_j = \frac{\sum_{i=1}^{n} \gamma_{ij} (\mathcal{X}_i - \mu_j)(\mathcal{X}_i - \mu_j)^\top}{\sum_{i=1}^{n} \gamma_{ij}}.$$

- **Convergence check**: If the change in the log-likelihood is less than ϵ, then stop.
- **Output**: Parameters θ.

How do we obtain the parameters given in the M-step? We can compute the derivatives of the log-likelihood with respect to the parameters and set them to zero. Given the log-likelihood

$$\log p(\mathcal{X}_1, \dots, \mathcal{X}_n \mid \theta) = \sum_{i=1}^{n} \log \left(\sum_{j=1}^{k} \pi_j \mathcal{N}(\mathcal{X}_i \mid \mu_j, \Sigma_j) \right),$$

we have

$$\frac{\partial}{\partial \pi_j} \log p(\mathcal{X}_1, \dots, \mathcal{X}_n \mid \theta) = \sum_{i=1}^{n} \frac{\mathcal{N}(\mathcal{X}_i \mid \mu_j, \Sigma_j)}{\sum_{j=1}^{k} \pi_j \mathcal{N}(\mathcal{X}_i \mid \mu_j, \Sigma_j)} = \sum_{i=1}^{n} \gamma_{ij},$$

so that when we set this to zero, we obtain the M-step update for π_j as $\pi_j = \frac{1}{n}\sum_{i=1}^{n} \gamma_{ij}$. Similarly, we can compute the M-step updates for μ_j and Σ_j by taking the derivatives of the log-likelihood with respect to μ_j and Σ_j and setting them to zero.

§19.2 Reinforcement learning

We will only present a general overview and will not go into much detail. In reinforcement learning, an agent interacts with an environment by taking actions and receiving rewards. The goal of the agent is to learn a policy that maximises the expected cumulative reward.

§19.2.1 Sequential decision-making problems under uncertainty

For easier understanding of our problem, we first make a comparison between decision making and supervised learning. In supervised learning, we work with a set of features and a target and learn a function mapping from the features to the target. The decision making problem actually builds upon that. In this setting, we not only want the prediction of the target to be correct, but want to learn about the environment as well, so that we can make decisions and interact with the environment. Towards this, we do learn a model about uncertainty and that helps us to decide what to do; upon that, we can make a decision, interact with the environment and observe new data, and continue the iteration again. Eventually, we will end up with a sequence of decisions.

There are four big types of sequential decision-making problems:

Stochastic optimisation Here, the model is known, the state is fixed, and the goal is to find the best action. For example, suppose we are a retail store manager and we need to decide what to put on the shelf for an entire winter season. Let's assume people's demands and preferences are approximately unchanged every year, so that the model is known if we look at the historic data. Also, people's demand is not affected by what we put on the shelf, so there's no impact of the actions on the state. What we want is to decide what to stock in the market and the reward is how much money we make. In this toy scenario, the state is the customer demand, the action is the stock we put on the shelf, and the reward is the money we make. The goal is to find the best action to maximise the reward.

Bandits Here, the model is unknown, the state is fixed, and the goal is to find the best action. Suppose we are designing an online e-portal where we need to decide what contents should be shown to a new customer on the website. Depending on the customer's preference and what we show to him, the customer may or may not continue to engage with our website and we want the customer to engage. We can assume all the customers are from some unknown population, which is the state of the decision making problem. The population will not be affected by the actions we take, thus there's no transition between states. The reward is positive if people continue to engage, and negative vice versa. In this toy scenario, the state is the population of customers, the action is the content we show, and the reward is the engagement of the customer.

Markov decision processes (MDPs) Here, the model is known, the state is dynamic, and the goal is to find the best action. Suppose we are designing an automatic chess player, but only to beat a specific player. This player is well known by us and can predict what he will do in any situations, which corresponds to the known transition kernel in our decision making problem. The board position, which is the state, constantly changes, and the actions is to decide is to play a move of the chess given a state. The reward will be positive if we win and negative if we lose. In this toy scenario, the state is the board position, the action is the move we play, and the reward is the result of the game. We also have a transition kernel that tells us the probability of moving from one state to another given an action.

Reinforcement learning Here, the model is unknown, the state is dynamic, and the goal is to find the best action. Suppose we are designing an autonomous vehicle that needs to navigate through a city. The city is a dynamic environment, and the vehicle needs to learn how to navigate through the city. The state is the current location of the vehicle, the action is the direction the vehicle takes, and the reward is the time taken to reach the destination.

Problems of these sort are frequently encountered in the fields of robotics, control theory, and game playing.

§19.2.2 Markov decision processes

A Markov decision process (MDP) is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $P\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0,1]$ is the transition kernel, $R\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $\gamma \in [0,1]$ is the discount factor that determines the importance of future rewards.

The goal is to find a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximises the expected cumulative reward

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t) \mid S_0 = s\right],$$

where S_t is the state at time t, A_t is the action at time t, and s is the initial state.

The value function of a policy π is

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t) \mid S_0 = s, \pi\right],$$

and the optimal value function is

$$V^*(s) = \max_\pi V^\pi(s).$$

§20 Lecture 20—4th April, 2023

§20.1 Neural networks

Recall that we saw in the exam that the XOR dataset is not linearly separable, so it cannot be learned by PERCEPTRON. One way to circumnavigate this issue is to use a richer model (a quadratic classifier for example) or to lift the data through a feature map ϕ. Both of these approaches are equivalent due to reproducing kernels. A neural network tries to learn the feature map ϕ and the classifier simultaneously.

§20.1.1 Multilayer perceptrons and backpropagation

We can set up the following layers:

- *Input layer*: $\mathbf{x} \in \mathbb{R}^d$.
- *Linear layer:* $\mathbf{z} = \mathbf{U}\mathbf{x} + \mathbf{c}$ for some $\mathbf{U} \in \mathbb{R}^{m \times d}$ and $\mathbf{c} \in \mathbb{R}^m$.
- *Hidden layer:* $\mathbf{h} = \sigma(\mathbf{z})$ for some nonlinear activation function σ.
- *Prediction layer:* $\hat{y} = \langle \mathbf{w}, \mathbf{h} \rangle + b$ for some $\mathbf{w} \in \mathbb{R}^m$ and $b \in \mathbb{R}$.
- *Output layer:* $\operatorname{sign}(\hat{y})$ or $\operatorname{sigmoid}(\hat{y})$.

In total, we need to learn $\mathbf{U}$, $\mathbf{c}$, $\mathbf{w}$, and b.

Example 20.1. The XOR dataset is learnable with a 2-layer neural network (even though it is not learnable by a single-layer perceptron). Let

$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 2 \\ -4 \end{bmatrix}, \quad b = -1,$$

and let $\sigma(t) = \max\{t, 0\}$, i.e. the ReLU activation function. Then the neural network will correctly classify the XOR dataset with $\operatorname{sign}(\langle \sigma(\mathbf{U}\mathbf{x} + \mathbf{c}), \mathbf{w} \rangle + b)$.

To do a multi-class classification, we can simply have a bunch of $\hat{y}$'s in a vector $\hat{y} = \mathbf{W}\mathbf{h} + \mathbf{b}$ where $\mathbf{W} \in \mathbb{R}^{m \times k}$ and $\mathbf{b} \in \mathbb{R}^k$, and make a prediction vector $\hat{\mathbf{p}} = \operatorname{softmax}(\hat{y})$.

Remark 20.2. *The hidden layer σ must be a non-linear function. If it were linear, then the entire network would be equivalent to a single linear layer since the composition of linear functions is linear.*

There are many choices for the activation function σ. Some common choices are:

- $\texttt{relu}(t) = \max\{t, 0\}$.
- $\texttt{gelu}(t) = t \cdot \Phi(t)$ where Φ is the CDF of the standard normal distribution.
- $\texttt{elu}(t) = \max\{t, 0\} + \min\{0, \alpha(e^t - 1)\}$ for some $\alpha > 0$.
- $\texttt{sigmoid}(t) = 1/(1 + e^{-t})$.

- $\texttt{tanh}(t) = \dfrac{e^t - e^{-t}}{e^t + e^{-t}} = 1 - 2 \cdot \texttt{sigmoid}(t)$.

We can also stack several layers together, repeating the pattern of linear layer followed by a non-linear activation function. This is called a *deep neural network*.

To train, we need a loss function ℓ and a data set $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$.

Notation 20.3. Write $[\ell \circ f](\mathbf{x}, y)$ to mean $\ell(f(\mathbf{x}, \mathbf{w}), y)$.

We can express the neural network as a minimisation problem:

$$\min_{\mathbf{w} \in \mathbb{R}^n} \frac{1}{n} \sum_{i=1}^n [\ell \circ f](\mathbf{x}_i, y_i, \mathbf{w}),$$

which gives the gradient descent update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left(\frac{1}{n} \sum_{i=1}^n [\ell \circ f](\mathbf{x}_i, y_i, \mathbf{w}) \right),$$

where η is the learning rate. This requires a full pass through the data set for each update, which can be slow. Instead of doing ordinary gradient descent, we can *minibatch* by picking a random subset $B \subseteq [n]$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \left(\frac{1}{|B|} \sum_{i \in B} [\ell \circ f](\mathbf{x}_i, y_i, \mathbf{w}) \right),$$

which trades off the accuracy of the gradient for speed. We can also use *stochastic gradient descent* where $|B| = 1$.

The learning rate has diminishing returns. Instead of keeping a constant η, we can parameterise η_t and say something like

$$\eta_t = \begin{cases} \eta_0 & \text{if } t \le t_0, \\ \eta_0/10 & \text{if } t_0 < t \le t_1, \\ \eta_0/100 & \text{if } t_1 < t \end{cases}$$

for an initial η_0 and specific epochs t_0 and t_1. Alternatively we can use *sublinear decay* $\eta_t = \eta_0/(1+ct)$ or $\eta_t = \eta_0/\sqrt{1+ct}$ for some $c > 0$ constant.

We will often need to calculate a lot of partial derivatives with respect to matrices.

Definition 20.4. *Let* $y(\mathbf{X}) \in \mathbb{R}$ *and* $\mathbf{X} \in \mathbb{R}^{m \times n}$. *Then the* partial derivative of y with respect to $\mathbf{X}$ *is the matrix* $\nabla_{\mathbf{X}} y \in \mathbb{R}^{m \times n}$ *where*

$$\nabla_{\mathbf{X}} y = \left[\frac{\partial y}{\partial \mathbf{X}_{ij}} \right]_{i,j} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \cdots & \frac{\partial y}{\partial X_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial X_{m1}} & \cdots & \frac{\partial y}{\partial X_{mn}} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

The best way to do this is to just "guess" analogously to the scalar case, and then check that the dimension is right, bearing in mind that $\dim \nabla_{\mathbf{X}} y = \dim \mathbf{X}$.

Consider the forward pass for a neural network with width k and output dimension c:

$$\begin{aligned}
\mathbf{x} &= \text{ input} && \mathbf{x} \in \mathbb{R}^d \\
\mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} && \mathbf{W} \in \mathbb{R}^{k\times d};\ \mathbf{z}, \mathbf{b} \in \mathbb{R}^k \\
\mathbf{h} &= \sigma(\mathbf{z}) && \mathbf{h} \in \mathbb{R}^k \\
\boldsymbol{\theta} &= \mathbf{U}\mathbf{h} + \mathbf{c} && \mathbf{U} \in \mathbb{R}^{c\times k};\ \boldsymbol{\theta}, \mathbf{c} \in \mathbb{R}^c \\
J &= \frac{1}{2}\|\boldsymbol{\theta} - \mathbf{y}\|^2 && \mathbf{y} \in \mathbb{R}^c.
\end{aligned}$$

Now we can just apply the chain rule to get the gradients:

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{\theta}} &= \boldsymbol{\theta} - \mathbf{y} \\
\frac{\partial J}{\partial \mathbf{c}} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \cdot \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{U}} = \underbrace{(\boldsymbol{\theta} - \mathbf{y})}_{\in\mathbb{R}^{c\times 1}} \cdot \underbrace{\mathbf{h}^\top}_{\in\mathbb{R}^{1\times k}} && \text{to get the right dimensions } c \times k \\
\frac{\partial J}{\partial \mathbf{c}} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \cdot \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{c}} = \underbrace{(\boldsymbol{\theta} - \mathbf{y})}_{\in\mathbb{R}^{c\times 1}} && \text{since } \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{b}} = \mathbf{I} \\
\frac{\partial J}{\partial \mathbf{U}} &= \frac{\partial J}{\partial \boldsymbol{\theta}} \cdot \frac{\partial \boldsymbol{\theta}}{\partial \mathbf{h}} = \underbrace{\mathbf{U}^\top}_{\in\mathbb{R}^{k\times c}} \cdot \underbrace{(\boldsymbol{\theta} - \mathbf{y})}_{\in\mathbb{R}^{c\times 1}} && \text{to get the right dimensions } k \times 1 \\
\frac{\partial J}{\partial \mathbf{z}} &= \frac{\partial J}{\partial \mathbf{h}} \cdot \frac{\partial \mathbf{h}}{\partial \mathbf{z}} = \underbrace{\mathbf{U}^\top (\boldsymbol{\theta} - \mathbf{y})}_{\in\mathbb{R}^{k\times 1}} \odot \underbrace{\sigma'(\mathbf{z})}_{\in\mathbb{R}^{k\times 1}} && \text{using } \odot \text{ for Hadamard product} \\
\frac{\partial J}{\partial \mathbf{W}} &= \frac{\partial J}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{W}} = \underbrace{(\mathbf{U}^\top (\boldsymbol{\theta} - \mathbf{y}) \odot \sigma'(\mathbf{z}))}_{\in\mathbb{R}^{k\times 1}} \cdot \underbrace{\mathbf{x}^\top}_{\in\mathbb{R}^{1\times d}} && \text{to get the right dimensions } k \times d \\
\frac{\partial J}{\partial \mathbf{b}} &= \frac{\partial J}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{b}} = \underbrace{(\mathbf{U}^\top (\boldsymbol{\theta} - \mathbf{y}) \odot \sigma'(\mathbf{z}))}_{\in\mathbb{R}^{k\times 1}}.
\end{aligned}$$

This is the *backpropagation* algorithm. Existing libraries like `PyTorch` and `TensorFlow` will do this for us.

Theorem 20.1 (Universal approximation theorem for neural networks)**.** *For any continuous function $f\colon \mathbb{R}^d \to \mathbb{R}^c$ and any $\varepsilon > 0$, there exists $k \in \mathbb{N}$, $\mathbf{W} \in \mathbb{R}^{k\times d}$, $\mathbf{b} \in \mathbb{R}^k$, and $\mathbf{U} \in \mathbb{R}^{c\times k}$ such that*

$$\sup_{\mathbf{x}\in[0,1]^d} \|f(\mathbf{x}) - g(\mathbf{x})\|_2 < \varepsilon,$$

where $g(\mathbf{x}) = \mathbf{U}\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$ and σ is the elementwise ReLU activation function.

Informally, a 2-layer neural network can approximate any continuous function to arbitrary precision, provided that the network is wide enough with a large enough number of hidden units or neurons.

However, these networks are not very efficient. In the worst case, a 2-layer multi-layer perceptron (MLP) needs $k = \exp(1/\varepsilon)$ but a 3-layer MLP can get away with $k = \text{poly}(1/\varepsilon)$. Deeper networks will have even smaller dimensionality requirements.

To avoid overfitting, we can apply *dropout*. That is, for each minibatch, we randoml select some hidden neurons to be active with probability q (and pretend the rest of them don't exist). Then, each training minibatch gets a "different" network, and so it's harder for neurons to "collude" to get overfitting. To make sure that dropout does not affect the overall expectation, multiply each $\mathbf{h}$ by $1/q$ during the backpropagation.

We can also do batch normalisation. This is a technique to make the input to each layer have mean 0 and variance 1, to help with the *vanishing gradient problem*, a phenomenon where the gradient becomes very small as it is backpropagated through the network.

§20.1.2 Convolutional neural networks

An MLP has a lot of parameters to learn. Instead of densely connecting every node in the input layer to the hidden layer, only connect some of them (i.e., make $\mathbf{W}$ sparse). Also, to reduce the number of parameters even more, make a bunch of the weights the same. Following a certain pattern, we get a convolution. These are useful for image processing/classification/segmentation but not for NLP.

The layers of CNN are roughly:

- *feature extraction*: a series of convolutions and ReLUs, where we use a sliding window to reduce the dimensions of the input while *pooling* inputs together to increase width to make up for decreased size;
- *vectorization*: convert the matrix into a vector;
- classification: a fully connected layer (i.e., MLP);
- probabilistic distribution: a softmax activation function.

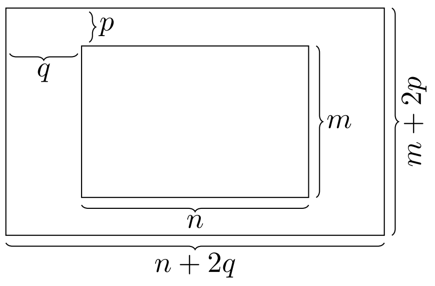
To process an image, split into sepraate channels for RGB values, then treat as a matrix of values. We will learn a *kernel* for the convolution with stochastic gradient descent. Convolutions have been shown to represent human visual cognition. Traditional image processing also uses convolutions. For example, edge detection and Gaussian smoothing.

For multi-channel input, "stack" the channels and use a "cube" (tensor) kernel. We can also apply a bias term $b \in \mathbb{R}$ to the output tensor (add b to every element).

In a CNN layer, we increase channels to account for decreased resolution. For example, with 3 RGB input channels, we might learn 5 different $3 \times 3 \times 3$ kernels. Then, we will end up with 5 output channels.

We can also control the size of the step taken during convolution. Instead of always moving 1-left and 1-down, we can have a larger *stride*. However, we want overlap between windows, so always make sure that the stride is less than the kernel size. We can also control the *padding*, adding 0s as necessary to keep boundary information.

Suppose we have input size $\overbrace{m \times n}^{\text{typical } m=n=224} \times c_{in}$, kernel size $\overbrace{a \times b}^{\text{typical } a=b=5} \times c_{in}$, stride $\overbrace{s \times t}^{\text{typical } s=t=1,2}$, and padding $\overbrace{p \times q}^{\text{typical } p=q}$ so that the preprocesssed input looks like



Then, the output size will be $\left\lfloor 1+\frac{m+2p-a}{s}\right\rfloor \times \left\lfloor 1+\frac{n+2q-b}{t}\right\rfloor$. If we want the input and output to have the "same" size, set $p=\left\lceil \frac{m(s-1)+a-s}{2}\right\rceil$ and $q=\left\lceil \frac{n(t-1)+b-t}{2}\right\rceil$.

§21 Lecture 21—9th April, 2023

§21.1 More on convolutional neural networks

Recall the convolution of $\mathbf{X} = \begin{bmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{bmatrix}$ and $\mathbf{W} = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \end{bmatrix}$ is

$$\mathbf{W} * \mathbf{X} = \begin{bmatrix} w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} & w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} \\ w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21} & w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22} \end{bmatrix}$$

such that the vectorisation is

$$\text{Vector}(\mathbf{W} * \mathbf{X}) = \begin{bmatrix} w_{00}x_{00} + w_{01}x_{01} + w_{10}x_{10} + w_{11}x_{11} \\ w_{00}x_{01} + w_{01}x_{02} + w_{10}x_{11} + w_{11}x_{12} \\ w_{00}x_{10} + w_{01}x_{11} + w_{10}x_{20} + w_{11}x_{21} \\ w_{00}x_{11} + w_{01}x_{12} + w_{10}x_{21} + w_{11}x_{22} \end{bmatrix}.$$

This is a linear transformation. Therefore, we can design a *circulant matrix* $\mathbf{W}_{\mathsf{circ}}$ such that $\mathbf{W}_{\mathsf{circ}}\text{Vector}(\mathbf{X}) = \text{Vector}(\mathbf{W} * \mathbf{X})$. Define

$$\mathbf{W}_{\mathsf{circ}} = \begin{bmatrix} w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 & 0 \\ 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 & 0 & 0 \\ 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} & 0 \\ 0 & 0 & 0 & 0 & w_{00} & w_{01} & 0 & w_{10} & w_{11} \end{bmatrix}$$

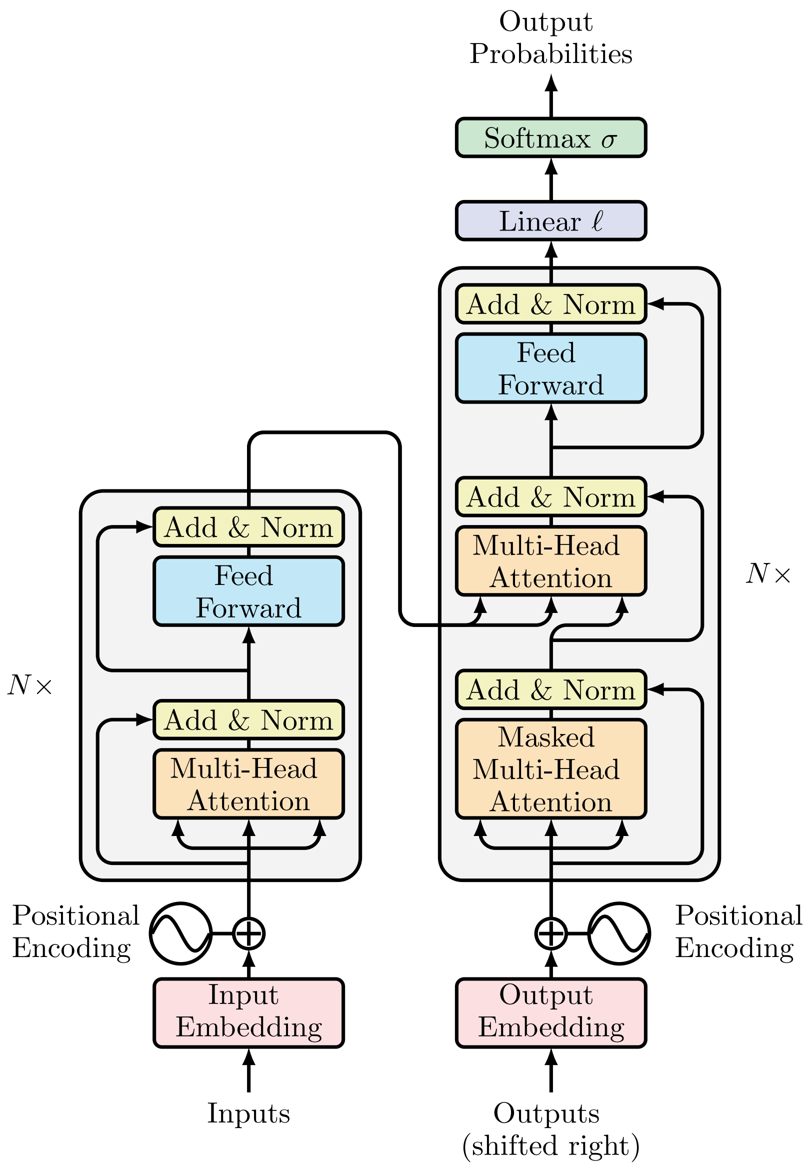
and it is clear that $\mathbf{W}_{\mathsf{circ}}\text{Vector}(\mathbf{X}) = \text{Vector}(\mathbf{W} * \mathbf{X})$. Now, notice that we only need to learn $\mathbf{W} = 4$ weights instead of $|\mathbf{W}_{\mathsf{circ}}| = 9 \times 4 = 36$ weights. We can also down-sample the input size using *pooling*. Just like convolution, we take a sliding window with some fixed size and stride and apply a transformation. Instead of the inner product, we can do *max-pooling* (take the max of the window) or *average-pooling* (take the mean of the window). *Global pooling* is where the window is the whole input, so we output a single scalar.

LeNet Given an input of size 32^2, convolve with six 5^2 kernels to 6 @ 28^2, subsample down by half to 6 @ 14^2, convolve with sixteen 5^2 kernels to 16 @ 10^2, subsample down by half to 16 @ 5^5, fully connect to a 120-wide layer, fully connect to an 84-wide layer. Gaussian connect to a 10-wide output, which is the number of classes. Then apply softmax.

AlexNet Given an input of size 3 @ 224×224, convolve with 96 kernels to 96 @ 55×55, and apply max-pooling to 96 @ 27×27. Then convolve with 256 kernels to 256 @ 27×27 and apply max-pooling to 256 @ 13×13. Then convolve with 384 kernels to 384 @ 13×13, convolve with 384 kernels to 384 @ 13×13, and convolve with 256 kernels to 256 @ 13×13. Then apply max-pooling to 256 @ 6×6. Then fully connect to a 4096-wide layer, fully connect to a 4096-wide layer, and fully connect to a 1000-wide layer. Then apply softmax.

§21.2 Transformers

The transformer is a neural network architecture that is based on self-attention.



Our goal is given a sequence of tokens (the prompt) $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, to find a sequence $Y = (\mathbf{y}_1, \ldots, \mathbf{y}_m)$ such that the maximum likelihood

$$\arg\max_{Y} p(\mathbf{y}_1, \ldots, \mathbf{y}_m \mid \mathbf{x}_1, \ldots, \mathbf{x}_n)$$

is achieved. We use an *auto-regressive* model where we greedily take

$$\arg\max_{\mathbf{y}_k} p(\mathbf{y}_k \mid \mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{y}_1, \ldots, \mathbf{y}_{k-1})$$

i.e., one token at a time. Note that m is not pre-defined; we keep generating until we reach the [END] token.

At each step, we input the embeddings of the prompt and the partially generated text. The text is converted to tokens, which are the smallest elements the model can understand. Then, the tokens are embedded in a high-dimensional vector space (typically, $d = 512$). The embedding should map similar words to similar locations.

The output of the prompt embedding is $X = [\mathbf{x}_1, \ldots, \mathbf{x}_n] \in \mathbb{R}^{n \times d}$ and the auto-regressive outputs $[\mathbf{y}_1, \ldots, \mathbf{y}_k] \in \mathbb{R}^{k \times d}$

Since word order matters, we also add a **positional encoding**. We define the matrix $W^p \in \mathbb{R}^{n \times d}$ as

$$W^p_{t,2i} = \sin(t/10000^{2i/d}), \quad W^p_{t,2i+1} = \cos(t/10000^{2i/d}), \quad i = 0, \ldots, \tfrac{d}{2} - 1$$

This is a fixed part of the model, and we simply add W^p to X. The auto-regressive output is also similarly positionally encoded.

These are then sent to attention layers.

The Attention function has an input value $V \in \mathbb{R}^{n \times d}$, a key $K \in \mathbb{R}^{n \times d}$, and a query $Q \in \mathbb{R}^{m \times d}$. It outputs an $\mathbb{R}^{m \times d}$ matrix.

Recall the softmax function as applied to vectors:

$$\operatorname{softmax}(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_i \exp(z_i)}, \ldots, \frac{\exp(z_n)}{\sum_i \exp(z_n)}\right]$$

Then, writing $\mathbf{v}_i^\top$, $\mathbf{k}_i^\top$, and $\mathbf{q}_i^\top$ as the rows of V, K, and Q:

$$\begin{aligned}
\operatorname{Attention}(V, K, Q) &= \operatorname{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V \\
&= \begin{bmatrix} \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_1 \rangle}{\sqrt{d}}\right) & \cdots & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_n \rangle}{\sqrt{d}}\right) \\ \vdots & \ddots & \vdots \\ \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_1 \rangle}{\sqrt{d}}\right) & \cdots & \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_n \rangle}{\sqrt{d}}\right) \end{bmatrix} V \\
&= \begin{bmatrix} \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_1 \rangle}{\sqrt{d}}\right) \mathbf{v}_1^\top + \cdots + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_1, \mathbf{k}_n \rangle}{\sqrt{d}}\right) \mathbf{v}_n^\top \\ \vdots \\ \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_1 \rangle}{\sqrt{d}}\right) \mathbf{v}_1^\top + \cdots + \operatorname{softmax}\left(\frac{\langle \mathbf{q}_m, \mathbf{k}_n \rangle}{\sqrt{d}}\right) \mathbf{v}_n^\top \end{bmatrix} \in \mathbb{R}^{m \times d}.
\end{aligned}$$

Each output “value” (i.e., row) here is a convex combination of the rows of V.

In the self-attention case, $Q = K = V =$ whatever the input is.

So far, there are no learnable parameters. To add learnable parameters, we do a linear layer with each of V, K, and Q. That is, $W_i^q, W_i^k, W_i^v \in \mathbb{R}^{512 \times 64}$ can be learnable linear layers such that $\operatorname{Attention}_i = \operatorname{softmax}\left(\frac{QW_i^q (KW_i^k)^\top}{\sqrt{d}}\right) VW_i^v$.

Each of these $i = 1, \ldots, h$ triplets is called a head. Typically $h = 8$. A multi-head attention layer concatenates each $\operatorname{Attention}_i$ and sends that through a final learnable linear layer.

A masked attention layer just ignores future tokens $\mathbf{y}_k, \ldots, \mathbf{y}_m$ during training.

The feed forward layers are just two-layer MLPs with ReLU activation:

$$\max(0, \mathbf{x}^\top W_1 + \mathbf{b}_1) W_2 + \mathbf{b}_2$$

where $W_1 \in \mathbb{R}^{d \times 4d}$ and $W_2 \in \mathbb{R}^{4d \times d}$. They also have residual connections and layer normalisation.

Summary There are three tunable hyperparameters: layers $N = 6$, output dimensions $d = 512$, and heads $h = 8$.

The cross-attention module has $V = K =$ encoder and $Q =$ decoder. The other attention modules are self-attentive, so $V = K = Q$.

We train by minimising the log-loss between true next words and predicted next words

$$\min_W \hat{\mathbb{E}} \left[- \left\langle Y, \log \hat{Y} \right\rangle \right]$$

where $Y = [\mathbf{y}_1, \dots, \mathbf{y}_l]$ is the one-hot output sequence and $\hat{Y} = [\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_l]$ are the predicted probabilities.

§22 Lecture 22—11th April, 2023

§22.1 Large language models

Generative Pre-Training (GPT-1)

GPT-1 is an open-source 12-layer decoder with 110M parameters. It is pre-trained unsupervised on next-word prediction. Then, fine-tuning is done on task-dependent architecture, i.e., there are specific architectures for different tasks.

Bidirectional Encoder Representations from Transformers (BERT)

BERT is an encoder-only model. It also has a pre-training phase and fine-tuning phase.

In the pre-training phase, the encoder is given masked sentences and is trained to generate the missing tokens (Masked LM; task A). Training on task A performs better than training on the left-to-right prediction task. The model is also trained on next-sentence prediction (NSP; task B), where it binary classifies whether two sentences follow or are unrelated.

$\text{BERT}_{\text{BASE}}$ has a similar number of parameters (110M) to GPT-1, but performs better. $\text{BERT}_{\text{LARGE}}$ (340M) performs better than both.

RoBERTa (Robustly Optimised BERT Approach) is just a larger BERT model with bigger batches and more data. It was also only trained on the Masked LM objective, but with longer sequences.

Sentence-BERT/RoBERTa trains the similarity task using two encoders (one for each sentence) and saves represented encodings. This saves a lot of inference time.

GPT-2 through 4

Basically the only thing done is make the model larger.

GPT-2 introduced a new dataset called WebText. The 1.5B-parameter model is around $10\times$ larger than GPT-1, and is trained in the same way. It is about on par with BERT on finetuning tasks. It is also the most recent OpenAI model to be open-sourced. However, it is very good at zero-shot learning. This means we no longer need task-dependent architecture.

GPT-3 is trained exactly the same as GPT and GPT-2, but 100× larger (175B parameters). At around the 100B-parameter level, we start to see emergent properties of in-context learning (zero-/few-shot prompts) and chain-of-thought (either one-shot or "let's do this step-by-step"). However, raw language models do not answer questions or behave in a chat-like way. For example, asking a quesiton to GPT-3 will result in a list of similar questions.

GPT-3.5 (InstructGPT) uses Reinforcement Learning from Human Feedback (RLHF). In RLHF, the agent uses a policy function (LLM) to take actions (outputs) given a state (prompt), and is returned a reward and new state based on the environment (another LLM):

1. Collect demonstration data, and train a supervised policy: train by overfitting GPT-3 to human-written desired outputs (the SFT model).
2. Collect comparison data, and train a reward model: train a new reward model (RM) using human rankings of outputs. We use pair-wise comparison logistic loss

   $$\text{loss}(\theta) = -\mathbb{E}_{(x,y_w,y_l)}[\log(\sigma(r_\theta(x, y_w) - r(x, y_l)))]$$

   for a prompt x and preferred output $r_\theta(x, y_w) \gg r_\theta(x, y_l)$. This trains a real-valued function r_θ so ChatGPT knows how much better y_w is than y_l without the unknown human element.
3. Optimise a policy against the reward model using reinforcement learning: update the SFT model using the RM model using proximal policy optimisation (PPO):

   $$\max_\phi \mathbb{E}_{(x,y)}[\underbrace{r_\theta(x, y)}_{\text{RM reward}} - \beta \underbrace{\log(\pi_\phi^{\text{RL}}(y \mid x)/\pi^{\text{SFT}(y|x)})}_{\text{model is close to SFT}}] + \gamma \underbrace{\mathbb{E}[\log(\pi_\phi^{\text{RL}}(x))]}_{\text{pretraining loss}}$$

In general, GPT $\ll$ prompted GPT $\ll$ SFT $\ll$ PPO $<$ PPO with pretraining mix. PPO-ptx is the base model for ChatGPT-3.5 and GitHub Copilot.

GPT-4 allows combined multimodal image/text input. The paper says nothing so nobody knows how it works.

§22.2 Generative adversarial networks

Suppose we are given training data $\{\mathbf{x}_i\} \sim q(\mathbf{x})$, i.e., with *data density* $q(\mathbf{x})$. Recall that $q(\mathbf{x})$ is a distribution if $\int_{-\infty}^{\infty} q(\mathbf{x})\,\mathrm{d}\mathbf{x} = 1$ and $q(\mathbf{x}) \geq 0$.

We will develop a *model density* $p_{\boldsymbol{\theta}}(\mathbf{x})$ parameterised by $\boldsymbol{\theta}$. We will find $\boldsymbol{\theta}$ by minimising some "distance" between q and $p_{\boldsymbol{\theta}}$. In particular, we will minimise the KL divergence

$$\begin{aligned} \text{KL}(q \parallel p_{\boldsymbol{\theta}}) &\coloneqq \int q(\mathbf{x}) \log \frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} \equiv \int -\log p_{\boldsymbol{\theta}}(\mathbf{x}) \cdot q(\mathbf{x})\,\mathrm{d}\mathbf{x} = \mathbb{E}_{\mathbf{x}\sim q(\mathbf{x})}[-\log p_{\boldsymbol{\theta}}(\mathbf{x})] \\ &\approx -\frac{1}{n}\sum_{i=1}^{n} \log p_{\boldsymbol{\theta}}(\mathbf{x}_i) \end{aligned}$$

Then, we will use $p_{\boldsymbol{\theta}}(\mathbf{x})$ to generate new data $X \sim p_{\boldsymbol{\theta}}(\mathbf{x})$.

However, we do not have a closed-form way to calculate $p_{\boldsymbol{\theta}}$. Suppose that we want $p(\mathbf{x})$ to be a d-variate Gaussian with means $\boldsymbol{\mu} \in \mathbb{R}^d$ and covariance matrix $S \in \mathbb{R}^{d\times d}$:

$$p(\mathbf{x}) = (2\pi)^{d/2}[\det(S)]^{-1/2}\exp[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top S^{-1}(\mathbf{x}-\boldsymbol{\mu})]$$

To draw from $p(\mathbf{x})$, start by drawing $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathrm{id})$. Then, we write $\mathbf{x} = L\mathbf{n} + \boldsymbol{\mu}$ where $LL^\top = S$ (the *Cholesky decompostion* of S), so that we have

$$\begin{aligned}\mathbb{E}[\mathbf{x}] &= E[L\mathbf{n} + \boldsymbol{\mu}] = \boldsymbol{\mu}\\ \mathbb{E}[(\mathbf{x}-\boldsymbol{\mu})(\mathbf{x}-\boldsymbol{\mu})^\top] &= L\cdot\mathbb{E}[\mathbf{n}\mathbf{n}^\top]\cdot L^\top = LL^\top = S\end{aligned}$$

and $p(\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, S)$, as desired.

We will simply replace the function $L\mathbf{n} + \boldsymbol{\mu}$ with a neural network.

Theorem 22.1 (Representation through push-forward)**.** *Let r be any continuous distribution on $\mathbb{R}^h$. Then, for any distribution p on $\mathbb{R}^d$, there exist* push-forward maps $\mathbf{T}\colon \mathbb{R}^h \rightrightarrows \mathbb{R}^d$ *such that*

$$Z \sim r \implies \mathbf{T}(Z) \sim p$$

This does not hold if r has a delta mass at any point. Without loss of generality, we take r to be standard Gaussian noise.

We will learn $\mathbf{T}$ using a neural network. In general, $\mathbf{T}$ is not unique, so we can optionally add restrictions to force uniqueness. It can be a really weird set of mappings if $h \ll d$.

Now, we are able to generate new data $X \sim p_{\boldsymbol{\theta}} = \mathbf{T}_{\boldsymbol{\theta}}(Z)$ where $Z \sim \mathcal{N}(\mathbf{0}, \mathrm{id})$. This means that we can easily *draw from $p_{\boldsymbol{\theta}}$* but we cannot *write the density function.*

We're stuck in a catch-22: we need the density to find the loss and train, and we need the final trained $\mathbf{T}$ to draw. Consider again the KL divergence:

$$\mathrm{KL}(q \parallel p_{\boldsymbol{\theta}}) = \int \log\frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})}\,\mathrm{d}\mathbf{x} \equiv \int \underbrace{\frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})}\left[\log\frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} - 1\right]}_{f(\frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})})} \cdot p_{\boldsymbol{\theta}}(\mathbf{x})\,\mathrm{d}\mathbf{x}$$

which we have rewritten as a function $f\colon \mathbb{R}_+ \to \mathbb{R}$, $f(t) = t\log t - t + 1$ of the ratio.

This is convex, because $\frac{\mathrm{d}^2}{\mathrm{d}t^2} f(t) = \frac{\mathrm{d}}{\mathrm{d}t}\log t = \frac{1}{t} > 0$.

Definition 22.1 (Fenchel conjugate)**.** *The conjugate of any function f is the convex function $f^*(s) := \max_t st - f(t)$.*

Then, if f is convex and continuous, then $f = f^{**}$.

Since our f is convex and continuous, let's try writing f as f^{**}. First,

$$\begin{aligned}f^*(s) &= \max_t st - f(t)\\ &= \max_t\{st - t\log t + t - 1\}\end{aligned}$$

Then, setting the derivative to 0, we get $s = \log t$, i.e., $t = \exp s$, so

$$f^*(s) = \exp s - 1$$

This gives us

$$f(t) = \max_s\{st - f^*(s)\} = \max_s\{st - \exp s + 1\}$$

Revisiting the above, we can replace f by f^{**}:

$$\begin{aligned}
\mathrm{KL}(q \parallel p_\theta) &\equiv \int \left[f\left(\frac{q(\mathbf{x})}{p_\theta(\mathbf{x})}\right) - 1\right] \cdot p_\theta(\mathbf{x})\,\mathrm{d}\mathbf{x} \\
&= \int \left[\max_s s\frac{q(\mathbf{x})}{p_\theta(\mathbf{x})} - \exp s\right] \cdot p_\theta(\mathbf{x})\,\mathrm{d}\mathbf{x} \\
&= \int \left[\max_s s \cdot q(\mathbf{x}) - \exp(s) p_\theta(\mathbf{x})\right]\,\mathrm{d}\mathbf{x} && p_\theta(\mathbf{x}) \text{ has no } s\text{-dependence} \\
&= \max_{S\colon \mathbb{R}^d \to \mathbb{R}} \int S(\mathbf{x}) q(\mathbf{x}) - \exp(S(\mathbf{x})) p_\theta(\mathbf{x})\,\mathrm{d}\mathbf{x} && s \text{ is parameterised by } x \\
&\approx \max_{S\colon \mathbb{R}^d \to \mathbb{R}} \frac{1}{n}\sum_{i=1}^n S(\mathbf{x}_i) - \frac{1}{m}\sum_{j=1}^m \exp\left(S(\mathbf{T}_\theta(\mathbf{z}_j))\right)
\end{aligned}$$

where $\mathbf{x}_i \sim q(\mathbf{x})$ and $\mathbf{z}_j \sim \mathcal{N}(\mathbf{0}, \mathrm{id})$. We write this as one line:

$$\min_{\boldsymbol{\theta}} \mathrm{KL}(q \parallel p_{\boldsymbol{\theta}}) \approx \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{n}\sum_{i=1}^n S_{\boldsymbol{\phi}}(\mathbf{x}_i) - \frac{1}{m}\sum_{j=1}^m \exp[S_{\boldsymbol{\phi}}(T_{\boldsymbol{\theta}}(\mathbf{z}_j))]$$

for a *generator* $\mathbf{T}_{\boldsymbol{\theta}}$ which maps latent noise $\mathbf{z}$ to observation $\mathbf{x}$, and a *discriminator* $S_{\boldsymbol{\phi}}$ which distinguishes data $\mathbf{x}$ from generation $\mathbf{T}_{\boldsymbol{\theta}}(\mathbf{z})$. Both are neural networks paramaterized by weights $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, respectively. This ends up being a minimax game between the generator and discriminator. At the equilibrium, the generator can make data and the discriminator cannot distinguish.

In reality, we do not use KL divergence, but instead the JS divergence:

$$\begin{aligned}
\mathrm{JS}(q \parallel p_{\boldsymbol{\theta}}) &\coloneqq \mathrm{KL}(q \parallel \tfrac{q+p_{\boldsymbol{\theta}}}{2}) + \mathrm{KL}(q \parallel \tfrac{q+p_{\boldsymbol{\theta}}}{2}) \\
&= \int q(\mathbf{x}) \log \frac{2q(\mathbf{x})}{q(\mathbf{x}) + p_{\boldsymbol{\theta}}(\mathbf{x})} + p_{\boldsymbol{\theta}}(\mathbf{x}) \log \frac{2p_{\boldsymbol{\theta}}(\mathbf{x})}{q(\mathbf{x}) + p_{\boldsymbol{\theta}}(\mathbf{x})}\,\mathrm{d}\mathbf{x} \\
&= \int \left[\frac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} \log \frac{q(\mathbf{x})/p_{\boldsymbol{\theta}}(\mathbf{x})}{q(\mathbf{x})/p_{\boldsymbol{\theta}}(\mathbf{x}) + 1} + \log \frac{1}{q(\mathbf{x})/p_{\boldsymbol{\theta}}(\mathbf{x}) + 1} + \log 4\right] \cdot p_{\boldsymbol{\theta}}(\mathbf{x})\,\mathrm{d}\mathbf{x} \\
&= \int f\left(\tfrac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})}\right) \cdot p_{\boldsymbol{\theta}}(\mathbf{x})\,\mathrm{d}\mathbf{x}
\end{aligned}$$

where $f(t) = t\log t - (t+1)\log(t+1) + \log 4$ is convex. Then, $f^*(t) = -\log(1 - \exp s) - \log 4$. If we do the same transformation, we can approximate

$$\min_{\boldsymbol{\theta}} \mathrm{JS}(q \parallel p_{\boldsymbol{\theta}}) \approx \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{n}\sum_{i=1}^n S_{\boldsymbol{\phi}}(\mathbf{x}_i) - \frac{1}{m}\sum_{j=1}^m \log[1 - \exp S_{\boldsymbol{\phi}}(T_{\boldsymbol{\theta}}(\mathbf{z}_j))] - \log 4$$

Why do we use the JS divergence? After the same transformations, we can apply the change of variable $S_{\boldsymbol{\phi}} \leftarrow \log S_{\boldsymbol{\phi}}$:

$$\min_{\boldsymbol{\theta}} \mathrm{JS}(q \parallel p_{\boldsymbol{\theta}}) \approx \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{n}\sum_{i=1}^n \log S_{\boldsymbol{\phi}}(\mathbf{x}_i) - \frac{1}{m}\sum_{j=1}^m \log[1 - S_{\boldsymbol{\phi}}(T_{\boldsymbol{\theta}}(\mathbf{z}_j))]$$

Let $\mathbf{y}(\mathbf{x}) = [\mathbf{x} \text{ is real data}]$. Let $\mathfrak{p}_1(\mathbf{x}) = S_\phi(\mathbf{x})$ be the (learnable) probability of $\mathbf{x}$ being real, and $\mathfrak{p}_0 = 1 - \mathfrak{p}_1$. Then, we have $\min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \hat{\mathbb{E}}_X \log \mathfrak{p}_{\mathbf{y}(X)}$ which is just the logistic regression, which we know well.

In fact, we can prove that if we pick any strictly convex function $f\colon \mathbb{R}_+ \to \mathbb{R}$ with normalization $f(1) = 0$, then we can define an f-divergence $\mathbb{D}_f$ such that $\mathbb{D}_f(q \parallel p) \geq 0$ iff $p = q$ and we can maximize

$$\begin{aligned}\min_{\boldsymbol{\theta}} \mathbb{D}_f(q \parallel p) &:= \min_{\boldsymbol{\theta}} \int f(\tfrac{q(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})}) p_{\boldsymbol{\theta}}(\mathbf{x}) \,\mathrm{d}\mathbf{x} \\ &\approx \min_{\boldsymbol{\theta}} \max_{\boldsymbol{\phi}} \frac{1}{n} \sum_{n=1}^{n} S_{\boldsymbol{\phi}}(\mathbf{x}_i) - \frac{1}{m} \sum_{j=1}^{m} f^*[S_{\boldsymbol{\phi}}(T_{\boldsymbol{\theta}}(\mathbf{z}_j))]\end{aligned}$$

MMD-GAN: Use a reproducing kernel to introduce non-linearity to the discriminator.

Wasserstein GAN: Suppose that the discriminator S is Lipschitz continuous. Then, paramaterize S as a neural network and optimize over all Lipschitz functions.

§23 Lecture 23—16th April, 2023

§23.1 Flows

Instead of minimizing distance, we can try to explicitly learn q as a function of $p_{\boldsymbol{\theta}}$.

Remark 23.1. *Let* $\mathbf{T}\colon \mathbb{R}^d \rightrightarrows \mathbb{R}^d$. *Write* $\nabla \mathbf{T} = (\nabla T_1, \ldots, \nabla T_d)\colon \mathbb{R}^d \rightrightarrows \mathbb{R}^d \otimes \mathbb{R}^d$.

Since $\mathbf{T} \circ \mathbf{T}^{-1} = \mathrm{id}$, *then* $\nabla \mathbf{T}(\mathbf{T}^{-1}) \cdot \nabla \mathbf{T}^{-1} = \mathrm{id}$ *by the chain rule.*

Theorem 23.1 (Push-forward as change-of-variable)**.** *Let r be any continuous distribution on* $\mathbb{R}^d$. *If the push-forward map* $\mathbf{T}\colon \mathbb{R}^d \to \mathbb{R}^d$ *is invertible, then the density of* $X := \mathbf{T}(Z)$ *is*

$$p(\mathbf{x}) = r(\mathbf{T}^{-1}\mathbf{x}) \cdot \left|\det(\nabla \mathbf{T}^{-1}\mathbf{x})\right| = \frac{r(\mathbf{T}^{-1}\mathbf{x})}{\left|\det(\nabla \mathbf{T}(\mathbf{T}^{-1}\mathbf{x}))\right|}.$$

Extreme proof sketch. Roughly speaking, this means that $p(\mathbf{x}) \,\mathrm{d}\mathbf{x} = r(\mathbf{z}) \,\mathrm{d}\mathbf{z}$, i.e. the "mass" of the distribution under the curve is preserved. We can "rearrange" to write $p(\mathbf{x}) = \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\mathbf{x}} \cdot r(\mathbf{z})$. By definition, $r(\mathbf{z})$ will be $\mathbf{T}^{-1}(\mathbf{x})$. The "derivative" $\frac{\mathrm{d}\mathbf{z}}{\mathrm{d}\mathbf{x}}$ will then be $\nabla \mathbf{T}^{-1}(\mathbf{x})$, but we need it as a scalar, so we take the determinant, and so it pops out as $\left|\det(\nabla \mathbf{T}^{-1}\mathbf{x})\right|$. □

Notation 23.2. Write $p = \mathbf{T}_{\#} r$ to notate the above definition for $p(\mathbf{x})$.

Now, suppose we plug this into the KL-divergence:

$$\min_{\mathbf{T}} \mathrm{KL}(q \parallel \mathbf{T}_{\#} r) \approx \max_{\mathbf{T}} \frac{1}{n} \sum_{i=1}^{n} \left[\log r(\mathbf{T}^{-1}\mathbf{x}_i) - \log \left|\det \nabla \mathbf{T}(\mathbf{T}^{-1}\mathbf{x}_i)\right|\right]$$

and learn $\mathbf{T}$. However, this is pretty hard to do, since we have to express the inverse of $\mathbf{T}$ and the determinant of the $d \times d$ gradient matrix of $\mathbf{T}$.

Instead, we can learn the inverse during training:

$$\max_{\mathbf{S}=\mathbf{T}^{-1}} \frac{1}{n}\sum_{i=1}^{n} [\log r(\mathbf{S}\mathbf{x}_i) - \log|\det \nabla \mathbf{S}\mathbf{x}_i|]$$

but then we need to invert $\mathbf{S}$ to recover $\mathbf{T}$ for sampling. These both suck. We prefer paying the cost during training, since that is a one-time cost.

We can be extremely clever with our construction of $\mathbf{T}$ to try to avoid this. Suppose that we construct $\mathbf{T}$ as a *triangular map* such that $\nabla\mathbf{T}(\mathbf{z})$ is lower triangular, i.e., T_i depends only on the first i inputs. This is very natural, since saying the i-th output can only look at elements "before it" sounds like causality.

Also, suppose that $\mathbf{T}$ is increasing on the j-th output for the j-th input, i.e., the diagonal of $\nabla\mathbf{T}(\mathbf{z})$ is positive.

Now, since $\nabla\mathbf{T}$ is triangular, it's cheap to calculate the determinant as the product of the diagonal. We went from an $O(d^3)$ operation to $O(d)$, which is way better.

Since T_i only depends on x_i and (already solved) previous elements, and it is increasing w.r.t. x_i, we can use binary search to invert each element. Bisections are basically free, so this is also $O(d)$.

Therefore, increasing triangular maps work very well for our purposes. We can also prove that they work.

Theorem 23.2 (Uniqueness for increasing triangular maps)**.** *For any two densities r and p on $\mathbb{R}^d$, there exists a unique (up to permutation) increasing triangular map $\mathbf{T}$ such that $p = \mathbf{T}_{\#}r$.*

If we fix r as noise, this means that any property of the probabilistic density p is fully captured in the deterministic map $\mathbf{T}$!

This means that we can optimize

$$\min_{\mathbf{T}} \mathrm{KL}(q \parallel T_{\#}r) \approx \max_{\mathbf{T}} \frac{1}{n}\sum_{i=1}^{n}\left[\log r(\mathbf{T}^{-1}\mathbf{x}_i) - \sum_{j=1}^{d}\log \nabla_j T_j(\mathbf{T}^{-1}\mathbf{x}_i)\right]$$

where it only takes $O(d)$ time for each training step. There are a lot of models that are just this in disguise.

Autoregressive models (like GPT) calculate $p_1(x_1), p_2(x_2 \mid x_1), \ldots, p_j(x_j \mid x_{<j})$ which will be a triangular map.

If we suppose that each one of these distributions are Gaussian, we get an increasing triangular map. However, nested Gaussians can only produce Gaussians. To add in non-normality, permute the entries randomly after each layer (masked AR flows).

The real-NVP model works by splitting the data into $\mathbf{z}_1 = \{z_1, \ldots, z_{l-1}\}$ and $\mathbf{z}_2 = \{z_l, \ldots, z_d\}$. The first segment is just copied $\mathbf{x}_1 = \mathbf{z}_1$ but the second part is fed through a map

$$T_j(z_j; z_1, \ldots, z_{l-1}) = \exp(\alpha_j(z_1, \ldots, z_{l-1}) \cdot \mathbb{1}_{j\geq l}) \cdot z_j + \mu_j(z_1, \ldots, z_{l-1}) \cdot \mathbb{1}_{j\geq l}$$

where T ends up as a triangular map. If we replace the linear wrapping of Gaussians with neural networks, we end up with a neural AR flow. If we use a polynomial, this is a sum-of-squares model.

Theorem 23.3 (Inverse sampling). *Let $Z \sim \mathcal{U}(0,1)$, F be the CDF of X, and $Q = F^{-1}$ be the quantile function of X. Then, $Q(Z) \sim F$.*

The function $\mathbf{T} \colon \mathbb{R}^d \to \mathbb{R}^d$ pushes the noise Z forward to observation X. The inverse map $\mathbf{T}^{-1}$ pulls observations X back to noise Z. This lets us generate anything from a uniform random generator. In particular, we can send noise to uniform, and then uniform to data.

§23.2 Diffusion models

Suppose we had infinite layers of the masked AR flows. Recall that we set

$$\mathbf{x}_{t+1} \approx \mathbf{x}_t + \eta_t \cdot \mathbf{f}_t(\mathbf{x}_t) =: \mathbf{T}_t(\mathbf{x}_t)$$

In the continuous case, we can express this as an ODE

$$\mathrm{d}\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t)\mathrm{d}t$$

for doing theory. In practice, we use the discrete form, since that's all we can do with real computers.

Suppose each $\mathbf{x}_t \sim p_t$. In particular, since $\mathbf{x}_{t+1} \sim p_{t+1}$, we can write

$$\begin{aligned}\log p_{t+1}(\mathbf{x}_{t+1}) &= \log p_t(\mathbf{x}_t) - \log|\det \partial_{\mathbf{x}} \mathbf{T}_t(\mathbf{x}_t)| \\ &= \log p_t(\mathbf{x}_t) - \log|\det[\mathrm{id} + \eta_t \cdot \partial_{\mathbf{x}} \mathbf{f}_t(\mathbf{x}_t)]| \\ &\approx \log p_t(\mathbf{x}_t) - \eta_t \cdot \langle \partial_{\mathbf{x}}, \mathbf{f}_t(\mathbf{x}_t)\rangle\end{aligned}$$

where we make the last approximation by Taylor expansion of $\log(x)$ at $x = 1$. Then, in the continuous limit as $\eta_t \to 0$, we can conclude that

$$\frac{\mathrm{d}\log p_t(\mathbf{x}_t)}{\mathrm{d}t} = -\langle \partial_{\mathbf{x}}, \mathbf{f}_t(\mathbf{x}_t)\rangle$$

From this, we can develop an MLE and learn.

We define the forward process as going from data to noise following

$$\mathrm{d}\mathbf{x} = \mathbf{f}_t(\mathbf{x}, t)\mathrm{d}t + g(t)\mathrm{d}\mathbf{w}$$

and the reverse process using stochastic gradient ascent following

$$\mathrm{d}\mathbf{x} = \left[\mathbf{f}(\mathbf{x}, t) - g^2(t)\nabla_{\mathbf{x}} \log p_t(\mathbf{x})\right]\mathrm{d}t + g(t)\mathrm{d}\overline{\mathbf{w}} \tag{$\clubsuit$}$$

for some *score function* $\nabla_{\mathbf{x}} \log p_t(\mathbf{x})$. The score function is the only thing we need to learn, because $\mathbf{f}$ is chosen (the forward process) and g is also chosen (the variance of the noise). We will use the forward process to learn the score function, just like how the discriminator in GANs is used during training and discarded.

To develop the addition of noise, we can write a stochastic differential equation

$$\mathrm{d}\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t)\mathrm{d}t + G_t(\mathbf{x}_t)\mathrm{d}\mathbf{n}_t$$

where $\varnothing_t$ is some noise, which we can discretize as

$$\mathbf{x}_{t+1} \approx \mathbf{x}_t + \eta_t \cdot \mathbf{f}_t(\mathbf{x}_t) + \mathbf{g}_t(\mathbf{x}_t) \tag{$\spadesuit$}$$

where $\mathbf{g}_t(\mathbf{x}_t) \sim \mathcal{N}(\mathbf{0}, \eta_t^2 G_t(\mathbf{x}_t) G_t(\mathbf{x}_t)^\top)$.

Trivially, an ODE is just an SDE with $G_t \equiv \mathbf{0}$. Conversely, any SDE is equivalent to an ODE by replacing $\mathbf{f}_t$ with $\mathbf{f}_t - \frac{1}{2}(G_t G_t^\top)\partial_{\mathbf{x}} - \frac{1}{2}(G_t G_t^\top)\partial_{\mathbf{x}} \log p_t$. Since we typically pick G_t to have no $\mathbf{x}$-dependence, the only important term here is the score function.

We can write the reverse-time SDE as

$$\mathrm{d}\overline{\mathbf{x}_{t+1}} = \overline{\mathbf{f}_t}(\overline{\mathbf{x}_t})\mathrm{d}t + G_t(\overline{\mathbf{x}_t})\mathrm{d}\overline{\mathbf{n}_t}$$

where $\overline{\mathbf{f}_t} = -\mathbf{f}_t + (G_t G_t^\top)\partial_{\mathbf{x}} + (G_t G_t^\top)\partial_{\mathbf{x}} \log p_t$ and time flows backwards for $\overline{\text{barred}}$ variables. We can equivalently write the discrete version of (♣), derived by integrating

$$\int_s^{s+\delta} \mathrm{d}\mathbf{x}_t = \mathbf{x}_{s+\delta} - \mathbf{x}_s$$

$$\int_s^{s+\delta} \left[\mathbf{f}(\mathbf{x}, t) - g^2(t)\nabla_{\mathbf{x}} \log p_t(x)\right], \mathrm{d}t = \left[f(\mathbf{x}, s) - g^2(s)\nabla_x \log p_s(x)\right] \cdot \delta$$

$$\int_s^{s+\delta} g(t)\,\mathbf{d}w = g(s) \cdot \sqrt{\delta} \cdot \varepsilon \qquad \varepsilon \sim \mathcal{N}(0, I)$$

to get

$$\mathbf{x}_{s+\delta} - \mathbf{x}_s = [f(\mathbf{x}, s) - g^2(s)\nabla_{\mathbf{x}} \log p_s(\mathbf{x})] \cdot \delta + g(s) \cdot \sqrt{\delta} \cdot \varepsilon$$

Score matching Here we want to get the score function $s_p(\mathbf{x}) = \partial_{\mathbf{x}} \log p(\mathbf{x})$ of p and q to be close using the Fischer divergence:

$$\begin{aligned}
\mathbb{F}(p \parallel q) &:= \frac{1}{2}\mathbb{E}_{X\sim q} \|\partial_{\mathbf{x}} \log p(X) - \partial_{\mathbf{x}} \log q(X)\|_2^2 \\
&= \mathbb{E}_{X\sim q}\left[\frac{1}{2}\|s_p(X)\|_2^2 + \langle \partial_{\mathbf{x}}, s_p(X)\rangle + \frac{1}{2}\|s_q(X)\|_2^2\right] \\
&\approx \hat{\mathbb{E}}_{X\sim q}\left[\frac{1}{2}\|s_p(X)\|_2^2 + \langle \partial_{\mathbf{x}}, s_p(X)\rangle\right]
\end{aligned}$$

where we simplify the cross-term in the second line

$$\begin{aligned}
\int -s_p(\mathbf{x}) \cdot \partial_{\mathbf{x}} \log q(\mathbf{x}) \cdot q(\mathbf{x})\mathrm{d}\mathbf{x} &= \int -s_p(\mathbf{x}) \cdot \partial_{\mathbf{x}} q(\mathbf{x}) \cdot \frac{1}{q(x)} \cdot q(\mathbf{x})\mathrm{d}\mathbf{x} && \text{(chain rule)} \\
&= \int -s_p(\mathbf{x})\,\mathrm{d}q(\mathbf{x}) && \text{(calculus)} \\
&= \cancelto{0}{-s_p(\mathbf{x})q(\mathbf{x})} + \int q(\mathbf{x})\,\mathrm{d}s_p(\mathbf{x}) && \text{(by parts)} \\
&= \int q(\mathbf{x}) \cdot \partial_{\mathbf{x}} s_p(\mathbf{x})\,\mathrm{d}\mathbf{x} \\
&= \mathbb{E}_{X\sim q} \partial_{\mathbf{x}} s_p(X)
\end{aligned}$$

but nobody actually does this because it requires calculating the Hessian of the neural network.

Instead, suppose we have a latent variable Z with joint density $q(\mathbf{x}, \mathbf{z})$. Then, we get instead

$$\mathbb{F}(p \parallel q) := \frac{1}{2}\mathbb{E}_{X\sim q} \|\partial_{\mathbf{x}} \log p(X) - \partial_{\mathbf{x}} \log q(X)\|_2^2$$

$$= \frac{1}{2}\mathbb{E}_{(X,Z)\sim q}\left[\|s_p(X) - \partial_{\mathbf{x}} \log q(X \mid Z)\|_2^2 + \|s_q(X)\|_2^2 + \|\partial_{\mathbf{x}} \log q(X \mid Z)\|_2^2\right]$$
$$\approx \hat{\mathbb{E}}_{(X,Z)\sim q} \|s_p(X) - \partial_{\mathbf{x}} \log q(X \mid Z)\|_2^2$$

If we consider X as the data Z plus some Gaussian noise $\varepsilon \sim \mathcal{N}(\mathbf{0}, I)$. Then, it is easy to obtain the conditional density of $X \mid Z \sim \mathcal{N}(Z, I)$. In particular, $q(X \mid Z) \propto \exp(-\frac{\|X-Z\|_2^2}{2})$ which means $\log q(X \mid Z) \propto -\frac{1}{2}\|X - Z\|_2^2 \propto \|\varepsilon\|$. This means we can interpret the score function as a predictor for the noise, which makes sense, because we are trying to figure out how to remove noise from the image.

Returning to (♠), we want to minimize the Fischer divergence across the entire interval $t \sim \mu$ and learn the score function

$$\min_{\boldsymbol{\theta}} \hat{\mathbb{E}}_{\substack{t\sim\mu \\ (X_t X_0)\sim q(\mathbf{x},\mathbf{x}_0)}} \lambda_t \|s_t(X_t; \boldsymbol{\theta}) - \partial_{\mathbf{x}} \log q(X_t \mid X_0)\|_2^2$$

where we learn a single network which takes in t and $\mathbf{x}_t$.

Then, to do inference, we can either stochastically simulate the discrete reverse-time SDE

$$\mathrm{d}\overline{\mathbf{x}_{t+1}} = -\mathbf{f}_t + (G_t G_t^\top)\partial_{\mathbf{x}} + (G_t G_t^\top) s_t(\overline{\mathbf{x}_t}; \boldsymbol{\theta})\mathrm{d}t + G_t(\overline{\mathbf{x}_t})\mathrm{d}\overline{\varnothing_t}$$

or deterministically follow the discrete reverse-time ODE

$$\mathrm{d}\mathbf{x}_{t+1} = \mathrm{d}f_t - \tfrac{1}{2}(G_t G_t^\top)\partial_{\mathbf{x}} - \tfrac{1}{2}(G_t G_t^\top) s_t(\overline{\mathbf{x}_t}; \boldsymbol{\theta})\mathrm{d}t$$

The problem is that this requires evaluating the score neural network multiple times.

To do interpolation, run the forwards-time model on two images to get noisy points in latent space, then average those and run the backwards-time model to recover an average image.

§24 Lecture 24—18th April, 2023

§24.1 Trustworthy machine learning

Here we are concerned with making machine learning models more trustworthy. This involves ensuring that the models are robust, fair, interpretable, and private, among other societal considerations.

§24.1.1 Robustness and fairness

We can measure models along a bunch of performance metrics: accuracy, training time, memory usage, inference speed, robustness, privacy, fairness, etc. Formally, to define the robustness of a classifier $f\colon \mathbb{X} \to \mathbb{Y}$, given an unseen pair of examples $(x, y) \in \mathbb{X} \times \mathbb{Y}$, $f(x)$ should be y. Attackers have found ways to construct noise that image classifiers will confidently classify as something wrong.

There will always exist *adversarial examples* $\mathbf{x} + \Delta\mathbf{x}$ such that $f(\mathbf{x} + \Delta\mathbf{x}) \neq f(\mathbf{x})$ for all non-constant classifiers. That is, f is not sufficiently smooth/continuous.

To find adversarial examples, we can maximize the loss over a small space. Consider a neighbourhood of size ε. Then, we can write

$$\max_{\|\mathbf{x}-\mathbf{x}_0\|\leq\varepsilon} f(\mathbf{x},\mathbf{y};\mathbf{w})$$

For a *targeted attack* we can set $f(\mathbf{x},\mathbf{y};\mathbf{w}) = \log p_{\tilde{\mathbf{y}}}(\mathbf{x};\mathbf{w})$, i.e., maximise the likelihood anything is labelled $\tilde{\mathbf{y}}$. For an *untargeted attack* we can set $f(\mathbf{x},\mathbf{y};\mathbf{w}) = -\log p_{\mathbf{y}}(\mathbf{x};\mathbf{w})$, i.e., maximise the loss for the correct label.

There are two algorithms we can use. Either take $\mathbf{x} \leftarrow \mathbf{x} + \varepsilon \cdot \operatorname{sign}(\nabla_{\mathbf{x}} f(\mathbf{x},\mathbf{y};\mathbf{w}))$ (fast gradient sign method) or $\mathbf{x} \leftarrow \mathbf{x} + \eta \cdot \nabla_{\mathbf{x}} f(\mathbf{x},\mathbf{y};\mathbf{w})$ (projected gradient method), and then project $\mathbf{x}$ onto the ε-neighbourhood. Naively, we can use the ℓ_2-norm $\|\mathbf{x}-\mathbf{x}_0\|_2 \leq \varepsilon$, but this requires a lot of calculation to project $\mathbf{x}$ back onto the ball. Instead, we can use the ℓ_∞-norm $\|\mathbf{x}-\mathbf{x}_0\|_\infty \leq \varepsilon$ so that projecting $P((x_1,\ldots,x_d)) = P((\min\{x_1,\varepsilon\},\ldots,\min\{x_d,\varepsilon\}))$.

To guard against adversarial examples, we can train across perturbations to get a min-max game between the model training and potential attackers:

$$\min_{\mathbf{w}} \hat{\mathbb{E}}\left[\max_{\|\Delta\mathbf{x}\|\leq\varepsilon} \ell(\mathbf{x}+\Delta\mathbf{x};\mathbf{w})\right]$$

but this loses accuracy in exchange for robustness.

In the case of the linear regression problem

$$\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2 \quad \text{where} \quad X\in\mathbb{R}^{n\times d}, \mathbf{y}\in\mathbb{R}^n$$

when we apply perturbations to each feature, the robust linear regression

$$\min_{\mathbf{w}\in\mathbb{R}^d} \max_{\forall j, \|\mathbf{z}_j\|_2\leq\lambda} \|(X+Z)\mathbf{w}-\mathbf{y}\|_2 \quad \text{where} \quad Z = [\mathbf{z}_1,\ldots,\mathbf{z}_d]\in\mathbb{R}^{n\times d}$$

is actually exactly equivalent to square-root Lasso regularization

$$\min_{\mathbf{w}\in\mathbb{R}^d} \|X\mathbf{w}-\mathbf{y}\|_2 + \lambda\|\mathbf{w}\|_1 \quad \text{where} \quad \|\mathbf{w}\|_1 = \sum |w_j|$$

To avoid the non-differentiability of $|\cdot|$, *Huber loss* wraps the norm with a Moreau envelope to get

$$\begin{aligned} M^\eta_{|\cdot|}(t) &= \min_s \frac{1}{2}(s-t)^2 + \eta|s| \\ &= \begin{cases} \frac{1}{2}t^2 & |t|\leq\eta \\ \eta|t| - \frac{1}{2}t^2 & |t|\geq\eta \end{cases} \end{aligned}$$

which smooths away the point with a quadratic curve. To make sure losses to not grow too big, use *variational loss* which smoothly clips loss functions by $\psi(\eta)$:

$$\tilde{\ell}(t) = \min_{\eta\in[0,1]} \eta\cdot\ell(t) + \psi(\eta)$$

§24.1.2 Data poisoning

Data poisoning is when an attacker injects malicious data into the training set. For example, Carlini *et al.* [CJCC+23] took 60 dollars and bought expired domains from datasets, then retrained models on the poisoned data. They were able to successfully poison the data. Tay was a Microsoft chatbot that used conversations as training data. It turned into a Nazi in a few hours.

Formally, we define a *training distribution* μ and *poisoning distribution* ν. Then, given some (hopefully small) *poisoning fraction* $\varepsilon_d = \frac{|\nu|}{|\mu|}$, we get a mixed distribution $\chi \propto \mu + \varepsilon_d \nu$.

We can formulate the problem as

$$\max_{\nu \in \Gamma} L(\tilde{\mu}; \mathbf{w}_*) \quad \text{s.t.} \quad \mathbf{w}_* = \arg\min_{\mathbf{w}} F(\mu + \varepsilon_d \nu; \mathbf{w})$$

where the attacker crafts poison data ν subject to some constraints Γ, then the defender re-trains $\mathbf{w}$ over the mixed data χ. This has a forced order because we cannot train ν by gradient descent.

Alternatively, a model could be corrupted via parameter corruption where the weights are perturbed directly, giving more control over behaviour.

Definition 24.1 (Model poisoning reachability)**.** *A target parameter* $\mathbf{w}$ *is* ε_d-poisoning reachable *if there exists a poisoning distribution* ν *such that*

$$g(\chi; \mathbf{w}) = g(\mu; \mathbf{w}) + \varepsilon_d g(\nu; \mathbf{w}) = \mathbf{0}$$

so $\mathbf{w}$ *has vanishing gradient over* χ*.*

Example 24.2. The logistic regression has a minimum poisoning transition threshold.

Proof. Recall the logistic regression loss $\ell(\mathbf{z}; \mathbf{w}) = \log(1 + \exp(-\mathbf{w}^\top \mathbf{x}))$ with, as before, gradient $g(\mathbf{x}) = -\frac{1}{1+\exp(\mathbf{w}^\top \mathbf{x})}\mathbf{x}$.
There exists a bound $-0.28 \approx -\mathscr{W}(\frac{1}{e}) = \inf_t \frac{-t}{1+\exp(t)} \leq \langle \mathbf{w}, g(\nu) \rangle$.
Suppose $\mathbf{w}$ is ε_d-reachable. Then,

$$\begin{aligned} g(\mu; \mathbf{w}) + \varepsilon_d g(\nu; \mathbf{w}) &= \mathbf{0} \\ \langle \mathbf{w}, g(\mu; \mathbf{w}) \rangle + \varepsilon_d \langle \mathbf{w}, g(\nu; \mathbf{w}) \rangle &= 0 \\ \varepsilon_d \geq \max \left\{ \frac{\langle \mathbf{w}, g(\mu; \mathbf{w}) \rangle}{0.28}, 0 \right\}, \end{aligned}$$

and so as long as the poisoning fraction remains below that threshold, $\mathbf{w}$ cannot be ε_d-reachable. □

§24.1.3 Differential privacy

In the late 2000s, Netflix held a competition to create a better recommendation algorithm. To do so, they released an anonymized dataset of real users' preferences. However, by cross-referencing anonymized Netflix users with public IMDb profiles, researchers could identify users within the dataset.

But what if we instead released only summary data? This can still leak information.

We can measure differential privacy by how much inclusion/exclusion of a data point affects the publicized result.

Definition 24.3 (Differential privacy)**.** *A randomised mechanism* $M\colon \mathcal{D} \to \mathcal{Z}$ *is* (ε, d)-differentially private *if for any* D *and* D' *in* $\mathcal{D}$ *differing by one data point,*

$$\Pr[M(D) \in E] \leq e^{\varepsilon} \cdot \Pr[M(D') \in E] + \delta$$

for all events $E \subseteq \mathcal{Z}$*.*

The smaller ε or δ are, the stricter the requirement. If $\delta = 0$, we say something is ε-DP.

Example 24.4. Suppose we want to estimate the percentage of cheaters. Each person tosses a coin. If it's heads, they answer honestly. If it's tails, they answer randomly.
This system M is useful and $(\log 3, 0)$-differentially private.

Proof. Notice that $\frac{3}{4}$ of cheaters say yes and $\frac{1}{4}$ of non-cheaters say yes. If p is the real percentage, the system will generate $p' = \frac{1}{4} + \frac{1}{2}p$. Then, recover $p = 2(p' - \frac{1}{4})$. That is, this is a useful system.
Now, notice that if $\delta = 0$, we can write the DP requirement as

$$\log \frac{\Pr[M(D) \in E]}{\Pr[M(D') \in E]} = \log \frac{\int_E p(\mathbf{x})\,\mathrm{d}\mathbf{x}}{\int_E q(\mathbf{x})\,\mathrm{d}\mathbf{x}} \leq \max_{\mathbf{x}} \log \frac{p(\mathbf{x})}{q(\mathbf{x})} \leq \varepsilon$$

Consider when $D = \{\text{cheater}\}$ and $D' = \{\text{non-cheater}\}$ for events $E = \{\text{reports cheating}\}$ and $E = \{\text{reports no cheating}\}$:

$$\log\left(\frac{\Pr[M(D) = \text{reports cheating}]}{\Pr[M(D') = \text{reports cheating}]}\right) = \log\frac{3/4}{1/4} = \log 3$$
$$\log\left(\frac{\Pr[M(D) = \text{reports no cheating}]}{\Pr[M(D') = \text{reports no cheating}]}\right) = \log\frac{1/4}{3/4} = -\log 3$$

Therefore, if we let $\varepsilon = \log 3$, the definition holds, and we can conclude M is $(\log 3, 0)$-DP. □

We can also view this as a hypothesis test. Let $H_0 : D$ be the null hypothesis and $H_1 : D'$ be the alternative hypothesis (or any other binary classification with $\mathcal{Y} = 0$ or 1). Then, let E be the event that $\hat{\mathcal{Y}} = 1$ (i.e., $\hat{\mathcal{Y}} = \mathbb{1}\{M(\cdot) \in E\}$).

We can interpret $\Pr[M(D) \in E] = \Pr[\hat{\mathcal{Y}} = 1 \mid \mathcal{Y} = 0]$ as the false positive rate and $\Pr[M(D') \in E] = \Pr[\hat{\mathcal{Y}} = 1 \mid \mathcal{Y} = 1]$ as the true positive rate. Then, to be (ε, δ)-differentially private means that the statistic has

$$\text{false positive rate} \leq e^{\varepsilon} \cdot \text{true positive rate} + \delta$$

This definition of DP can be generalised.

Definition 24.5 (Rényi differential privacy)**.** *A randomised mechanism* $M\colon \mathcal{D} \to \mathcal{Z}$ *is* (α, ε)-Rényi differentially private *if*

$$\mathbb{D}_{\alpha}(M(D) \parallel M(D')) = \frac{1}{\alpha - 1} \log \mathbb{E}_{X \sim q}\left(\frac{p(X)}{q(X)}\right)^{\alpha} \leq \varepsilon$$

where p and q are the densities of $M(D)$ and $M(D')$.

The Rényi divergence $\mathbb{D}_\alpha$ is defined for $\alpha \in (1, \infty)$. As $\alpha \to 1$, $\mathbb{D}_\alpha$ approaches the KL-divergence. As $\alpha \to \infty$, (α, ε)-RDP approaches ε-DP.

Proposition 24.6. *(ε, δ)-DP has a few useful properties:*

1. *Post-processing: If M is (ε, δ)-DP, then $T \circ M$ is also (ε, δ)-DP for any T.*
2. *Parallel composition: Let $D = \bigcup_k D_k$. If each M_k is (ε, δ)-DP, then $(M_1(D_1), \ldots, M_k(D_k))$ is (ε, δ)-DP.*
3. *Sequential composition: If M is (α, ε_M)-RDP and N is (α, ε_N)-RDP, then releasing $M(D)$ and $N(D, M(D))$ is $(\alpha, \varepsilon_M + \varepsilon_N)$-RDP.*
4. *Group of k: If M is $(\varepsilon, 0)$-DP, then having D' differ from D in k places leads to $(k\varepsilon, 0)$-DP.*

Proof. Omitted; consult [DR14]. □

Example 24.7 (Gaussian mechanism)**.** Let $M(D) = f(D) + \varepsilon$ where $\varepsilon \sim \mathcal{N}(\mathbf{0}, \Sigma)$. Then, M is (α, ε)-RDP with $\varepsilon = \frac{\alpha}{2}\Delta_2 f$ where $\Delta_2 f$ is the ℓ_2-sensitivity $\sup \|f(D) - f(D')\|^2_{\Sigma^{-1}}$.

Proof. Exercise. □

Notice that the privacy is proportional to the sensitivity.

To introduce differential privacy into stochastic gradient descent, we will clip gradients at some maximum bound C (so that no one batch adds too much information)—and this step can be skipped if the gradient is Lipschitz continuous—and add noise to the averaged gradient (using the Gaussian mechanism).

Thus, we can write the DP-SGD algorithm as follows:

Algorithm: Differentially private stochastic gradient descent (DP-SGD)

```
• For $t = 0, 1, 2, \ldots$ do:
    – Sample a random batch $B_t$ of size $b$.
    – For $i \in B_t$ do:
        ∗ Set $\mathbf{g}_i \leftarrow \nabla_{\mathbf{w}} \ell(\mathbf{x}_i; \mathbf{w})$          // compute gradient
        ∗ Set $\mathbf{g}_i \leftarrow \mathbf{g}_i / \max\{1, \|\mathbf{g}_i\|_2 / C\}$          // clip gradient
    – Set $\mathbf{g}_t \leftarrow \left(\frac{1}{b}\sum_{i \in B_t} \mathbf{g}_i\right) + \sigma C \varepsilon$          // add noise
    – Update $\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \mathbf{g}_t$          // update weights
    – Set $w \leftarrow \Pi_{\mathcal{W}}(\mathbf{w})$          // project weights
```

This gives a model that is $\varepsilon\sqrt{k}$-DP, where k is the number of iterations.

§25 Lecture 25—23rd April, 2023

§25.1 Interpretability

We want to understand how a model arrives at a certain output. In particular, given some specific neuron, what is the purpose of that neuron? One way is to run the entire test set and check the neuron's activation for each item. Then, looking at the high-ranked items might give some insight.

Alternatively, maximise the neuron activation by manipulating the data. This will give a fuzzy idea of what the neuron is measuring.

Gradient saliency visualises the gradient at a gradient descent step to see what pixels have the highest response.

Suppose $y = x_1 \vee x_2$. Then, gradient methods fail because if we fix x_1, x_2 does nothing. Likewise, if we fix x_2, x_1 does nothing. Therefore, gradient analysis would find that neither input matters.

We need to find a way to "divide" the contribution of both $x_1 = x_2 = 0$ between the two neurons.

Problem 25.1. *Consider a* coalition game *with n players and a payoff function $u : 2^{[n]} \to \mathbb{R}$ (where $u(\varnothing) = 0$). What is the value of each player $i \in [n]$?*

> **Example 25.2.** In a naive Bayes model trained on the dataset of the *Titanic* passengers, we predict that a 3rd-class adult female passenger survives with probability 0.671.
> The contribution can be broken down as -0.344 for being 3rd-class, -0.034 for being an adult, and $+1.194$ for being male.

Let $u\colon 2^{[n]} \to \mathbb{R}$ be a set function, e.g., accuracy of a subset. We want to find an additive approximation of u, i.e., $\phi\colon 2^{[n]} \to \mathbb{R}$ where $\phi(S) = \sum_{i \in S} \phi(\{i\})$.

To do this, we can find the marginal contribution of i, i.e., $u(S \cup \{i\}) - u(S \setminus \{i\})$. In particular we can leave-one out and find $u([n]) - u([n] \setminus \{i\})$.

Definition 25.3 (Probabilistic value; Banzhaf and Shapley values)**.** *The* probabilistic value *of a player is an additive approximation of the player's contribution to the coalition game, i.e.*

$$\phi_i^p = \phi^p(\{i\}) = \sum_{S \not\ni i} p_{|S|} \cdot [u(S \cup \{i\}) - u(S)]$$

for some given weights p_s.

If $p_s = \frac{1}{2^{n-1}}$ (equally weighted), this is the Banzhaf value*. If $p_s = \frac{s!(n-s-1)!}{n!}$, this is the* Shapley value*.*

> **Example 25.4.** Consider three sellers. The first sells a right hand and the other two sell left hands. The payout for selling a pair is 1 dollar. That is, the utility function is $u(1,2) = u(1,3) = u(1,2,3) = 1$ and 0 otherwise.
> The sellers should probably divide the dollar by giving the first seller a bit more. How much more?

Solution. Calculate the Banzhaf value ϕ_1. The sets $S \not\ni 1$ are $\varnothing$, $\{2\}$, $\{3\}$, and $\{2,3\}$. In the latter three cases, the marginal utility is 1. Since every $p_s = \frac{1}{4}$, $\phi_1 = \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$.
Now, consider ϕ_2 and ϕ_3. The sets $S \not\ni 2$ are $\varnothing$, $\{1\}$, $\{3\}$, and $\{1,3\}$. and $\phi_2 = \phi_3 = \frac{1}{4}$ because only adding to $\{1\}$, changes the value.
However, in this case, $\sum \phi_i > 1$.
The Shapley value assigns $\frac{1}{6}$ to subsets of size 1 and $\frac{1}{3}$ to subsets of size 2. Then, $\phi_1 = \frac{1}{6}+\frac{1}{6}+\frac{1}{3}$ and $\phi_2 = \phi_3 = \frac{1}{6}$. □

The Banzhaf and Shapley values measure the occurances of swing votes. This makes them useful for measuring the "power" of a given voting bloc or representative in a democratic system (and was, in fact, the original motivation for developing these formulae).

Definition 25.5 (Random order value)**.** *Let π be a permutation (a random order) of $[n]$. Suppose $i = \pi(k)$ and define*

$$\psi_i(u, \pi) = u[\pi(1), \ldots, \pi(k)] - u[\pi(1), \ldots, \pi(k-1)]$$

i.e., the utility marginal utility of i joining. Then, define $\phi_i(u) = \mathbb{E}_\pi \psi_i(u, \pi)$. This is the random order value*.*

Shapely articulated axioms that are nice to have for a probabilistic value:

- Linearity: $\phi_i(u + v) = \phi_i(u) + \phi_i(v)$
- Symmetry: if $u(S \cup i) = u(S \cup j)$ for all $S \not\ni i, j$, then $\phi_i = \phi_j$
- Null: if $u(S \cup i) = u(S)$ for all $S \not\ni i$, then $\phi_i = 0$
- Efficiency: $\sum_i \phi_i(u) = u([n])$

Shapely also proved that the Shapely value is the only one that satisfies these axioms. In fact, the Shapely value is equivalent to the random order value with π drawn uniformly.

We can estimate the probabilistic value by estimation with a Monte Carlo simulation:

```
Algorithm: MONTE CARLO ESTIMATION OF PROBABILISTIC VALUE

• For $i = 1, 2, \ldots, n$ do:
    – Set $\phi_i \leftarrow 0$
    – For $k = 1, 2, \ldots, m$ do:
        ∗ Sample a random subset $S$ not containing $i$ with probability $\propto \binom{n-1}{|S|} p_{|S|}$
        ∗ Set $\phi_i \leftarrow \phi_i + u(S \cup \{i\}) - u(S)$
    – Set $\widehat{\phi}_i \leftarrow \phi_i / m$
```

However, this is computationally expensive, since we need to evaluate u twice for distinct n and m, so we would need to train $n \times m$ neural networks. Also, this is not efficient because we might draw the same S for multiple i.

Suppose we minimize

$$\min_{\phi \in \mathbb{R}^n} \sum_{S \subseteq [n]} q_s \cdot [u(S) - \phi(S)]^2 \quad \text{s.t.} \quad u([n]) = \sum_i \phi_i$$

where $q_s = p_s + p_{s-1}$. Then, we recover a normalized probabilistic value. The Shapely value corresponds to $q_s = \binom{n-2}{s-1}^{-1}$. This can be calculated in $O\left(\frac{n}{\varepsilon^2} \log \frac{n}{\delta}\right)$ samples.

In summary, the Shapely value is an example of a random order value, which are special cases of probabilistic values, which are special cases of least-square values.

References

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[CJCC$^+$23] Nicholas Carlini, Matthew Jagielski, Christopher A Choquette-Choo, Daniel Paleka, Will Pearce, Hyrum Anderson, Andreas Terzis, Kurt Thomas, and Florian Tramèr. Poisoning web-scale training datasets is practical. *arXiv preprint arXiv:2302.10149*, 2023.

[DR14] Cynthia Dwork and Aaron Roth. *Algorithmic foundations of differential privacy.* Now Publishers Inc, 2014.

[GBCB16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning.* MIT press, 2016.

[Hal10] Daumé III Hal. *A course in machine learning.* CreateSpace, 2010.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[SSBD14] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: from theory to algorithms.* Cambridge University Press, 2014.