COMP34312: Mathematical Topics in Machine Learning

Notes pack last updated: January 29, 2024

Gavin Brown *Gavin.Brown@manchester.ac.uk*

Anirbit Mukherjee *Anirbit.Mukherjee@manchester.ac.uk*

In this module, we will not to teach you a list of the latest fashionable ML models. You can learn that from other online resources. Even if we did, given how fast the field moves, they'd be out of date in 1-2 years. Instead, we decided to help you understand a fundamental open question, that challenges the state of the art in our field. The topics were selected to be relatively close to our research interests, meaning we can help you understand some of the very latest issues. So, here we go....

Modern ML is going **BIG**. It seems like Google or Facebook put out a press release every other week, about their latest 100 billion parameter deep learning model. Or is it now 200 billion, or 500 billion parameters? This module will give a formal, mathematical treatment to the following question:

"Are bigger models always better models?"

For example, if we keep making bigger and bigger neural networks, will they just keep getting smarter? Is scale really all we need? There's a lot of hype out there. It's hard to know what's real.

The simple answer is 'no'. The performance of a machine learning model is determined by a combination of factors, including the quality and quantity of the training data, the choice of model architecture, and the skill of the person tuning the model. These are all practical issues, that vary with the skill of the practitioner, and availability of compute/data resources. There are however, several fundamental/theoretical issues that apply to everybody. We aim to give you the mathematical and conceptual tools to discuss all this in an informed manner. **We focus on theoretical issues, i.e. there will be no coding of large models in this module.**

Lecture: Tuesdays, 1pm, Simon Lecture Theatre A.
Examples class: Wednesdays 12pm / Thursdays 4pm. Kilburn Building, g23.
Assessment: 100% closed-book exam in late May.

Chapter...	is for the lecture on...	is about...
1	30 January	Recap of COMP24112
2	6 February	Empirical Risk Minimisation
3	13 February	Generalisation bounds
4	20 February	Bias & Variance
5	27 February	Ensemble Learning
6	5 March	Theory of Ensembles
7	12 March	Some Background mathematics
	EASTER BREAK	
8	9 April	Gradient Descent
9	16 April	GD in over-parameterized models
10	23 April	Rademacher Complexity 1
11	30 April	Rademacher Complexity 2

The module will be delivered in weeks 1-6 by Gavin, and in weeks 7-11 by Anirbit.

Contents

0 Notation used in this module

There will be a *lot* of mathematical notation over the next 11 weeks. We have tried to collect the majority of this notation here, for quick reference. At the beginning of the module, you won't know the meaning of all the terms, but they will get slowly used over the weeks, so have patience, and don't panic.

$\boldsymbol{x}$	Input vector
$\boldsymbol{y}$	Output/target vector
$\mathcal{D}$	A probability distribution over examples $(\boldsymbol{x}, \boldsymbol{y})$
$\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})}[\cdots]$	Expectation (i.e. average) over possible examples drawn from $\mathcal{D}$
$\mathcal{S}_n$	A data set $\mathcal{S}_n \coloneqq \{(\boldsymbol{x}_i, \boldsymbol{y}_i) \mid i = 1, \ldots, n\}$, where each $(\boldsymbol{x}, \boldsymbol{y}) \sim \mathcal{D}$
$\mathbb{E}_{\mathcal{S}_n}[\cdots]$	Expectation (i.e. average) over possible datasets $\mathcal{S}_n$
d	Dimension of $\boldsymbol{x}$ (i.e. number of features)
k	Dimension of $\boldsymbol{y}$ (e.g. number of classes)
$f(\boldsymbol{x})$	A model predicting $\boldsymbol{y}$, evaluated at a point $\boldsymbol{x}$.
$\ell(y, f(\boldsymbol{x}))$	A (non-negative) loss function, evaluated at a point $\boldsymbol{x}, \boldsymbol{y}$.
$R(f) \coloneqq \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\ell(\boldsymbol{y}, f(\boldsymbol{x}))\right]$	Population risk of a model
$\hat{R}(f, \mathcal{S}_n) \coloneqq \frac{1}{n}\sum_{i=1}^{n}\left[\ell(\boldsymbol{y}_i, f(\boldsymbol{x}_i))\right]$	Empirical risk of a model, evaluated on a set $\mathcal{S}_n$
Ω	The space of all possible (measurable) functions.
$\mathcal{F} \subset \Omega$	A function class, as a subset of all possible functions.
$f_{erm} \coloneqq \operatorname*{arginf}_{f\in\mathcal{F}}\left[\hat{R}(f, \mathcal{S}_n)\right]$	Empirical risk minimizer in $\mathcal{F}$
$f^* \coloneqq \operatorname*{arginf}_{f\in\mathcal{F}}\left[R(f)\right]$	Population risk minimizer in $\mathcal{F}$.
$y^* \coloneqq \operatorname*{arginf}_{f\in\Omega}\left[R(f)\right]$	Population risk minimizer in Ω, known as the Bayes-optimal model.
$\hat{\mathcal{R}}_n(\mathcal{H})$	Empirical Rademacher complexity of a function class $\mathcal{H}$
$\mathcal{R}_n(\mathcal{H})$	Averaged Rademacher complexity of a function class $\mathcal{H}$
	Given a $\mathcal{F}$ and ℓ as above our typical instance of $\mathcal{H}$ will be the corresponding "loss class", $\mathcal{H} = \{(\boldsymbol{x}, y) \mapsto \ell(y, f(\boldsymbol{x})) \in [0, \infty) \mid f \in \mathcal{F}\}$. Though in the above setups we would also be interested in the Rademacher complexity of $\mathcal{F}$ and accordingly $\hat{\mathcal{R}}_n(\mathcal{F})$ and $\mathcal{R}_n(\mathcal{F})$ will be defined.
$F : \mathbb{R}^p \to \mathbb{R}^q$	An arbitrary function which we may use with gradient descent.

1 Recap of your 2nd year ML module, COMP24112

Motivation. This week we will cover what this module is about, as well as reviewing the necessary background material in the basics of Machine Learning.

We assume you have a good understanding of everything in COMP24112. However, there are some topics where we will focus more than others. In particular, COMP24112 chapters 5 and 6, on loss functions and training schemes. In this chapter we will briefly review this, though not in depth—refer to your COMP24112 notes for full information. The accompanying lecture will cover some of this, plus review the current issues around large models.

Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation.

- the training error, testing error
- overfitting versus underfitting
- the squared loss,
- the cross-entropy loss,
- the gradient of a loss, and global vs local minima,
- gradient descent versus stochastic gradient descent,
- regression trees, and the meaning/importance of tree 'depth'.

1.1 Supervised Learning basics: models and loss functions

In this module we will focus on *supervised learning* tasks. For any input, we have access to a 'label' (sometimes known as a 'target'), which we regard as the correct response for the input. Though it wasn't mentioned last year, the underlying principle for most supervised learning is a mathematical framework called *empirical risk minimization*, which will be introduced over the coming weeks.

Let's look at a concrete (but toy) learning problem, which will illustrate our points. So, here's your *training set*, of $n = 20$ datapoints. Each input is a value on x-axis, the target label you get is the corresponding point on the y-axis. This is a result of some *true* underlying function, plus some noise, i.e. $y = f_{true}(x) + \epsilon$, where $\epsilon = \mathcal{N}(\mu = 0, \sigma = 0.5)$, a random value drawn from a Gaussian with mean 0 and standard deviation 0.5.

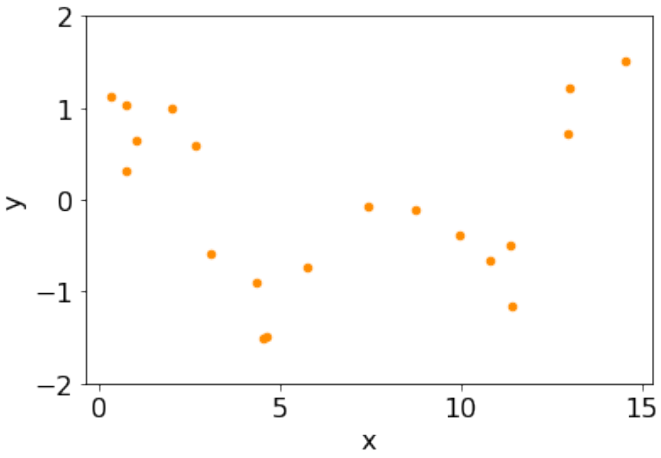
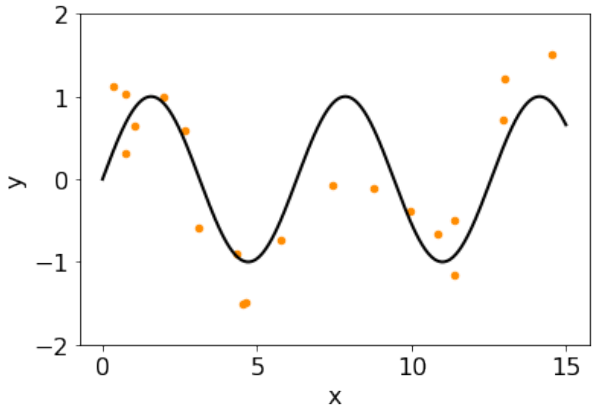


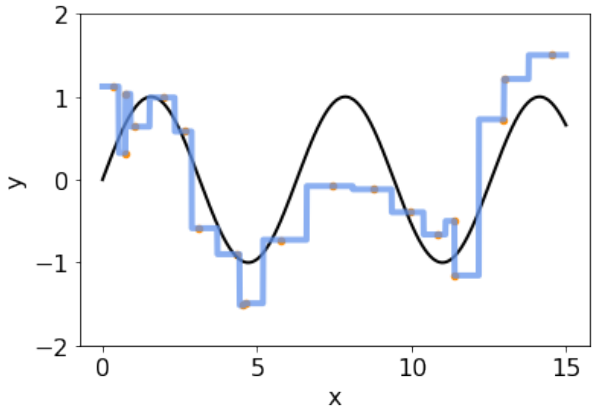
Figure 1: A sample of $n = 20$ points to use as a training set. Can you tell what the true function is?

The task is to learn what the function f_{true} is, from this small (and noisy) sample. Can you see what it is? I'd be surprised if you could...

In fact it's $f_{true} = sin(x)$, shown above. The sample is so small, and so noisy, that it missed the middle



'peak' of the sine wave. Let's see what a Machine Learning model can do with this data. We will use a k-nearest neighbour regression, with a squared distance measure. **If you don't remember what k-nn regression is, go back to your COMP24112 notes (Lecture 2A) for some revision.** The fitted model looks like this:



The blue line shows the fitted predictions of the k-nn regression model, with $k = 1$. We fitted very well (perfectly in fact) to this training set. However, we can see visually that performance on the true underlying function (black line, sine wave) was *terrible*.

We used $k = 1$ here—this determines the *fit* of the model. Maybe if I give you more data, and try different parameters, we can get different performance. Trying $k = 1$, $k = 8$, and $k = 20$, we get the following fits to our new bigger dataset, where $n = 100$.

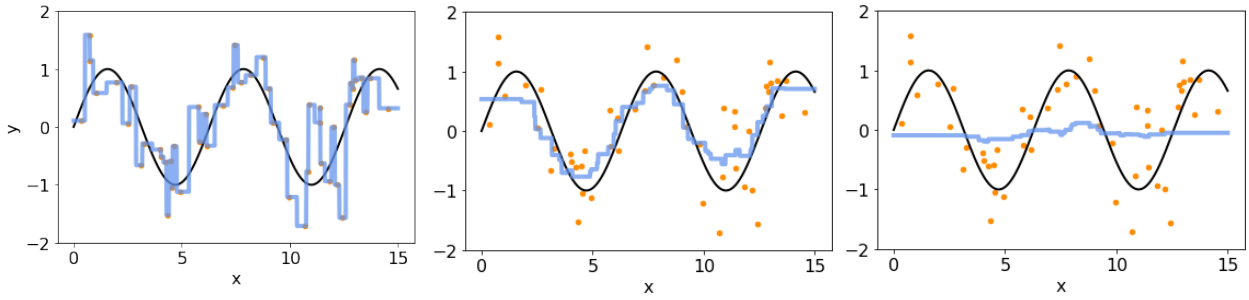


Figure 2: LEFT: $k = 1$, overfitting. MIDDLE: $k = 8$, a good fit. RIGHT: $k = 20$, underfitting.

Hopefully you are remembering some keywords from COMP24112, **over-fitting** and **under-fitting.** The model for $k = 1$ (left) is **over-fitted** because it is "fine-tuned" to the training data, unable to represent the true function—i.e. it has high testing error. The model for $k = 20$ (right) is **under-fitted**, because it has virtually ignored the training data, and is again unable to represent the function, giving again high testing error. The model for $k = 8$ (middle) is a good fit, at least as can be seen visually. As you can see, the idea of over- and under-fitting are intrinsically linked to the notion of *complexity*. The model on the left is very *complex*. The model on the right is very *simple*.

The k-nn is a *memory-based* algorithm, also known as *instance-based learning*. It just memorises the training data, then returns the average label of the k closest neighbours in the feature space, where 'close' is defined by a distance measure—in our case we chose squared distance. This necessitates storing all the training data in memory. As a consequence, it's not great at generalising beyond this training data to new scenarios. So, let's consider other types of models, which are sometimes known as *function approximators*. You met a couple of these in COMP24112, i.e. linear regression, and neural networks, but there are many others:

- Linear/Polynomial Regression
- Support Vector Machines
- Neural networks (e.g. convolutional neural nets, but also logistic regression)
- Naive Bayes (or more generally, probabilistic models)
- Decision Trees (for both regression and classification)
- Ensemble models (i.e. combining some of the models above as a 'committee' model)

The thing that these models have in common is that they work by trying to minimise **loss functions.** Let's see some details of two very common losses.

Loss functions

We can denote a generic model as $f(\boldsymbol{x})$. Here, $\boldsymbol{x}$ is the feature vector, which we will assume is a real-valued vector of length d, or more formally $\boldsymbol{x} \in \mathbb{R}^d$. **In this notation, we have left it implicit that the model has some parameters**, which we denote $\boldsymbol{w} \in \mathbb{R}^p$, i.e. there are p parameters. The model might be for example a neural network, but as you see in the list above, there are many other non-neural models that can perform the same job. For a given model f, we evaluate its performance with a loss function.

For *regression* problems, we predict a value in $\mathbb{R}$, and a very common loss is the *squared* loss function:

$$\ell(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2. \tag{1}$$

Thus, we have a measure of performance for our model f, in predicting the correct value y based on input $\boldsymbol{x}$. We have a *training dataset* of n points: $\mathcal{S}_n^{train} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, and we define the *training error* as the loss averaged over this set:

$$\ell_{train}(f) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(\boldsymbol{x}_i))^2. \tag{2}$$

We then modify the parameters $\mathbf{w}$ so as to minimise this training error:

$$\mathbf{w}^* \;:=\; \underset{\mathbf{w}}{\operatorname{argmin}}\Big[\ell_{train}(f)\Big]. \tag{3}$$

However, our ultimate measure of performance is not the training error, but the performance on a final testing set, that the model has never seen. You can think about this like some lectures (your training set) and a final exam (your testing set).

For a *classification* problem, our model f will output a vector of predicted class probabilities, and we use the *cross-entropy* as our loss. As a reminder, for two classes, the cross-entropy is:

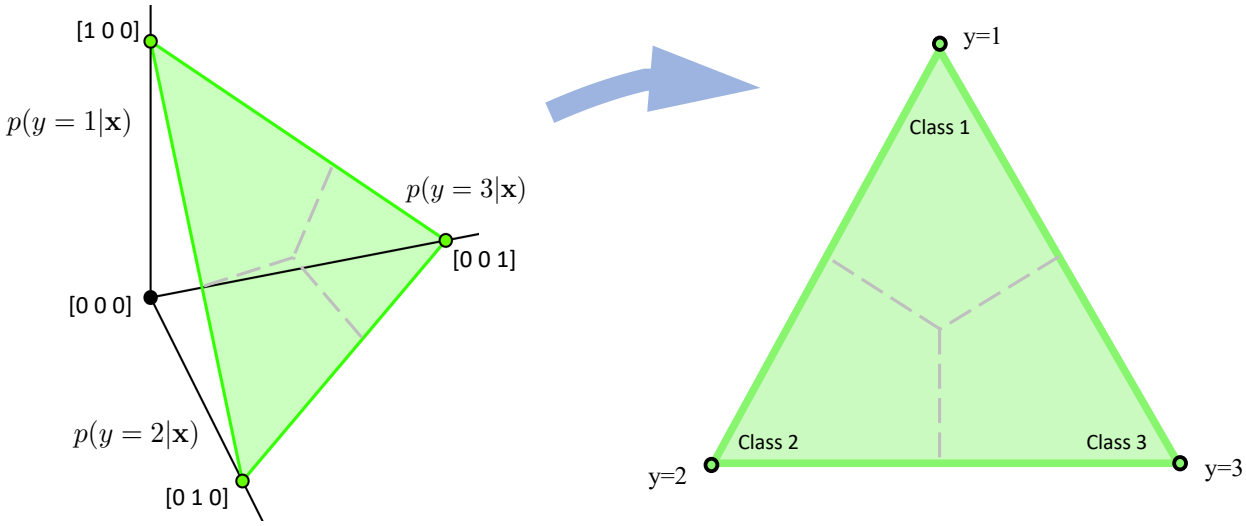
$$\ell(y, f(\boldsymbol{x})) \;:=\; -\Big[y \ln f(\boldsymbol{x}) + (1-y)\ln(1-f(\boldsymbol{x}))\Big] \tag{4}$$

where $y \in \{0,1\}$ and $f(\boldsymbol{x}) \in (0,1)$. For the general multi-class case, it is:

$$\ell(\boldsymbol{y}, f(\boldsymbol{x})) \;:=\; -\Big[\boldsymbol{y} \cdot \ln f(\boldsymbol{x})\Big] \tag{5}$$

where $\boldsymbol{y}$ is a **one-hot** vector. This is a vector of all zeroes, apart from one, which is 1 in the index of the true class. For example, if we have 3 classes, and the correct class for a given example is 3, then $\boldsymbol{y} = [0,0,1]$.

Our model f returns a vector of probability estimates for each class, e.g. $[0.2, 0.3, 0.5]$. We can visualise the 3-class case as a *probability simplex*, where a single prediction $f(\boldsymbol{x})$ is a point on the simplex.



Again, refer back to COMP24112 notes (lectures 5A, 5B, and 5C) for some revision.

Concept Check...
I give you an example where the true label is $y = 1$.
ho My model predicts the probability of $y = 1$ as $f(\boldsymbol{x}) = 0.95$.

Q1. Calculate the cross-entropy loss. Remember to use the **natural** logarithm.

Now, I give you an example $\boldsymbol{x}$, where $\boldsymbol{y} = [1,0,0]$.
My model predicts $f(\boldsymbol{x}) = [0.3, 0.6, 0.1]$.

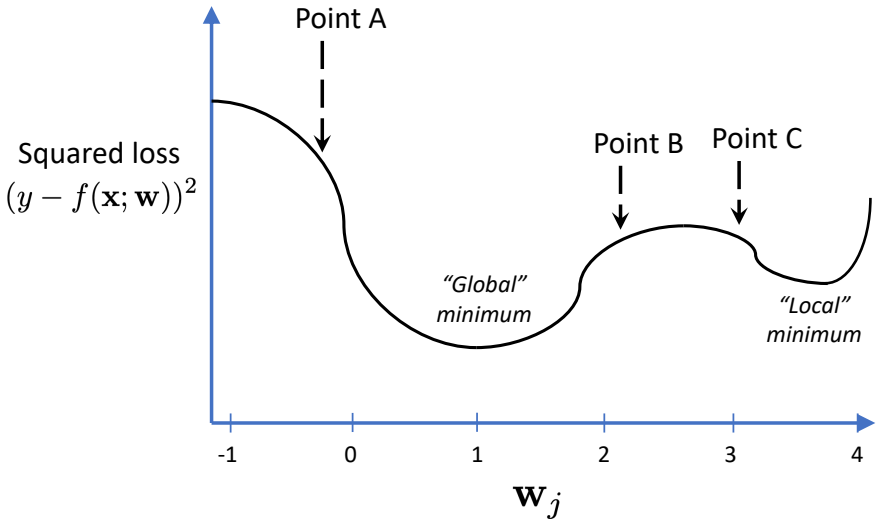
Q2. Calculate the cross-entropy loss again.

Q3. Roughly where in the simplex does this model sit?

1.2 Gradient Descent

One of the most heavily used principles for a learning algorithm is *gradient descent*. In the final weeks of this module, we will be studying advanced aspects of GD, so it's important you understand the basics.

Imagine we have a squared loss function, applied to the output of a big neural network. Somewhere within that neural network there is a weight, $\mathbf{w}_j$, that we decide to change. We vary $\mathbf{w}_j$ *manually* up and down, and record what the squared loss is. The result looks like this:



But we can't manually plot the loss like this, as it's too computationally expensive. If we could, we'd know immediately to set our parameter to the **'global' minimum**, where $\mathbf{w}_j = 1$. And, we'd want to avoid the inferior **'local' minimum** where $\mathbf{w}_j = 3.7$, or indeed any other place on the landscape. But we can't. Instead we can only guess some initial value for $\mathbf{w}_j$, measure the loss, and then guess how we should change it. Consider the situation if we started from point A, B, or C, above.

The factor that differentiates point A from B is the *gradient*, i.e. the slope of the function. Point A is a negative (downhill) gradient, while point B is a positive (uphill) gradient. The principle of gradient descent is to minimize a loss, in the following manner.

> **If the gradient is NEGATIVE, we INCREASE the parameter.**
>
> **If the gradient is POSITIVE, we DECREASE the parameter.**

You can see that by increasing the parameter from point A, we do indeed approach the global minimum. The reverse is true for point B, decreasing the parameter would cause us to minimise the loss. It's important to see that, if we had followed this principle from point C, we would have ended up at a sub-optimal place, the 'local' minimum.

We can calculate the gradient *at any location* by taking derivative with respect to $\mathbf{w}_j$. Remembering that we are using the squared loss,

$$\ell(y, f(\boldsymbol{x})) = (y - f(\boldsymbol{x}))^2, \tag{6}$$

then the gradient is:

$$\frac{\partial \ell(y, f)}{\partial \mathbf{w}_j} = \frac{\partial \ell(y, f)}{\partial f(\boldsymbol{x})} \frac{\partial f(\boldsymbol{x})}{\partial \mathbf{w}_j} = 2\Big(f(\boldsymbol{x}) - y\Big) \frac{\partial f(\boldsymbol{x})}{\partial \mathbf{w}_j}. \tag{7}$$

If you don't know how this gradient is calculated, you should remind yourself of basic calculus—see also Lecture 6C from COMP24112.

Take a look at the landscape on the previous page. If we were at point A (negative gradient) we would increase $\mathbf{w}_j$, exactly as we decided we needed to do in the plotted landscape, and similarly for point B (positive gradient). This can be built into an algorithm.

Algorithm 1 Gradient Descent for parameter vector $\mathbf{w}$ on a differentiable loss $\ell(y, f(\boldsymbol{x}))$

1: **Choose :** A value $T > 0$ for the number of steps to take.
2: **Choose :** A constant η, for the size of each step.
3: **Input:** An initial vector $\mathbf{w}$ of length p.
4: **for** $t = 1, \ldots, T$ **do**
5: $\quad \mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \nabla \ell(y, f(\boldsymbol{x}))$
6: **end for**
7: **Output : w**

Notice that here we used the 'nabla' notation. ∇. which denotes the fact that $\boldsymbol{w}$ is a vector. For a vector $\boldsymbol{w}$ of length d, we have the gradient vector:

$$\nabla \ell\Big(y, f(\boldsymbol{x})\Big) = \left[\frac{\partial \ell(y, f(\boldsymbol{x}))}{\partial \mathbf{w}_1}, \frac{\partial \ell(y, f(\boldsymbol{x}))}{\partial \mathbf{w}_2}, \ldots, \frac{\partial \ell(y, f(\boldsymbol{x}))}{\partial \mathbf{w}_d}\right]^T. \tag{8}$$

Notice also in the above, we are calculating the gradient only for a *single* datapoint $(\boldsymbol{x}, y)$. In reality we will have a full training set to deal with, so what does this mean?

Look at line 5 in the above algorithm. This is known as the **update rule** for gradient descent. If we had a training set of n datapoints, we could replace this with the following, sometimes known as **full batch gradient descent**.

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{n} \sum_{i=1}^{n} \left[\nabla \ell(y_i, f(\boldsymbol{x}_i))\right] \tag{9}$$

Alternatively, we could randomly pick a sample $\mathcal{S}$ of m datapoints, and use the following update rule, known as **mini-batch gradient descent**:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \cdot \frac{1}{m} \sum_{(\boldsymbol{x}_i, y_i) \in \mathcal{S}} \left[\nabla \ell(y_i, f(\boldsymbol{x}_i))\right]. \tag{10}$$

If we take the extreme, $m = 1$, this is **stochastic gradient descent** (SGD), though terminologies vary across the ML community, so you may see this referred to as SGD even if $m > 1$. One reason to use a smaller batch size is to have stochastic estimates of the gradient, which tend to help in escaping local minima.

Note: The above algorithm is a *first-order* gradient descent algorithm. There are alternatives, which utilise second-order derivatives of the loss landscape, for example the popular ADAM algorithm.

Concept Check...
Prove Equation 7.

1.3 Decision trees

One important model family is *decision trees*. These are used widely in industry, as they are fast to both train and deploy. They are excellent models for so-called **tabular** data, which means basically anything that fits in a spreadsheet—not images, or speech signals, or videos.

Decision trees come in two types—for classification, and regression. The basic idea is to recursively split the dataset into subsets based on the values of input features, and then fit a simple model (such as a constant label, or a linear regression) to each subset. Once the tree is built, it can be used to predict the target variable for new samples by traversing the tree from the root to a leaf node, and then using the simple model associated with that node to make the prediction. Below you can see a classification tree, partitioning a 2D space and predicting the probability of a blue circle in each sub-area.

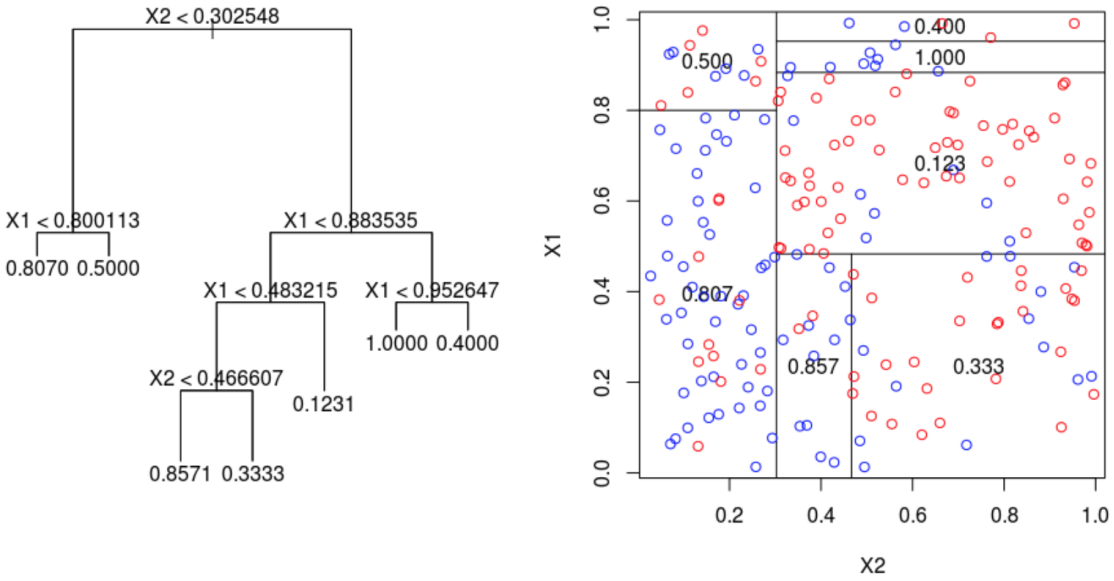


Figure 3: A classification tree.

Regression trees look similar to this, but in each subset of the data they predict a **continuous value**, such as the price of a house. The recursive splits are chosen in such a way as to minimize the variability of the target variable within each subset. We won't go into details of the training procedure here.

With every ML model, there are parameters for you to tune. With trees, the main one is how *deep* it is. This translates to a more fine grained partitioning of the space shown above, equating to a more complex decision boundary. In Figure 5 you can see the decisions from a regression tree as I increase the depth. This is using the sine wave toy problem shown earlier.

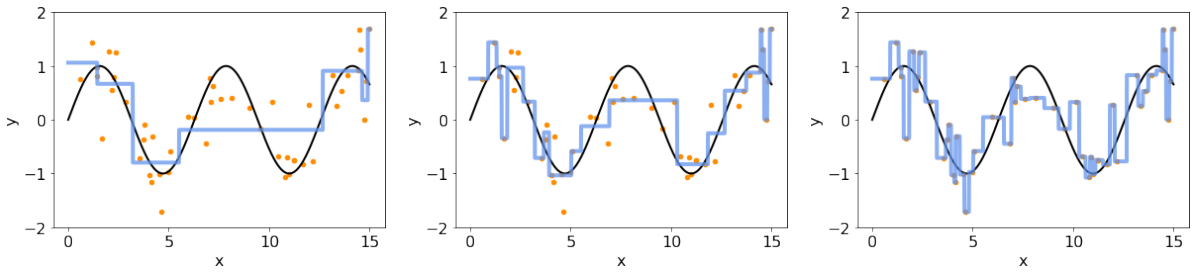


Figure 4: Decision boundaries for trees of increasing depth. From left to right, depth 3, 5, and 16.

Plotting the full range of depth from 1 to 16 reveals that over-fitting occurs around depth 5 or 6. Below we see this, where the darker red lines are averages over 50 repeats of the experiment, and the faint red lines are the 50 individual runs.

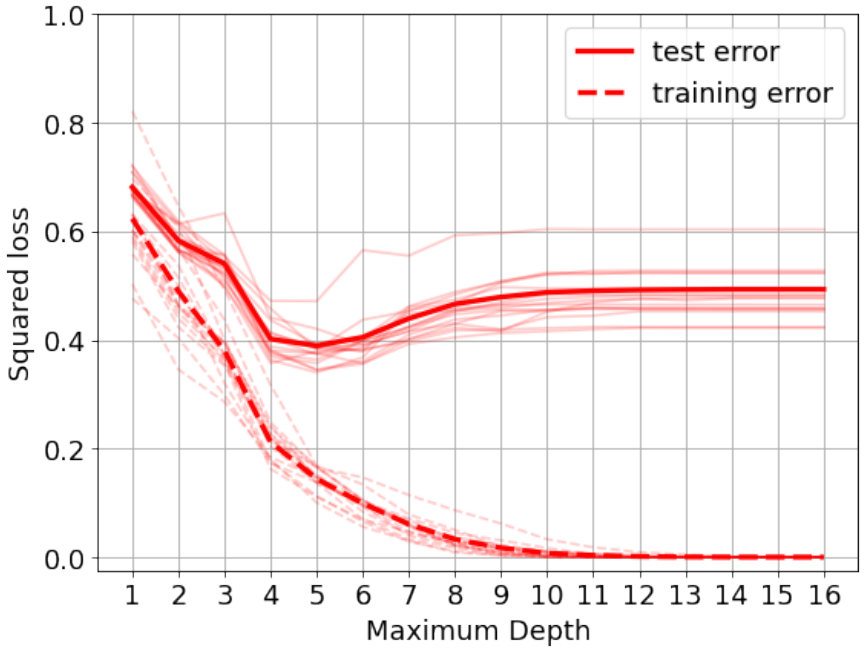


Figure 5: Training and testing loss, as we increase the depth of a regression tree.

1.4 Summary and Outlook for this Module

This brings us to the main question for the module—are bigger models always better? The deep trees on the right are very BIG. They are very deep, and make a complex decision boundary. But you can see they have over-fitted. So here, certainly very big models are not doing well. However, you'd have to have been living under a rock in the desert to not notice the popular press talking about 'deep learning' in the past few years. The idea here is you make very deep/complex neural networks with lots of parameters to tune. And they are doing AMAZING things. You may have heard of MidJourney, Stable Diffusion, and most recently chatGPT. These are models with BILLIONS of parameters to learn.

In this module we will address the theoretical issues around such large models, starting next week with the principle of *empirical risk minimisation*, leading to the wider framework of *statistical learning theory*. SLT provides a formal language to understand the performance of machine learning algorithms and to make informed decisions. It can be used to analyze the robustness and stability of machine learning algorithms, which can be important in safety-critical applications. It also gives an insight on the "sample complexity" of the algorithms which helps in understanding how much data is required to train a model with a certain level of accuracy, this can be useful in cases when data is scarce or expensive to collect.

To prepare for next week, make sure everything in this week's session is **easy** for you. We'll encounter much more challenging mathematics in the weeks to come.

2 Empirical Risk Minimization

Motivation. Today we start the journey addressing our question "are bigger models always better models?". To do this, we will study the mathematical framework of 'statistical learning theory', which we can use to reason about Machine Learning models, data, and algorithms. The first step on the journey is 'empirical risk minimisation', which is the basis of almost all supervised learning algorithms. In fact, you've met this before—you just didn't know the formal details....

Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation.

- **loss**, **empirical risk**, and **population risk** of a model
- empirical risk minimizer
- the best model within a model family/class
- Bayes model
- approximation error
- estimation error
- optimisation error

2.1 Let's start with some terminology

Models. Let's say we want to solve a *classification problem*, e.g., our ML model should take an image and predict whether it's a cat or a dog. We can denote this in abstract mathematical form (i.e. not referring to an particular *type* of ML model) as a function $f : \mathcal{X} \to \mathcal{Y}$. The notation here means a function that maps from some *input* space $\mathcal{X}$ to some *output* space $\mathcal{Y}$. The form of the function itself is determined by the choice of model—e.g., what architecture of neural network, or whether you choose a decision tree, or a logistic regression, or any other model that you've heard of. We denote the **family of models** that we choose by the set $\mathcal{F}$. This is of course a subset of the space of all possible models, which we denote by Ω. It's an important thing to realise that $\mathcal{F}$ is a subset of Ω, i.e. we *always* have a restricted choice of models—the matter of precisely *how* you restrict the model is the key to solving the supervised learning problem.

Input and Output spaces. For a classification problem, the input space is $\mathcal{X} = [0, 255]^{128\times128}$, i.e. an image of resolution 128 by 128 pixels, with each pixel in the range 0 to 255. The output space is $\mathcal{Y} = \{cat, dog\}$, or perhaps for convenience of manipulating the data in a computer we might say it's $\mathcal{Y} = \{0, 1\}$. We might instead define this output space to be a *probabilty* specifying the probability of the image being a dog—then $\mathcal{Y} = [0, 1]$. In general, we will say that our input space is of dimension d and output space is dimension k. In the above example, $d = 16384 = 128\times128$, and $k = 1$ if we predict the probability. For simplicity below, we will assume $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}^k$, but many different learning scenarios and challenges are raised by considering the form these two spaces take.

Data sets. You've been taught the concept of a *training* set, and a *testing* set. In these, each possible data point $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^d \times \mathbb{R}^k$ is assumed to be an independent *sample* from a distribution $\mathcal{D} = P(\boldsymbol{x}, \boldsymbol{y})$. The overall data set $\mathcal{S}_n$ is then a sample from a joint random variable $P(\boldsymbol{x}, \boldsymbol{y})^n$, i.e. sampling n times, independently. Formally, this is known as the *independent and identically distributed* (iid) assumption.

Loss Functions. The framework of *Empirical Risk Minimization* rests on the definition of a *loss function*, which specifies the quality of a prediction, relative to the true answer. This loss function can be written in similarly abstract form, $\ell : \mathbb{R}^k \times \mathbb{R}^k \to [0, \infty)$, meaning it takes two values (the true label and the model prediction), and returns a loss that ranges from 0 upwards (not including infinity). Note, by historical convention, the *first argument* is always the true label, so the loss of model is $\ell(y, f(\boldsymbol{x}))$.

2.2 Empirical versus Population Risk

Until now, you've been taught that the 'error' of a model is the *average loss over a data set*. In the framework of Statistical Learning Theory, this terminology of 'error' is usually referred to as the *risk*. Given a set of data $\mathcal{S}_n = \{(\boldsymbol{x}_i, \boldsymbol{y}_i), i = 1, \ldots, n\}$, we define the **empirical** risk as follows.

Definition 1 (Empirical risk of a model)**.**

$$\hat{R}(f, \mathcal{S}_n) := \frac{1}{n} \sum_{i=1}^{n} \ell(\boldsymbol{y}_i, f(\boldsymbol{x}_i)). \tag{11}$$

The empirical risk can contrasted with the **population risk**, which can be thought of as when $n = \infty$:

Definition 2 (Population risk of a model)**.**

$$R(f) := \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(\boldsymbol{y}, f(\boldsymbol{x}))\Big] = \int \int \ell(\boldsymbol{y}, f(\boldsymbol{x}))p(\boldsymbol{x}, \boldsymbol{y})d\boldsymbol{x}d\boldsymbol{y} \tag{12}$$

Here the notation $\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}$ should be read as the 'expectation with respect to the true data distribution', or informally, the average over all possible data points. The population risk expresses the error that a model f would make, on an average, over all possible inputs. This is also known as the generalisation error. Note that this *does not yet depend on any choice of training data*. We have not yet talked about training a model, we just assume we have f in our hand, somehow.

Learning a model by ERM. Now, again, you've been taught that to obtain a model, we use a training data sample. First, let's assume we have our finite training data $\mathcal{S}_n$, and some very clever learning algorithm that can somehow find a *global minimum* of the empirical risk. This is is (perhaps unsurprisingly) referred to as an *empirical risk minimizer*.

Definition 3 (Empirical risk minimizer in $\mathcal{F}$)**.**

$$f_{erm} := \operatorname*{arginf}_{f\in\mathcal{F}} \hat{R}(f, \mathcal{S}_n). \tag{13}$$

This is the model we would get if we could find the global minimum[1] on the data sample $\mathcal{S}_n$, when we have restricted ourselves to a model family $\mathcal{F}$. Equally, we can define a *population* risk minimizer :

Definition 4 (Population risk minimizer in $\mathcal{F}$, also known as the "best in family" model)**.**

$$f^* := \operatorname*{arginf}_{f\in\mathcal{F}} R(f). \tag{14}$$

This has removed one of our assumptions—we assumed we have infinite data. The final assumption we have is the restriction of a model family, i.e., using $\mathcal{F}$ instead of Ω. If we remove that final restriction, we can define the population risk minimizer as:

Definition 5 (Population risk minimizer in Ω, also known as the Bayes model)**.**

$$y^* := \operatorname*{arginf}_{f\in\Omega} R(f) \tag{15}$$

This is a **hypothetical** model, the optimal model with no restriction on the model family, and no restrictions on the data. This is also referred to as the *Bayes prediction* or *Bayes model*. For squared loss $y^* = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\boldsymbol{x}]$, and for 0/1 loss, $y^* = \operatorname{argmax}_y p(y|\boldsymbol{x})$. **The proof is optional reading at the end of this chapter.** These are however not computable in real scenarios, since it requires the full data distribution (as we can see from the definition of R), to which we don't have access.

[1] Note, the arginf instead of argmin—this acknowledges the fact that the minimizer may not be unique.

Before we proceed any further, it's important you understand these definitions. The empirical risk is the error on a *sample* of data, of a specific size n. The population risk is the *true* risk, where we assume we could somehow get an infinite supply of data. We visualise this below.

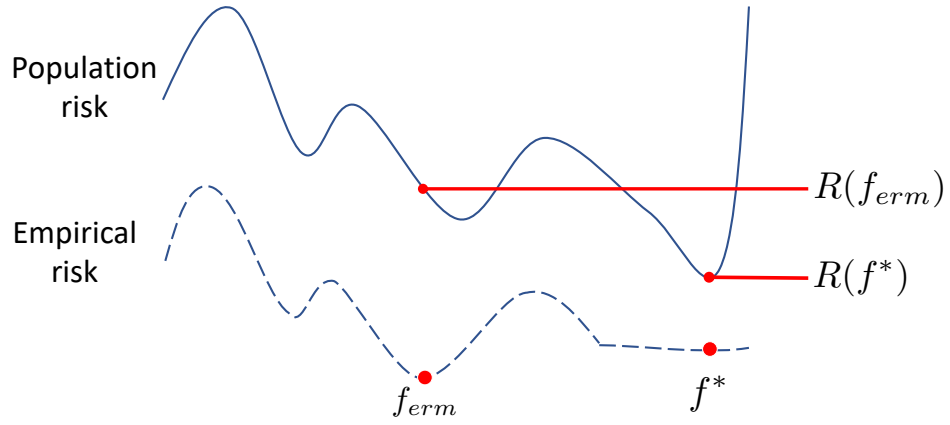


Figure 6: A hypothetical scenario: empirical (i.e. estimated) risk, versus population (i.e. true) risk. Note that here I have offset the landscapes from each other by a constant, so you can see them more clearly. This is fine to do, as the absolute value of the risk does matter, only the gradients and location of the optima.

The horizontal axis represents a chosen space of possible models—you might imagine it corresponds to one parameter somewhere within a neural network. By varying this parameter we obtain different predictive models, with different risks. The model f_{erm} is the global minimum of the empirical risk, and f^* is the global minimum of the population risk. Note that these do not necessarily coincide. This is one of the primary challenges to solve in ML—a small data sample can mis-lead you.

Concept Check...
Note that we have not illustrated the Bayes model y^* in this diagram. Why?

2.3 The Approximation/Estimation decomposition

So far, we have defined three very important models.

- f_{erm} the empirical risk minimizer
- f^* the best model in in our family
- y^* the Bayes model

We can talk about the *risk* for each of these models. The expression $R(f_{erm})$ denotes the population risk of an ERM, and $R(f^*)$ is the population risk of f^*. The term $R(y^*)$ denotes the population risk of the Bayes model, which we conventionally refer to as the *Bayes risk*. Using these, we introduce another important piece of terminology, the **excess risk**.

Definition 6 (Excess risk)**.** *The excess risk of a model f is the difference between its population risk and the Bayes risk, i.e., $R(f) - R(y^*)$.*

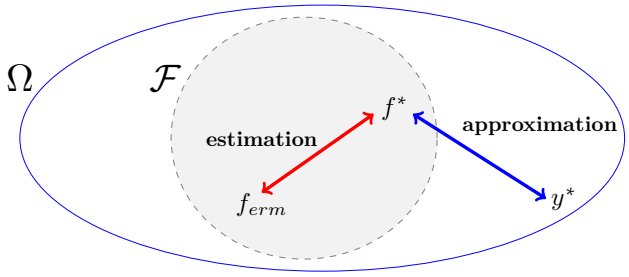
This quantity tells us how much further we could hypothetically reduce the population risk, since we know it is impossible to do any better than the Bayes model y^*. Note that this is the excess risk for an arbitrary model f, but we can also define the excess risk of a different model, e.g., the excess risk of an ERM is the quantity $R(f_{erm}) - R(y^*)$, which will be important to understand the next important concept....

We can now define the so-called **approximation/estimation decomposition**.

Definition 7 (Approximation-Estimation decomposition)**.** *Given an empirical risk minimizer f_{erm}, the excess risk decomposes as follows.*

$$\underbrace{R(f_{erm}) - R(y^*)}_{\text{excess risk of } f_{erm}} = \underbrace{R(f_{erm}) - R(f^*)}_{\text{estimation error}} + \underbrace{R(f^*) - R(y^*)}_{\text{approximation error}} . \tag{16}$$

Whilst this seems a trivial decomposition, the resulting components actually have meaningful interpretations as parts of the excess risk. **Estimation error** is there because we had a limited sample $\mathcal{S}_n$ with which to find f_{erm}. If we had a bigger data set, our ERM would (probably) have been closer to f^*. This quantity depends on the random sample $\mathcal{S}_n$, and thus is a random variable. **Approximation error** is there because we had to restrict our model family. This depends only on the choice of model family, and thus is a constant. We can illustrate[2] the components as follows.

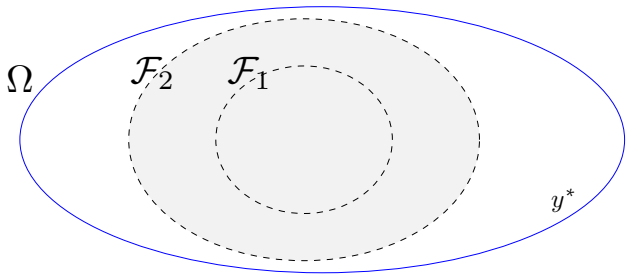


The treats the learning problem in two stages: *approximation*, where we choose a family $\mathcal{F}$; and, *estimation*, where we estimate which model within $\mathcal{F}$ is the best, using our finite data sample.

Changing the size of our model family

Our family, $\mathcal{F}$, might be for example be the weight space for a neural network with a fixed architecture. Or, or all decision trees of a fixed depth. Or, the weight space of a simple logistic regression. The key thing is that we are constrained to only be able to learn things that can be represented by that family of functions $\mathcal{F}$, as a subset of Ω, the space of *all* functions.

Take the neural network example, and call it the set $\mathcal{F}_1$. Now give the network more layers (i.e. make it bigger) and we have the set $\mathcal{F}_2$. These two sets are represented below.

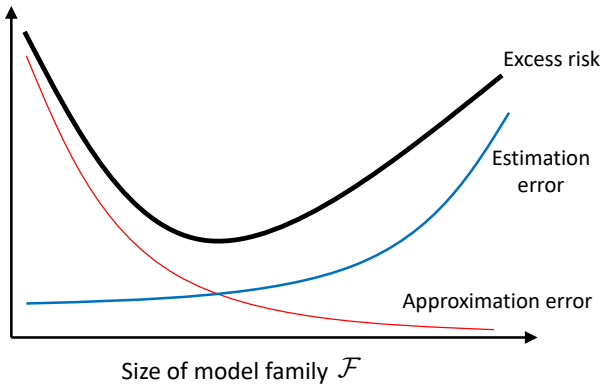


Increasing the complexity of the model has expanded the size of the function class. In other words, a more complex architecture has given us more degrees of freedom, and therefore $\mathcal{F}_2$ can represent more functions than $\mathcal{F}_1$. We also see that in this case, expanding the size of the function class has brought us 'closer' to containing the Bayes model. Be careful, this is not a literal distance, but purely a conceptual illustration.

There are some subtleties here, regarding the relation between the size of the function class, and the complexity of the functions within it. This will become clearer as the course goes on.

[2] Be careful with interpretations of this—it is just an illustration, and the notion of distance is not to be taken literally.

There is a natural trade-off (see below) as we change the size of $\mathcal{F}$, keeping data size fixed. We denote the size of the function class by $|\mathcal{F}|$. As we increase this, approximation error will likely decrease (potentially to zero, if $y^* \in \mathcal{F}$), but estimation error will increase, as it becomes harder to find f^* in the larger space. The reason it is harder, is not just the vast number of functions to evaluate, but also the problem of distinguishing between them with a fixed amount of data. This is in effect the multiple hypothesis testing problem[3]. Since the excess risk is the sum of the two, this tends to give a U-shaped curve. **Do you recognise this phenomenon?**



Recall the idea of model 'over-fitting' from previous modules. If your model is too much fine-tuned to its training data, then it will tend to perform badly on testing data. This is over-fitting.

Overfitting tends to occur when our function class is too **large**, relative to the amount of data we have.

Underfitting tends to occur when our function class is too **small**, relative to the amount of data we have.

Concept Check...
If the Bayes model y^* is in $\mathcal{F}$, then the approximation error is zero. Can you state clearly why?

2.4 The Approximation/Estimation/Optimisation decomposition

I have one final point to make... in the decomposition above we assumed that we can find the empirical risk minimizer. In other words, we assumed we can find the global minimum of the empirical risk on the training data set. This is not realistic, in many scenarios. In most scenarios we can only have a sub-optimal model f. An additional risk component then emerges: $R(f) - R(f_{erm})$, referred to as the *optimisation error*. In special cases (e.g., linear models or very deep decision trees) this will be zero, but in general it is not. **Optimisation error** is the component of the excess risk caused by a poor learning algorithm. In this situation, the excess risk of f decomposes into a sum of *optimisation* error, estimation error, and approximation error:

$$\underbrace{R(f) - R(y^*)}_{\textbf{excess risk of } f} = \underbrace{R(f) - R(f_{erm})}_{\textbf{optimisation error}} + \underbrace{R(f_{erm}) - R(f^*)}_{\textbf{estimation error}} + \underbrace{R(f^*) - R(y^*)}_{\textbf{approximation error}}. \tag{17}$$

This decomposition highlights the 3 design elements that we have to consider to solve any supervised learning problem. These terms describe the learning process in abstract form: accounting respectively for the choice of **learning algorithm**, the quality/amount of **data**, and the choice of **model**. This triple, of *data/model/algorithm* gives you a template which all supervised learning procedures must address.

Poor source of data... your estimation error (and/or the risk of the Bayes model) will be high.
Poor choice of model family... your approximation error will be high.
Poor choice of learning algorithm... your optimisation error will be high.

However, these three components also **interact with each other**. It's not an easy problem.

[3]Google it, or ask chatGPT for a plain english summary.

2.5 OPTIONAL READING: Proving the form of the Bayes model

We defined the Bayes model as the population risk minimizer over Ω. It turns out that we can obtain a closed-form expression for this, for particular loss functions.

Squared loss: For a model, f, the population risk with squared loss is,

$$R(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[(f(\boldsymbol{x}) - y)^2\Big]. \tag{18}$$

Theorem 2.1. *The population risk minimizer (also known as the Bayes prediction) for the squared loss is the expected label, given an input, i.e.* $y^*(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]$.

Proof. The population risk is:

$$R(f) \coloneqq \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[(f(\boldsymbol{x}) - y)^2\Big] \tag{19}$$

At each point $\boldsymbol{x}$, we will minimise the expected loss, i.e. $y^* \coloneqq \operatorname{arginf} \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[(f(\boldsymbol{x}) - y)^2]$.

Now expand the square:

$$\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\Big[(f(\boldsymbol{x}) - y)^2\Big] = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\Big[\,(\,f(\boldsymbol{x})^2 + y^2 - 2yf(\boldsymbol{x})\,)\Big] \tag{20}$$

To find the minimum, we differentiate[4] with respect to f, and set to zero:

$$\frac{\partial}{\partial f(\boldsymbol{x})}\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\Big[\,(\,f(\boldsymbol{x})^2 + y^2 - 2yf(\boldsymbol{x})\,)\Big] = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\Big[\,(\,2f(\boldsymbol{x}) - 2y\,)\Big] = 0 \tag{21}$$

Now...

$$\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\Big[\,(\,2f(\boldsymbol{x}) - 2y\,)\Big] = (\,2f(\boldsymbol{x}) - 2\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\,) = 0. \tag{22}$$

Solving for f, we see for any given input $\boldsymbol{x}$, the minimizer is $f(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]$, which proves the theorem. □

The 0/1 classification loss: The squared loss is appropriate for regression problems, i.e. those where we are estimating a real-valued quantity. If we are instead performing a classification task, e.g. predicting the class label of an image, then a more appropriate option here is the 0/1 loss.

$$\ell_{0/1}(y, f(\boldsymbol{x})) = \begin{cases} 1 & \text{if } y \neq f(\boldsymbol{x})); \\ 0 & \textit{otherwise.} \end{cases} \tag{23}$$

i.e. there is a loss of 1 if we get it wrong, otherwise no loss at all. We define the population risk,

$$R(f) = \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell_{0/1}(y, f(\boldsymbol{x}))\Big] = p(f(\boldsymbol{x}) \neq y) \tag{24}$$

which is just the probability of the prediction not equalling the true label. Similarly, the empirical risk is the is mis-classification rate on a sample of data,

$$\hat{R}(f, \mathcal{S}_n) = \frac{1}{n}\sum_{i=1}^{n}\Big[\ell_{0/1}(y_i, f(\boldsymbol{x}_i))\Big]. \tag{25}$$

Though a proof of this is beyond the scope of this module, the Bayes model for the 0-1 loss is,

$$y^* \coloneqq \operatorname*{argmin}_{f\in\Omega} R\,(f) = \operatorname*{argmax}_{y} p(y \mid \boldsymbol{x}) \tag{26}$$

i.e. the class label y that has the highest (true) probability according to the distribution $p(y|\boldsymbol{x})$.

See a slightly more detailed / different presentation of all this at :
https://stephens999.github.io/fiveMinuteStats/decision_theory_bayes_rule.html

[4] For those worried about differentiating under the integral, it's fine. The Leibniz integral rule applies in almost every problem we might consider, as our spaces $\mathcal{X}, \mathcal{Y}$ can be bounded by constants less than infinity. If you don't know what I'm on about, don't worry, it's beyond the scope of this module.

2.6 Summary

The framework of statistical learning theory gives us a mathematical language with which we can reason about learning problems. The approximation/estimation/optimisation decomposition allows us to discuss the capacity of a model, separately from the data we provide to it, and from the learning algorithm.

A problem with the decomposition is that we cannot *estimate* these quantities on real data. If we could, the quantities would give us some insight into *why* any given algorithm succeeds or fails—in effect they would act as *diagnostics* for what we should be doing to solve the problem.

Concept Check...
Can you state clearly why its impossible to estimate the terms on real data?

There is however, another decomposition, where we can estimate terms. The *bias-variance* decomposition will be the topic of section 4.

3 Generalisation Bounds

Motivation. If I deploy ML models in safety-critical or sensitive domains (e.g., for public/governmental services) then I want to be able to give *guarantees* that the model will perform as I say it will. For example, I want to state *with confidence* that it will maintain a certain level of accuracy. A generalisation bound is a mathematical inequality that enables this. It tells you what the future performance is likely to be, in terms of the training error, and the *complexity/size* of the model.

Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation.

- generalisation bounds
- Hoeffding's inequality
- finite vs infinite function classes

3.1 Lots of maths today. But what will we end up with?

There is a lot of maths today. More than usual. In the end, we will end up with a statement of the form:

$$\textbf{Population risk} \leq \textbf{Empirical risk} + \epsilon \tag{27}$$

where $\epsilon \geq 0$ is a term that depends on (1) the size of the function class $\mathcal{F}$ that you use, (2) the number of data points, n, that you have, (3) a confidence parameter δ. This is called a *generalisation bound*. For example, imagine we have a function class of size $|\mathcal{F}| = 10,000$. We want to make a statement about models learned from this family, with 99% confidence, so $\delta = 0.01$. We will evaluate the models on a testing set of $n = 2000$ datapoints.

Given these three parameters ($|\mathcal{F}| = 10,000$, $n = 2000$, $\delta = 0.01$), our calculation today will tell us that $\epsilon \approx 6\%$. So, if we train a model, and see an empirical risk of 10%, the *true* (population) risk will be at most 16%.

Using this sort of technique, we could provide *guarantees* on the future performance of a model. This might be useful in *safety-critical* or *sensitive* areas: e.g., using ML in governmental/public services, we might quantify the potential dangers of deploying a model.

The particular type of bound we will see today is very simple, and just one of many—the research area is very active. You will see another type of bound in the final weeks of this module.

3.2 How good is our estimate of the generalisation error?

We want a model that minimizes population risk, as far as possible. We know (from last week) that there we can't **measure** the population risk, but instead only **estimate** it, in the form of a data sample of size n, known as the *empirical* risk. The *law of large numbers* is something you've probably heard of. Informally, this states that an estimate of some number will approach the true value, as the sample size increases. But *how fast* does it approach? How many samples do we need to get within a distance ϵ of the true value?

The answer is given by *Hoeffding's inequality*. The full version of the inequality applies for any bounded range $[a, b]$. Today we will illustrate it for the case of $[0, 1]$. Any other finite range can be scaled to this, and the same principle applies. With this, we have the following inequality.

Definition 8 (Hoeffding's inequality, simplified version). *Assume $z_1, z_2, ..., z_n$ are independent instantiations of a random variable Z, such that $z_i \in [0, 1]$. Hoeffding's inequality states:*

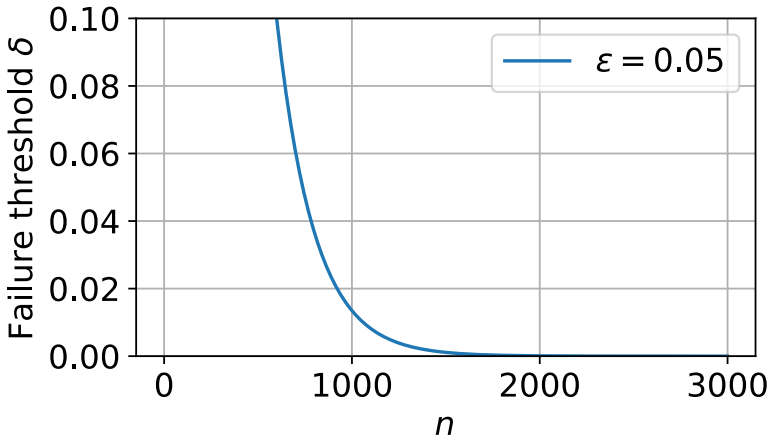
$$p\left(\left| \mathbb{E}[Z] - \frac{1}{n} \sum_{i=1}^{n} z_i \right| \geq \epsilon \right) \leq \delta = 2\exp(-2n\epsilon^2) \tag{28}$$

Proof of this is outside the scope of the module. In plain English, this is saying:

> *Given a true value $\mathbb{E}[Z]$, and an empirical estimate $\frac{1}{n}\sum_i z_i$, the probability of the estimate deviating by more than ϵ from the true value is less than or equal to δ.*

That's quite a sentence, so I'll say it another way. We are estimating some quantity $\mathbb{E}[Z]$, and we want our estimate to be within some tolerance ϵ of this. The chances of being *outside* this tolerance (i.e., having a really bad estimate) is the probability δ. We will refer to δ our *failure threshold*, or *failure probability,* in that we tried to estimate $\mathbb{E}[Z]$ and it 'failed', i.e., was outside our tolerance zone.

Let's plot the function, $\delta = 2\exp(-2n\epsilon^2)$, to see what it looks like and get some more intuition. Here we fix our tolerance at $\epsilon = 0.05$, and vary $n \in [1, 3000]$ samples.



With $n \approx 1000$, we have $\delta \approx 0.0135$. This is the probability of our estimate being more than ϵ away from the true value. If we had a larger sample, $n = 2000$, this reduces drastically, to ≈ 0.00009.

Concept Check...
Calculate δ for $n = 2000$, $\epsilon = 0.05$, exactly. You should end up with $\delta \approx 0.00009$.
If I increase ϵ, i.e. increase my tolerance, what will happen to δ?

So how do we use this in our ML problem? Well, if we use the 0/1 loss, i.e. a classification problem, our empirical/population risks will be values in the range $[0, 1]$, so we can apply the inequality above.

In the inequality above, we will substitute $Z = \ell(y, f(\boldsymbol{x}))$, i.e. the Z is the loss at a random point $(\boldsymbol{x}, y)$.

The **population** risk is then the 'true' value ... $\mathbb{E}_{(\boldsymbol{x},y)}[Z] = \mathbb{E}_{(\boldsymbol{x},y)}[\ell(y, f(\boldsymbol{x}))] = R(f)$

The **empirical** risk is then our 'estimate' ..... $\frac{1}{n}\sum_{i=1}^{n} z_i = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(\boldsymbol{x}_i)) = \hat{R}(f, \mathcal{S}_n)$

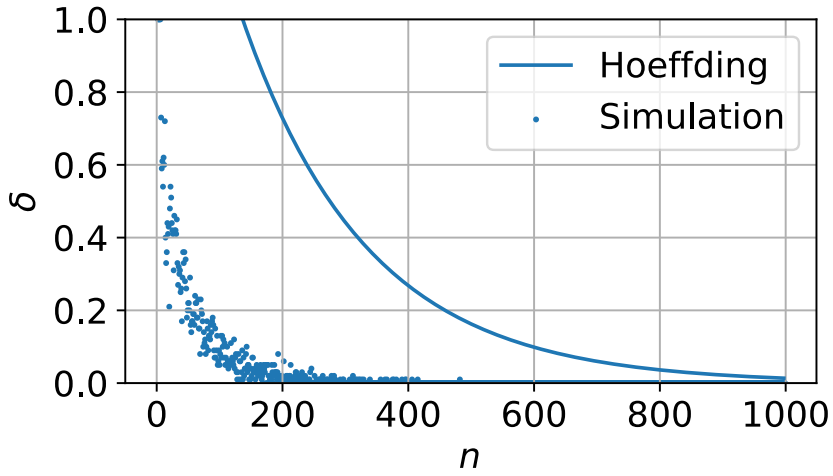
Writing this out ...

$$p\left(\left|R(f) - \hat{R}(f, \mathcal{S}_n)\right| \geq \epsilon\right) \leq \delta = 2\exp(-2n\epsilon^2) \tag{29}$$

This says that, if we have an empirical risk for a **particular** (pre-trained) model, f, on a data sample of size n, the chances of your estimate being more than ϵ away from the true value is bounded by δ.

An important thing to note here, is that **the bound is often extremely *loose***. In the scenario above, this has the consequence that the $n = 2000$ is way more samples than we actually need. We can simulate a hypothetical situation to see this.

Imagine we have a population risk $R(f) = 0.1$. I generate a sequence of n binary values, with $p(z = 1) = 0.1$. To get my estimate, I average these values. I then check whether the estimate is within the tolerance ϵ of the true value, 0.1. I repeat this process over 100 trials, and see how many of the trials were outside the tolerance zone—this is my failure rate, which I can compare to the Hoeffding bound, plotted below.

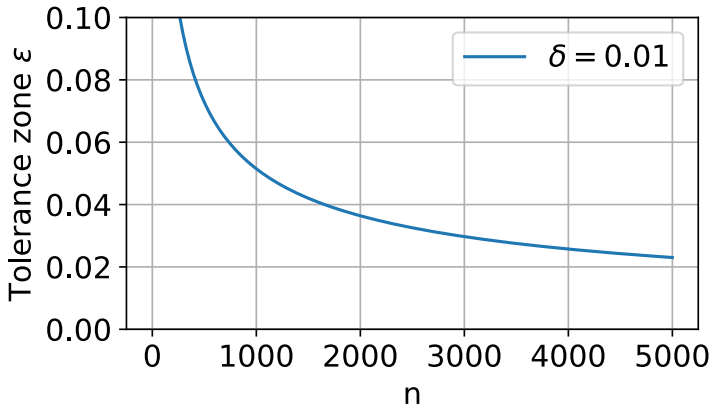


In practice, the probability of being outside $\epsilon = 0.05$ decays to practically zero by about $n \approx 400$, so in this case it's about a 5× overestimate.

We can also rearrange Hoeffding's inequality to give different forms. Solving the expression for ϵ:

$$\epsilon = \sqrt{\frac{\ln(2/\delta)}{2n}}. \tag{30}$$

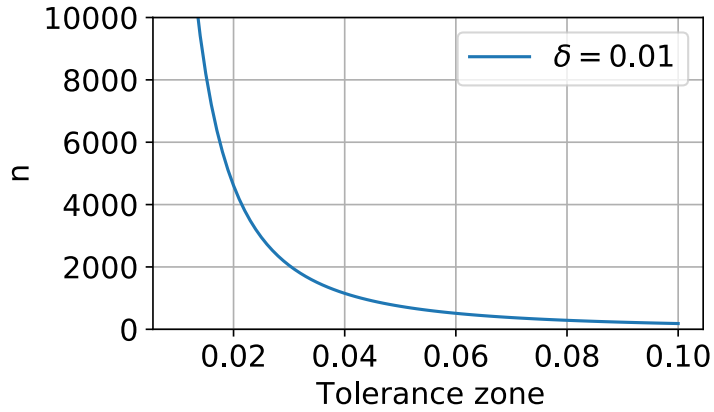
We can now fix our $\delta = 0.01$—which says we want to be 99% confident—then vary n and ask what tolerance ϵ we will have, i.e. how close we will be to the true value.



With 3000 samples, we can be 99% confident we'll be within 0.03 of the true value. With 5000, this gets even closer. Take note... as you saw with the simulation, this bound is loose too.

Concept Check...
Adapt my simulation code to show how close we are in reality, with different numbers of samples.

Or, we can flip this round to see another interpretation. Let's say we're happy with being 95% confident ($\delta = 0.05$), and solving for n (I'll leave that up to you) we get the following figure.



Again, there are things we can read into this—notably we see how many samples we need to be within certain distances of the true value. If we want to be within $\epsilon = 0.02$, we'll need at least $n = 4000$.

3.3 A Generalisation Bound for Finite Function Classes

We're about to see how all this relates to our main question—are bigger models always better?

Hoeffding's inequality as we applied it in Equation (29) is talking about a single (hypothetical) pre-trained model f, i.e. we have already chosen the parameters for the model, and we are just evaluating its risk on our data set. But, when training a model we start with a space[5] of *possible* models, and then use our training data to pick one. As last week, the space of possible models we will refer to as a **function class** and denote it by $\mathcal{F}$.

Since we could in theory end up picking any of the models from $\mathcal{F}$, we'd like to have some guarantee that they will all yield good estimates of the risk, i.e. not violate the tolerance zone we discussed above.

We can do this by considering the probability that **at least one** of the models has a risk estimate outside the tolerance zone. To understand how we are going to compute this, imagine we flip two coins, independently. Define two boolean events, V_1 and V_2. The event V_1 is **true** if coin 1 comes up as heads, and similarly V_2 is **true** if coin 2 comes up as heads. The probability rule for the **at least one** scenario is:

$$P(V_1 \text{ or } V_2) \;=\; P(V_1) + P(V_2) - P(V_1 \text{ and } V_2). \tag{31}$$

Here, the probability of V_1 or V_2 being true, is equal to the sum of the two independent probabilities, minus the probability of both occurring at the same time.

Since $P(V_1 \text{ and } V_2) \geq 0$, we know that $P(V_1 \text{ or } V_2) \leq P(V_1) + P(V_2)$. This inequality generalises to M events, where the logical OR is denoted by the 'union' operator, and is known as Boole's inequality.

Definition 9 (Union bound, also known as Boole's Inequality)**.**

$$P\left(\bigcup_{i=1}^{M} V_i\right) \leq \sum_{i=1}^{M} P\left(V_i\right) \tag{32}$$

Applying this inequality, we find that...

$$P\left(\exists f \in \mathcal{F},\ \left|R(f) - \hat{R}(f, \mathcal{S}_n)\right| \geq \epsilon\right) \;\leq\; \sum_{i=1}^{M} P\left(\left|R(f_i) - \hat{R}(f_i, \mathcal{S}_n)\right| \geq \epsilon\right) \tag{33}$$

where f_i is the ith model in the function class.

Notice that the sum of probabilities on the right is in the form we had from Hoeffding's inequality for one model, $\delta = 2\exp(-2n\epsilon^2)$, and there are simply M of these terms, repeated. We can now state our inequality over the whole space, known as the **uniform deviation bound.**

Definition 10 (Uniform Deviation Bound)**.** *Assume we are picking a model f from an overall model family $\mathcal{F}$, of size $|\mathcal{F}|$. The following bound holds.*

$$p\Big(\exists f \in \mathcal{F},\ \left|R(f) - \hat{R}(f, \mathcal{S}_n)\right| \;\geq\; \epsilon\Big) \leq \delta = 2|\mathcal{F}|\exp(-2n\epsilon^2) \tag{34}$$

Notice the key change, that this bound is now a function of $|\mathcal{F}|$, the size of the function class.

This says: the probability that there exists a model which violates our tolerance zone is less than or equal to δ. Equivalently, we can say that with probability $1 - \delta$, all of our estimates are within the zone, i.e. $\forall f$, we have $|R(f) - \hat{R}(f, \mathcal{S}_n)| < \epsilon$.

[5] It's important to note that we are assuming this space, $\mathcal{F}$, is *finite*. An example of a finite space is the set of all decision trees of fixed depth. However, we are often picking from an **infinite** space—e.g. the space of models defined by d real-valued weights in a neural network. We'll only deal with the finite case for now.

Finally (I'm sure you're glad we are finally there) we can state our **generalisation bound.** First, we take the upper tail of the distributiopn. This is the expression $\delta = |\mathcal{F}|\exp(-2n\epsilon^2)$. Notice we removed the 2 from the right hand side of the bound as we used before. This is because the bound is two a two-tailed scenario. We are only interested in checking our risk is bounded from one direcgtion. We now solve for ϵ.

$$\epsilon = \sqrt{\frac{\ln(|\mathcal{F}|) + \ln(1/\delta)}{2n}} \tag{35}$$

We plug this into $R(f) - \hat{R}(f, \mathcal{S}_n) < \epsilon$ and rearrange:

Definition 11 (Generalisation bound for finite function classes)**.**

$$R(f) \leq \hat{R}(f, \mathcal{S}_n) + \sqrt{\frac{\ln\big(|\mathcal{F}|\big) + \ln(1/\delta)}{2n}} \tag{36}$$

This reads as *"the true error is less than the training error plus a term, dependent on the number of training samples and the function capacity"*. If $\delta = 0.05$, it further translates that we can be 95% confident in this statement.

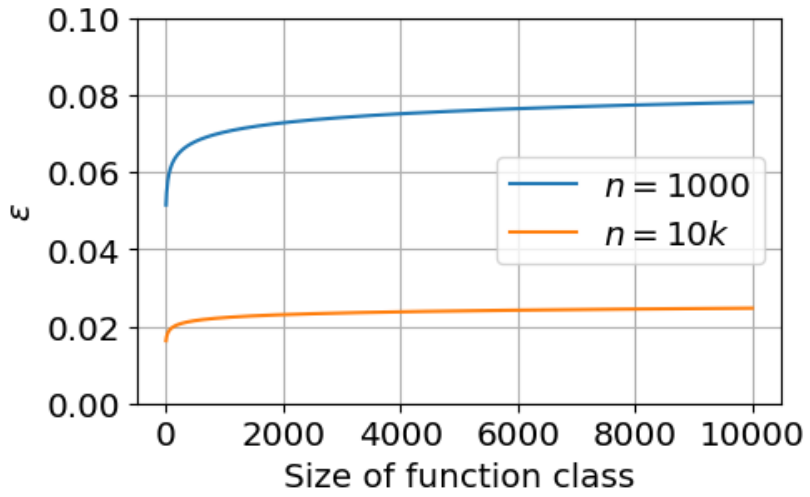


Figure 7: $\delta = 0.05$, i.e. 95% confidence.

3.4 Summary

Today we derived a **generalisation bound**. This is a milestone in this module. We explicitly related the size of the dataset (n), and the capacity of the model (in the form of the function class $\mathcal{F}$) to the generalisation error. It showed us that, if we increase the size of the model, our confidence in the error estimate drops, unless we also increase the size of the data alongside it.

Whilst this is all very interesting, it is somewhat esoteric. Next week we'll see a little more of a practical element, in the *bias-variance* decomposition. This is kind of like the approximation-estimation decomposition, but where we can additionally estimate the two terms, and plot them for real models.

4 The Bias-Variance decomposition

Motivation. In week 2 we met the approximation-estimation decomposition. This showed there were two *components* to the excess risk, with different interpretations. It turned out to be impossible to estimate the terms from data. But, there is another decomposition, with similar interpretations, and we can estimate the terms. Today we meet the *bias-variance* decomposition, and the bias-variance trade-off. If you fully understand this, it will fundamentally shift how you think about ML. Let's go.

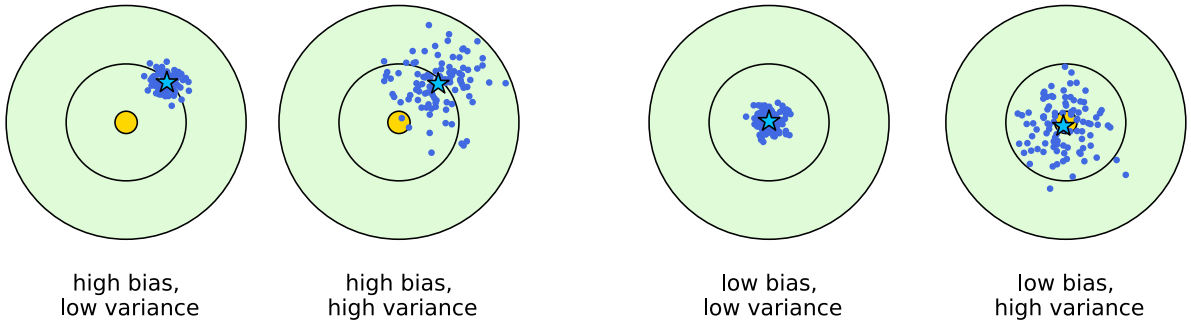
Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation.

- the expected risk of a model
- the bias-variance decomposition for squared and cross-entropy losses
- the bias-variance trade-off
- the trade-off in 'over-parameterized' models

4.1 The intuitive explanation

Imagine I want to train a model to throw darts at a dartboard. In the visualisation below, the bullseye (yellow circle) is the target for a *single test point*. I take a training set $\mathcal{S}_n$, train a model, and make a prediction. Each blue dot is a prediction from a model using a different randomly sampled *training* set.



(a) Predictions from two models with high bias.

(b) Predictions from two models with low bias.

First, look at the pair of dartboards on the left. Both models here have missed the target by quite a long way—so we call them *high bias* models. However, one is a tight cluster, and one is more spread out. If the cluster is tight, it means the predictions did not change much as a result of getting a different training set. This is called a *low variance* model, and is clearly quite *insensitive* to the training data, producing mostly the same prediction each time. The other one is very spread out, sensitive to which training data it got, so we call it a *high variance* model.

Now, look at the pair of dartboards on the right. Both these clusters are centred on the bullseye, so they are called *low bias* models. The tightly clustered one is not so sensitive to its training data, thus we call it a *low variance* model. The one that is more spread out is clearly quite sensitive to which training data set we provide, so is called *high variance*.

This is the very rough idea. The **variance of a model** is a measure of sensitivity to the training data. The **bias of a model** is how far the cluster of predictions are from the target, which roughly translates to a measure of strength in the predictor. These are only rough descriptions—for example, there is another component called the 'noise' which we cannot easily illustrate with the dartboard analogy. To fully understand the details, it requires us to go into the mathematics.

4.2 The mathematical explanation

Note: *Our model f is dependent on the training data $\mathcal{S}_n$ that we give to it. So, in the text below, strictly, we should ideally be using notation $f(\boldsymbol{x}; \mathcal{S}_n)$, instead of just $f(\boldsymbol{x})$. But for compactness we chose not to do so, so you'll have to remember this dependency yourself.*

Our training set $\mathcal{S}_n$ is just one from the space of *possible* training sets that we could have seen. Each data point $(\boldsymbol{x}, \boldsymbol{y}) \in \mathbb{R}^d \times \mathbb{R}^k$ is assumed to be an *independent* sample from an unknown distribution $\mathcal{D} = P(\boldsymbol{x}, \boldsymbol{y})$. The training data set $\mathcal{S}_n = \{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1}^n$ is therefore a sample from a joint random variable $P(\boldsymbol{x}, \boldsymbol{y})^n$, i.e. sampling n times independently. We can therefore consider the **average risk**, over *all possible training sets* we might have encountered. We call this *expected* risk—we'll first look at *squared loss*, and progress to others.

Definition 12 (Expected squared risk)**.** *The expected risk is the population risk averaged over all possible training data sets of a fixed size n:*

$$\mathbb{E}_{\mathcal{S}_n}\Big[R(f)\Big] = \mathbb{E}_{\mathcal{S}_n}\Big[\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\left[(f(\boldsymbol{x}) - y)^2\right]\Big] \tag{37}$$

There are two expectations in this expression. The outer one, $\mathbb{E}_{\mathcal{S}_n}$, is averaging over possible *training sets*. The inner one, $\mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}$, is averaging over possible *testing points*. **Take a minute to reflect on this**.

The bias-variance decomposition (Geman et al., 1992) breaks this down into three components.

Definition 13 (Bias-Variance decomposition for squared risk)**.** *Given a model f, and a random variable over training data sets, the expected squared risk decomposes into three components.*

$$\underbrace{\mathbb{E}_{\mathcal{S}_n}\Big[R(f)\Big]}_{\text{expected squared risk}} = \mathbb{E}_{\boldsymbol{x}}\Bigg[\underbrace{\mathbb{E}_{y|\boldsymbol{x}}\Big[\big(y - \mathbb{E}_{y|\boldsymbol{x}}[y]\big)^2\Big]}_{\text{noise}} + \underbrace{\Big(\mathbb{E}_{\mathcal{S}_n}\left[f(\boldsymbol{x})\right] - \mathbb{E}_{y|\boldsymbol{x}}[y]\Big)^2}_{\text{bias}} + \underbrace{\mathbb{E}_{\mathcal{S}_n}\left[\Big(f(\boldsymbol{x}) - \mathbb{E}_{\mathcal{S}_n}\left[f(\boldsymbol{x})\right]\Big)^2\right]}_{\text{variance}}\Bigg]. \tag{38}$$

where $\mathbb{E}_{\mathcal{S}_n}\left[f(\boldsymbol{x})\right] = \int \Big[f(\boldsymbol{x})\Big] p(\mathcal{S}_n)\, \mathrm{d}\mathcal{S}_n$ is the "expected model".

Note: The formal proof of this is optional reading, in subsection 4.5.

The first term is called the *noise*. This is an irreducible constant, independent of any model parameters. It is caused by choice of data/features, and not by the model. The only way to reduce it is to get better quality labelled data, with more informative features. Adding more examples will not reduce this. You might notice that it is equal[6] to the *Bayes Risk*, i.e. $R(y^*)$. We saw this back in Definition 6 of section 2. This reflects that there is a relationship between this bias-variance decomposition and the approximation-estimation decomposition. They are *not* equivalent, but are strongly related.

The second term is the *bias*. This is the loss of the *expected model* against $\mathbb{E}_{y|\boldsymbol{x}}[y]$. The *expected model* is the average response we would get if we could average over all possible training data sets. The main way to reduce this is to increase the flexibility of our model. In the terminology from section 2, this means to increase the size of our model family. We could potentially reduce this by adding more features.

The third term is the *variance*. This captures variation in f due to different training sets, varying around the expected model. If the model is *too* flexible, this will grow large. We could potentially decrease this by increasing the number of training examples, or adding some regularization to the model. Another way to reduce it is the *Bagging* algorithm—this will be covered in section 5.

A bias-variance decomposition holds for several losses, not just the squared loss. For example, it holds for the very commonly used *cross-entropy*, which we reviewed all the way back in subsection 1.1.

[6] Well, as it's labelled in the equation above, not exactly. Can you see the difference?

Bias/Variance for Cross-entropy

Cross-entropy is (by far) the most commonly used loss to train neural networks. The bias-variance decomposition for this case is written in terms of the *Kullback-Leibler* divergence, or KL-divergence, which is related to the cross-entropy in the following manner. Let's assume[7] we have a true class distribution, $\boldsymbol{y} = [0.1, 0.1, 0.8]$. This means for a given input $\boldsymbol{x}$, the most likely outcome is that the true class is 3, but there is a 10% chance that it could be class 1 or 2. The KL divergence is :

$$K(\boldsymbol{y} \,\|\, f(\boldsymbol{x})) \coloneqq \sum_{c=1}^{3} \boldsymbol{y}_c \ln \frac{\boldsymbol{y}_c}{f_c(\boldsymbol{x})} \tag{39}$$

where $f_c(\boldsymbol{x})$ denotes the probability assigned to the cth class by our model.

Concept Check...
Assume $\boldsymbol{y} = [0.1, 0.1, 0.8]$, and $f(\boldsymbol{x}) = [0.2, 0.2, 0.6]$. Compute the KL divergence $K(\boldsymbol{y} \,\|\, f(\boldsymbol{x}))$.

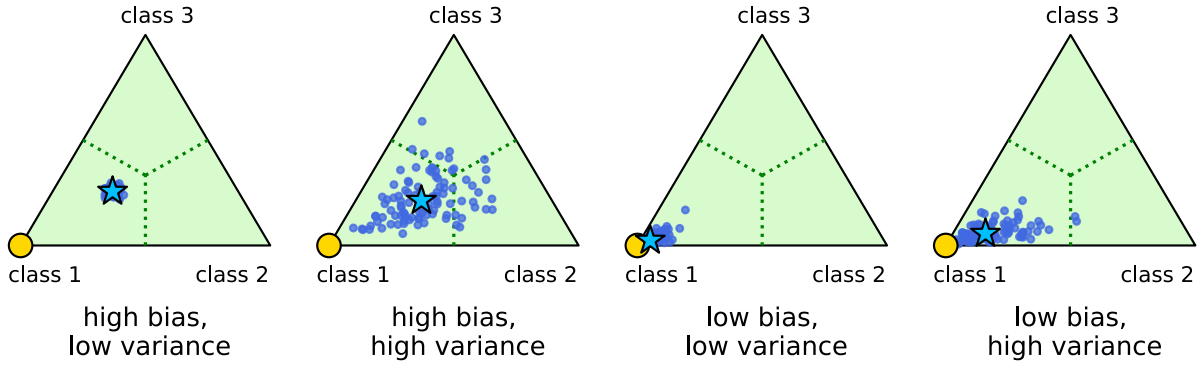
Now, assume $\boldsymbol{y}$ is a one-hot vector, i.e. the label for a *single* $\boldsymbol{x}$. The KL is now equal to:

$$K(\boldsymbol{y} \,\|\, f(\boldsymbol{x})) \;=\; \sum_{c=1}^{3} \boldsymbol{y}_c \ln \frac{\boldsymbol{y}_c}{f_c(\boldsymbol{x})} \;=\; \underbrace{-\sum_{c=1}^{3} \boldsymbol{y}_c \ln f_c(\boldsymbol{x})}_{cross\text{-}entropy}, \tag{40}$$

where use the convention that $0 \ln 0 = 0$. If we use notation $\ell(\boldsymbol{y}, f(\boldsymbol{x}))$ for the cross-entropy, then the following decomposition holds.

$$\underbrace{\mathbb{E}_{\mathcal{S}_n}\Big[\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\Big[\ell(\boldsymbol{y}, f(\boldsymbol{x}))\Big]\Big]}_{\text{expected cross-entropy risk}} \;=\; \mathbb{E}_{\boldsymbol{x}}\Bigg[\underbrace{\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[K(\boldsymbol{y} \,\|\, \boldsymbol{y}^*)\right]}_{noise} + \underbrace{K(\boldsymbol{y}^* \,\|\, \mathring{f}(\boldsymbol{x}))}_{bias} + \underbrace{\mathbb{E}_{\mathcal{S}_n}\left[K(\mathring{f}(\boldsymbol{x}) \,\|\, f(\boldsymbol{x}))\right]}_{variance}\Bigg], \tag{41}$$

where, $\boldsymbol{y}^* = \mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\boldsymbol{y}]$, and $\mathring{f} \coloneqq Z^{-1} \exp(\mathbb{E}_{\mathcal{S}_n}[\ln f])$ is the *normalized geometric mean* of the model distribution. So again, we have **noise**, **bias**, and **variance** terms. These are expressed in KL-divergences, but they have the same interpretation as in the squared loss case. Note that the *geometric* mean has taken the place of the *arithmetic* mean $\mathbb{E}_{\mathcal{S}_n}[f]$ from the squared loss decomposition, so we don't have the 'expected model' any more, and instead refer to it as the 'centroid' model. **Proof of this is well outside the scope of this module, but if you're interested I can point you at relevant literature.** Incidentally, the 'dartboard' analogy can also be made, below.



[7] This does occur in practice, e.g. the problem could be to diagnose a medical condition—and different doctors will have different opinions, so the true outcome is actually subjective, or 'noisy'.

4.3 The bias-variance trade-off

The bias/variance terms can be estimated from data—the Python code for this is provided on Blackboard. We will build regression trees of increasing depth, and predict a noisy sine wave, as we did in one of the previous sessions. On the x-axis we increase the depth of a decision tree. The faint red lines are 50 repeats of the model being trained from scratch with a newly sampled training data set. The bold red line is the average of these, i.e. an estimate of the expected squared risk of the model f.

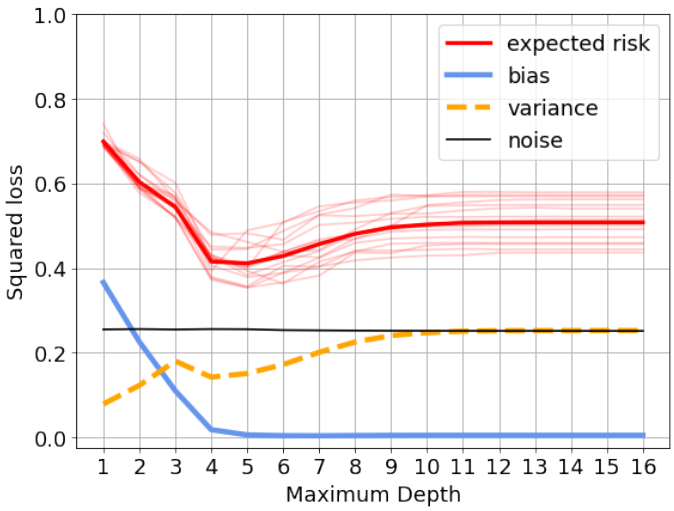


Figure 9: Building regression trees of increasing depth. **Code for this is on Blackboard.**

We see that, as the tree depth increases, the bias tends to *decrease*, and the variance tends to *increase*: this is commonly referred to as the *bias-variance trade-off*. The depth here is our proxy for how *powerful* the model is, or how much *capacity* it has to learn. If it is too 'powerful' it tends to pick up on the noise present in the training data, and hence over-fit.

Whilst we showed this for decision trees, the same principle applies for *all models*—though for some, the trade-off is not so straightforward as shown above.

Linear regression. In linear regression, high bias is associated with the assumption that the relationship between input features and output is linear. If the true relationship is more complex, linear regression may exhibit high bias, leading to underfitting. Variance is generally low in linear regression. You can control the trade-off with L2 weight regularization.

Decision trees. Tree can have low bias, as they can represent complex relationships in the data. However, decision trees are prone to high variance, especially when they grow deep and capture noise in the training data. Techniques like pruning can be applied to control variance and avoid over-fitting.

k-Nearest Neighbors. A k-NN can have low bias, especially in complex, non-linear datasets. However, it may suffer from high variance, particularly in the presence of noisy data or irrelevant features. Choosing an appropriate value of k manages the bias/variance trade-off here.

Neural Networks. Neural networks, especially deep ones, can model highly complex relationships and have low bias. However, they are prone to high variance, especially if the network is too large or trained for too long. Regularization techniques, such as dropout, and early stopping are used to manage variance. Variance can also be kept low through *implicit regularization*—details will be covered in later weeks of this module.

By thinking about the bias-variance decomposition with various models, we gain a nuanced understanding of how different algorithms balance between capturing underlying patterns (low bias) and avoiding noise (low variance). This knowledge helps us make informed decisions when selecting and fine-tuning models.

4.4 Double Descent, and the trade-off in "over-parameterized" models

Now we get to the core question of our module. What happens to bias and variance when you have a HUGE model? Let's quantify this. Imagine we had p parameters, and n training data points.

Definition 14 (Over-parameterization ratio)**.** *With p parameters to learn, and n training points, the over-parameterization ratio is $\rho = p/n$. A model is said to be over-parameterized if $\rho > 1$, i.e. $p > n$.*

With really huge neural nets, we almost always have $\rho \gg 1$. So in these scenarios, when we increase complexity... will the bias go down, as it did in the simple decision tree scenario? Will the variance always go up? The answers are a bit more complex. It turns out that deep learning models display a non-monotonic behaviour in the variance: it first goes up, then down again. See Figure 10.

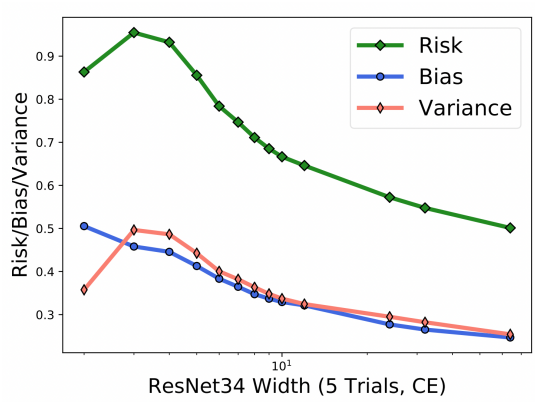
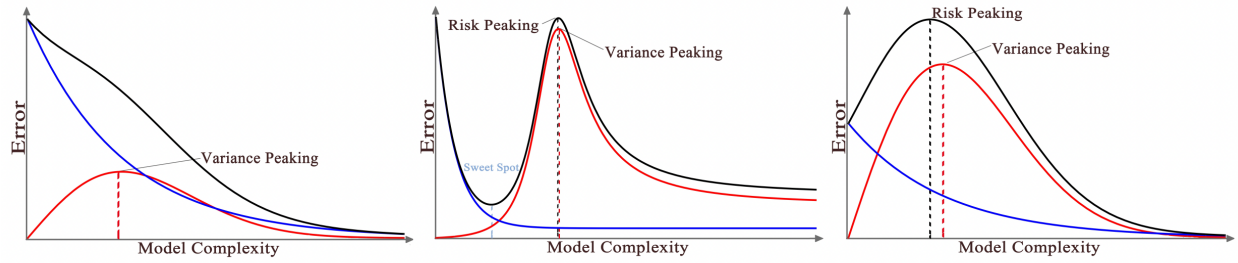


Figure 10: Typical bias-variance tradeoff in a deep neural network as we increase the width of the network, i.e. the number of neurons in a single layer. Figure borrowed from Yang et al. (2020).

There are 3 scenarios which commonly occur with neural nets. These are visualised below.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 11: Typical scenario for expected risk (black) in a deep network: bias (blue) and variance (red). Figure borrowed from Yang et al 2020.

The left scenario is very common. The risk (black line) seems to decrease slowly, but then accelerates somehow. The bias was going down, but the variance was going up a little bit. This is even more pronounced in the far-right scenario, where the risk actually rises a lot, then drops.

The middle scenario is an important and recent issue called 'double descent': the risk initially decreases, then increases, then decreases again. **Exactly why this happens is an open research question.** Current thinking is that the networks are *implicitly regularized* by the stochastic gradient descent algorithm, but the overall issue is far from resolved. The details of this will be covered in section 9.

You can also read more in Yang et al. (2020), if you want to.

4.5 OPTIONAL READING: Proof of Bias/Variance decomposition for Squared Loss

Extra Reading (Non-Assessed)
The proof below is not part of the assessed material, so you will not be asked to reproduce it in an exam. But you may find it interesting, so here we go....

Our model $f(\boldsymbol{x})$, is dependent on a training sample $\mathcal{S}_n$. The decomposition is:

$$\underbrace{\mathbb{E}_{\mathcal{S}_n}\left[\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[(f(\boldsymbol{x})-y)^2\right]\right]}_{expected\ risk} =$$

$$\mathbb{E}_{\boldsymbol{x}}\left[\underbrace{\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[(y-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y])^2\right]}_{noise} + \underbrace{\left(\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2}_{bias} + \underbrace{\mathbb{E}_{\mathcal{S}_n}\left[(f(\boldsymbol{x})-\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})])^2\right]}_{variance}\right] \tag{42}$$

We will now prove this. We note that $\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}[\cdots]$ can be factorized into $\mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\cdots]\right]$, which is true by definition of the expectation. We take the left hand side, for the moment ignoring the expectation over $\mathcal{S}_n$.

$$\begin{aligned}
&\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[(f(\boldsymbol{x})-y)^2\right] \\
&\quad= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]+\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y)^2\right] \\
&\quad= \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2+\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)^2+2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)\right] \\
&\quad= \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2+\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)^2+2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)\right]\right] \\
&\quad= \mathbb{E}_{\boldsymbol{x}}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2\right]+\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)^2\right]+\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)\right]
\end{aligned} \tag{43}$$

We claim that the final term on the right is equal to zero.

$$\begin{aligned}
\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)\right] &= 2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)\right] \\
&= 2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right) \\
&= 0.
\end{aligned} \tag{44}$$

The first step here holds because the term $2\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)$ is independent of the expectation $\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\cdots]$. Now, we reapply the expectation over $\mathcal{S}_n$, and have:

$$\mathbb{E}_{\mathcal{S}_n}\left[\mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[(f(\boldsymbol{x})-y)^2\right]\right] = \mathbb{E}_{\mathcal{S}_n}\left[\mathbb{E}_{\boldsymbol{x}}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2\right]\right]+\mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}\left[\left(\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]-y\right)^2\right]\right] \tag{45}$$

Notice that the expectation over $\mathcal{S}_n$ does not apply to the final term here, as it is independent of the model—this is the *noise* term. Now take the first term on the right, and develop it as follows.

$$\begin{aligned}
&\mathbb{E}_{\mathcal{S}_n}\left[\mathbb{E}_{\boldsymbol{x}}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2\right]\right] \\
&= \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\mathcal{S}_n}\left[\left(f(\boldsymbol{x})-\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]+\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2\right]\right] \\
&= \mathbb{E}_{\boldsymbol{x}}\left[\mathbb{E}_{\mathcal{S}_n}\left[(f(\boldsymbol{x})-\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})])^2+\left(\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)^2+2\left(f(\boldsymbol{x})-\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]\right)\left(\mathbb{E}_{\mathcal{S}_n}[f(\boldsymbol{x})]-\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[y]\right)\right]\right]
\end{aligned}$$

Again we claim that the final term here is equal to zero. The proof is identical to the earlier claim, but using $\mathbb{E}_{\mathcal{S}_n}[\cdots]$ in place of $\mathbb{E}_{\boldsymbol{y}|\boldsymbol{x}}[\cdots]$. Combining all results, we have the decomposition as presented in (42).

4.6 Summary

We've examined the bias-variance decomposition for squared and cross-entropy loss. It shows that a model's expected behaviour (with respect to the random training data you provide it) can decompose into two meaningful components: the bias, and the variance. These allow us to diagnose what's going on inside a model. It also raises some non-intuitive behaviours with *over-parameterized models,* leading to the *double-descent phenomenon.*

It is VERY important to note that such decompositions do not hold for ALL losses. For example, the decomposition as described above does not hold for the 0/1 loss, also known as the classification error.

Next week we'll take a break from all the theory, and look at a practical element—*ensemble methods.* These are **committees of models** that can cooperate to solve problems.

Extra Reading (Non-Assessed)

In section 2, we met the approximation-estimation decomposition. This is **not the same as** the bias-variance decomposition, but there are obvious similarities in the interpretation and behaviour of the components. If you want to read more and find out the exact relation, there is a recent research paper on this issue (Brown & Ali, 2024).

5 Ensemble Methods: bigger, and better?

Motivation. This week we meet something slightly more practical, but still around our main question. If I take a given model, and *duplicate it M* times, with some random perturbations to each, I then have a 'committee' or 'ensemble' of models, each slightly different from one another. The ensemble has more parameters, and thus seems to be *bigger* than a single model. But how does this type of *big* model behave in practice? Just duplicating models seems a bad idea—so how can we build good ensembles?

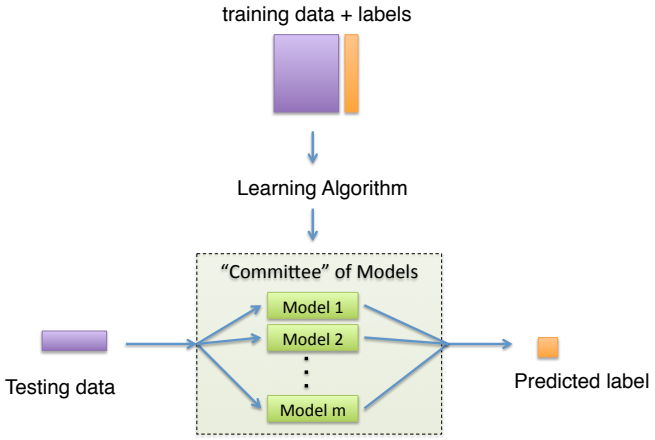
Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation about ML.

- the Bagging algorithm
- the Random Forests algorithm
- the Adaboost algorithm
- the ambiguity decomposition

5.1 Learning with Ensembles of Models

The word "*ensemble*" refers to the idea of having *multiple* predictive models, and *combining*[8] their predictions, treating them as a committee.



With ensembles, we learn multiple models, not just one. When we get a new test datapoint, we pass that point to the different models, and they all predict what the answer should be. Their predictions are merged in some way, so the whole group is used to make the final prediction. or predicting class labels, perhaps directly (e.g. predict $y = 1$), or perhaps via probabilities (e.g. predict $p(y = 1|\boldsymbol{x}) = 0.7$), or a ranking of the set of possible labels. The decisions can be combined by many methods, including voting, averaging, or various probabilistic methods.

The underlying principle of the field is a recognition that in real-world situations, every model has limitations and will make some errors. The "trick" that ensembles can exploit is that when a single model cannot properly fit the data, we can make multiple versions of that same model — each of which make errors in different ways — then, we can vote or average their predictions, cancelling out the errors of the individuals by the

[8] Interesting fact 1: it's actually French, meaning "together". Interesting fact 2: The Beatles wrote a song containing the line: "*C'est sont mots qui vont trés bien ensemble*" – which translates as 'these are words that go together well'.

committee decision. In most cases, the same ensemble algorithm ideas can be applied regardless of what type of model you are using — whether a simple decision tree, or a deep[9] convolutional neural network.

Time has shown, again and again, that while the field comes up with new ideas for building better predictive models, the constant that never goes away is to make *ensembles* out of of them. If we can answer why, when, and how particular ensemble methods can be applied successfully, we will have made progress toward a powerful new tool for Machine Learning: *the ability to automatically exploit the strengths and weaknesses of different learning systems.*

We will start with the idea of combining multiple predictions of a real-valued quantity, so a *regression* problem, and continue with the idea of combining *votes*, for a *classification* problem.

Combining Real-Valued Predictions

There is a reasonably well known incident from 1907 — Francis Galton, a notable scientist of the time, attended a county fair in Cornwall; it was common at the time to play a game where people would pay a penny to guess the weight of an ox, and the person with the closest guess won a prize. Galton recorded all the guesses, and he did publish a paper in Nature about his experiences. The purpose of the study was simply to highlight to farming communities the potential utility of recording data and analysing it.

Galton's paper reported the true weight at 1198lb, and various properties of the 787 guesses from local people on the day. The median guess of the crowd was 1207lb, and the *mean* was 1197lb. So, if a group people on the day had colluded together, and averaged their guesses, there is a good chance they would have been closer to the truth than any individual person — and hence won the prize.

Just one year before, in 1906, a result had been proven that would have added to Galton's paper quite nicely. We know it as *Jensen's inequality*. Imagine we had a set of M real valued numbers, $x_1, ..., x_M$. Jensen's inequality says that

$$\phi\Big(\sum_{i=1}^{M} \lambda_i x_i\Big) \leq \sum_{i=1}^{M} \lambda_i \phi\Big(x_i\Big) \tag{46}$$

where $\sum_i \lambda_i = 1$, and ϕ is any convex function. This seems quite abstract, until we find the analogy to the situation Galton was dealing with. Imagine the values x_i are the M guesses about the weight of the cow, from people on the day, and we have the true weight of the cow, y. Now, we choose $\phi(x) = (x-y)^2$. Further, we assume that $\lambda_i = \frac{1}{M}$ for all i, and use a shorthand $\bar{x} = \frac{1}{M}\sum_i x_i$. With this, Jensen's inequality gives us:

$$\Big(\bar{x} - y\Big)^2 \leq \frac{1}{M}\sum_i \Big(x_i - y\Big)^2 \tag{47}$$

This says that the distance of the mean guess $\bar{x}$ from the truth is less than (or equal to) the average of the distances for each individual guess x_i. The error of the combined guess of the group is *guaranteed* to be less than the average error of each individual. So how much better exactly? It's not too difficult to extend this and show:

$$\Big(\bar{x} - y\Big)^2 = \frac{1}{M}\sum_i \Big(x_i - y\Big)^2 - \frac{1}{M}\sum_i \Big(x_i - \bar{x}\Big)^2 \tag{48}$$

Try proving this for yourself — it's not difficult[10]. Summarising this in machine learning terms, for an ensemble of regression estimators, *the squared error of the ensemble is guaranteed to be less than or equal to*

[9] Yes, even Deep Learning uses ensemble methods. It's not magic.

[10] A hint: start with the right hand side of the inequality, and add/subtract $\bar{x}$ inside the square.

the average squared error of the individual estimators. In the ensemble academic research community, this is known as the *Ambiguity decomposition* (Krogh et al., 1995).

But, the results above are phrased for regression, and we are quite often interested in classification. Perhaps we should think about classifiers like decision trees, that output class labels – if we had an ensemble combined by a majority voting scheme, what guarantees do we have?

Combining votes

In 1785, a French aristocrat and noted mathematician and political scientist, Nicolas de Caritat contributed a work now known as the Condorcet Jury Theorem. The theorem assumes we have a group of voters, who make decisions independently of one another. It also assumes there is a single "correct" decision — obviously not always the case in politics, but let's assume there is one, for now. It also assumes each member makes the *wrong* decision (i.e. commits an error) with known probability ϵ, which is less than 0.5 – so in other words, each voter has a better than 50/50 chance of guessing correctly. In this case, the chances that group (combined by majority vote) makes the correct decision is guaranteed to *increase* as the number of members of the group increases. The first work to apply this to ensemble learning was Dietterich (2000), which is a really nice paper, I recommend you read it. As we said, imagine a single voter has a probability ϵ of making an error, then among a group of M voters, the probability of *exactly* k making errors is given by:

$$p(\text{exactly } k \text{ errors}) = \binom{M}{k} \epsilon^k (1-\epsilon)^{(M-k)} \tag{49}$$

We can then sum this probability for all k greater than half plus one of the voters, in order to get the probability of error in a majority vote.

$$p(majority\ vote\ error) = \sum_{k \geq \lceil \frac{M+1}{2} \rceil} \binom{M}{k} \epsilon^k (1-\epsilon)^{(M-k)} \tag{50}$$

This is illustrated in the figures below.

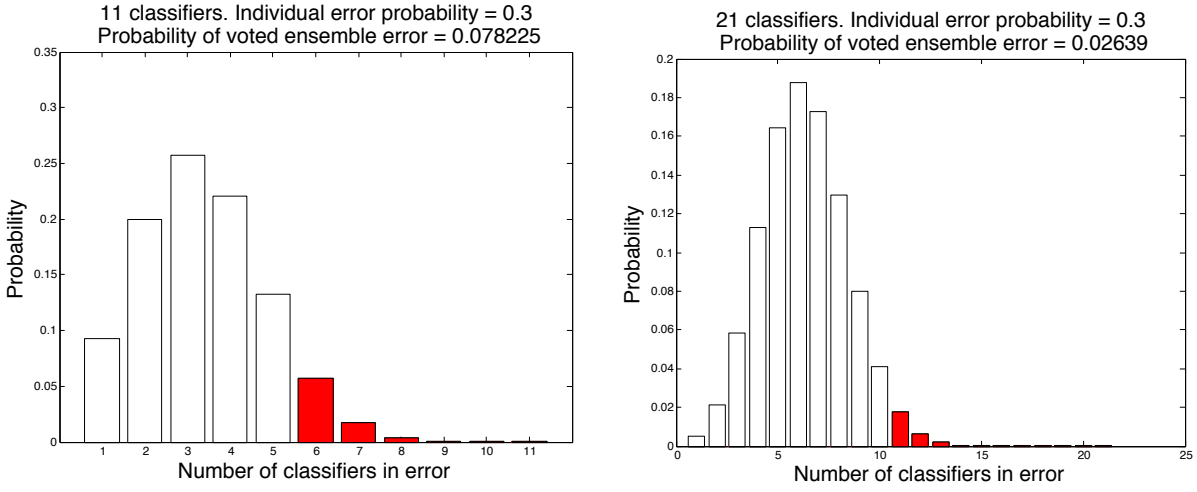


Figure 12: Probability of error for voting ensembles of different sizes (left: 11 classifiers, or right: 21.)

The shaded red area is the probability that the voting ensemble makes an error. Notice in the left figure (with 11 voters) the area is quite big, with a 7.8% chance of ensemble error. In the right figure, (with 21 voters) it is smaller, at about 2.6% chance of error. Yet in both, the chances of an error for an *individual* voter is the same, $\epsilon = 0.3$. The voters are the same, but the ensemble increases in performance because there are more independent voters.

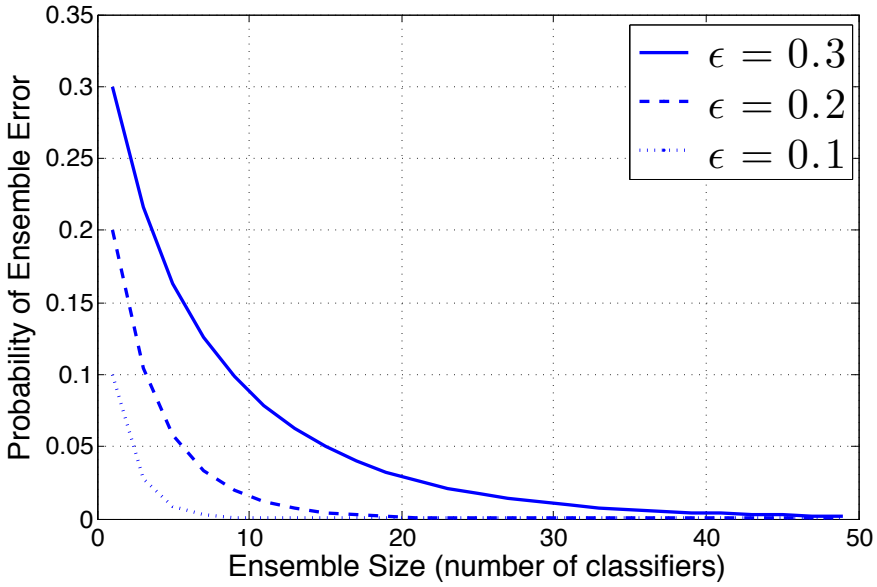


Figure 13: Theoretical majority voting error as we increase the ensemble size (number of voters), *assuming errors are statistically independent.* If each model has only 20% error rate ($\epsilon = 0.2$), the ensemble will have $< 1\%$ error if we combine 15 of them, or $< 0.1\%$ if we combine 20 of them.

We can illustrate this idea further, and plot the voting error as the number of voters increases. Figure 13 shows this, for different values of ϵ. Note that the error approaches zero as the number of voters increases.

Seems like magic, right? No — the critical assumption here is that the voters make *statistically independent* errors. An interesting question is, how realistic is this voting independence assumption — if we generate classifiers, will they be statistically independent? Probably not. Can you see why?

5.2 Training a good ensemble... is not so easy.

If we were to train two classifiers from two *identical* training sets, the only difference between the final models will emerge from their learning algorithms. If the learning algorithm was the same, then we get identical models. We don't want that in an ensemble — otherwise there's no point in having a set of predictors — we want some sort of "diversity" in their predictions. So, what if we took our training set, and *divided* it amongst the classifiers? If we had for 2000 examples, we could make two classifiers by giving them 1000 examples each. We know the usual assumption that our training data is independent and identically distributed — so surely this will result in perfectly independent classifiers? We can extend this idea – dividing the 2000 examples amongst 3, 4, or more classifiers.

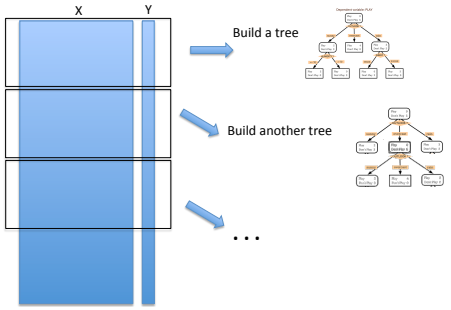


Figure 14: Should we divide the training set between our classifiers?

The results of this process are plotted below, using two different types of classifier – on the left is decision trees, and on the right, Naive Bayes. So we've supposedly supplied independent training sets – and (quite disappointingly) the error of the decision tree ensemble is remains flat as we add more trees! This should have matched the behaviour predicted in Figure 13. What went wrong?

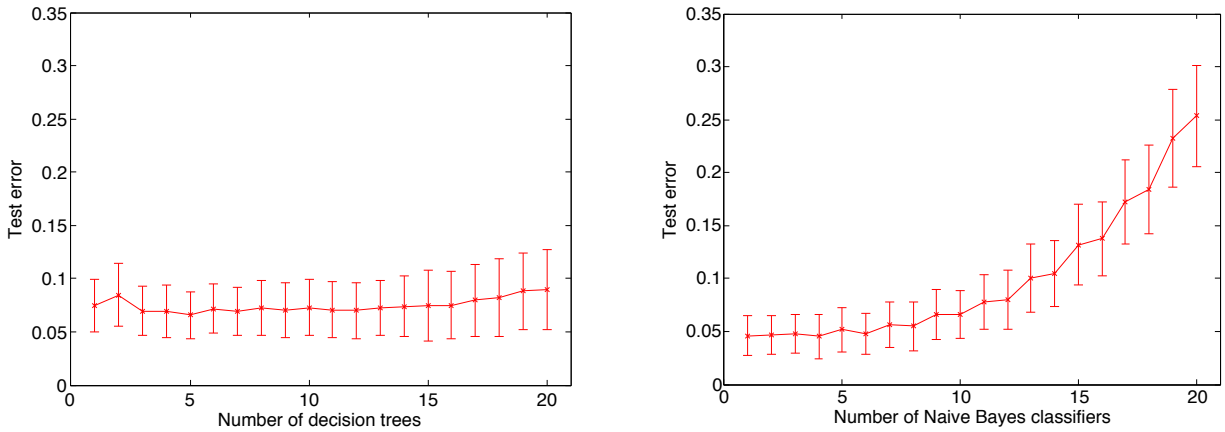


Figure 15: Splice data (3 classes, 60 features, 3175 examples). Using disjoint training sets with ensembles of decision trees (left) and Naive Bayes (right). The loss in performance due to reduced training set size outweighs the gain in performance from having an ensemble.

Remember here, that as we increase the size of the ensemble, *the size of the individual training sets necessarily decreases*, since there are more ensemble members to share the n datapoints between. Consequently, whilst the individuals are very different from each other, they become less and less accurate on testing data, as their training set size decreases. This seems to be a dilemma — decrease the overlap between training sets to make them different, but that causes them to lose accuracy. We need ways of training models such that they are both individually quite accurate, but also sufficiently different from one another that we can achieve something close to the behaviour predicted in Figure 13.

5.3 Parallel Algorithms: Bagging and Random Forests

We're now going to see two interesting algorithms that try to create this "diversity" between classifiers, whilst maintaining a reasonable degree of accuracy. They are both based around the idea of randomly perturbing the classifiers by feeding them slightly different training sets.

5.3.1 The Bagging algorithm

A data *bootstrap* is a random sample of examples from our original dataset. Given a dataset of size n, we randomly select n examples, *with replacement*. The *with replacement* part is very important. It means that we randomly pick one example to be in our training set, then put it back, ensuring that it could be picked *again*. An example dataset and two bootstrap samples are shown below.

Original data

id	x1	x2	x3	y
1	0.18	0.45	0.8	0
2	0.11	0.82	0.07	0
3	0.87	0.3	0.21	1
4	0.34	0.49	0.18	1
5	0.95	0.64	0.63	0
6	0.03	0.59	0.15	1

Bootstrap 1

id	x1	x2	x3	y
1	0.18	0.45	0.8	0
1	0.18	0.45	0.8	0
3	0.87	0.3	0.21	1
4	0.34	0.49	0.18	1
5	0.95	0.64	0.63	0
5	0.95	0.64	0.63	0

Bootstrap 2

id	x1	x2	x3	y
1	0.18	0.45	0.8	0
2	0.11	0.82	0.07	0
3	0.87	0.3	0.21	1
6	0.03	0.59	0.15	1
6	0.03	0.59	0.15	1
6	0.03	0.59	0.15	1

This shows a dataset (left) and two bootstrap samples taken from it (right). Notice that the first bootstrap (top right) contains **2** copies of the first example, but none of the second or sixth examples. The second bootstrap is generated by following the same randomised procedure, but results in a different training set—with **3** copies of the sixth example, but no copies of the fourth example.

The bootstrapping procedure generates slightly different training sets. These differences between training sets are exploited to build different models, using an algorithm — **B**ootstrap **Agg**regat**ing**, or **Bagging** (Breiman, 1996). The combination method is majority vote if we have a classification problem, or a simple average if doing a regression problem. Optionally we could use non-uniform weights; but, this would require an extra hold out dataset to optimise these weights and very often this tends to overfit.

Bagging (requires training data+labels $\mathcal{S}_n$, and choice of number of models M)

---

for $j = 1$ to M **do**
  Take a *bootstrap* sample $\mathcal{S}'_n$ from $\mathcal{S}_n$
  Build a model using $\mathcal{S}'_n$.
  Add the model to the set.
end for
return set of models
For a test point $\boldsymbol{x}$, get a response from each model, and combine predictions.

The result of this procedure varies due to the inherent randomness, and, a typical outcome is trees that look like the following:

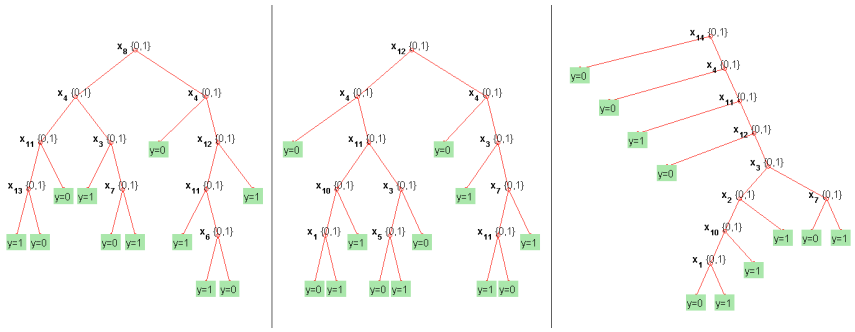


Figure 16: Typical result of Bagging: individual trees are different but still reasonably accurate.

Here, the trees are clearly not identical, and they have different splits as well as different depths. Importantly, they are not as accurate as they could have been, if they had access to all the data, but the differences between them can offset this problem, at least in theory. Let's now see how well it performs in practice. The red curve is for the ensemble using bootstraps (i.e. Bagging), while the blue line shows the performance from a single predictor using all the data.

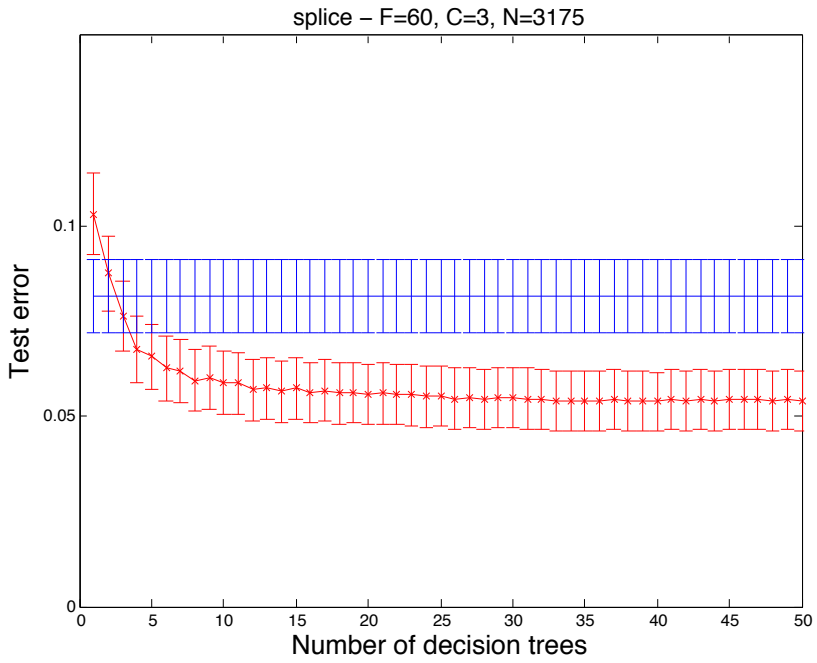


Figure 17: Bagging trees on the Splice dataset.

These results can be contrasted with the theoretical performance predicted in Figure 13. Notice that a single bagged tree (red line) here has an error of approximately 10.1%, so according to the theory prediction, by the time we get to an ensemble of size $M = 10$, we should be well under 1% error — but in fact we have about 6%. Still, this is better than a single tree working on *all* the data, shown as the blue (flat) line above. Let's examine why we might not be achieving this theoretical error.

Analyzing Bagging

Maybe the individual trees are not accurate enough? We are indeed throwing away training data in our bootstrapping procedure. Let's look closer, how much data are we discarding. Well, we know there is a uniform $\frac{1}{n}$ probability of selecting each example. Now imagine a *specific* example. I can tell you that the chances of including that particular example in a data bootstrap is given by:

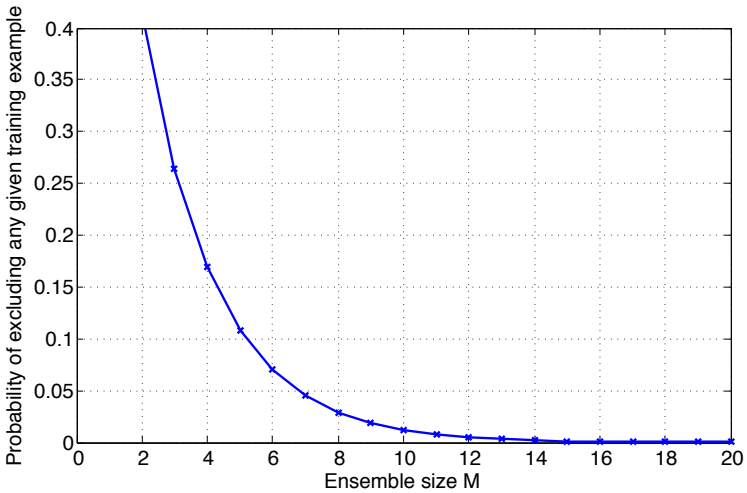
$$p = 1 - \left(1 - \frac{1}{n}\right)^n \tag{51}$$

Let's understand this. Breaking it down, if we imagine we focus on a single specific example, there is a $\frac{1}{n}$ probability of including it in a single example selection, therefore a $1 - \frac{1}{n}$ probability of *not* including it; if we repeat this, there is a $(1 - \frac{1}{n})^n$ chance of *not including* it in the sample of size n. Finally, 1 minus this quantity tells is the chance of including a single example given n repeated random selections. As the data set size increases, this converges to a probability of about 0.6321.

$$\lim_{n\to\infty} \left\{1 - \left(1 - \frac{1}{n}\right)^n\right\} = 1 - e^{-1} \approx 0.6321 \tag{52}$$

The consequence is that for any given bootstrap, it will contain only 63.2% of the original data, and the other 36.8% of examples will be excluded.

You may think this worrying that we are effectively throwing away data, but similar probabilistic arguments can be made to illustrate the chance of ignoring any given example as the ensemble size grows — i.e. what is the chance that a particular example gets completed ignored by an ensemble of size M? With This probability, of excluding a given example from the ensemble, decreases rapidly – shown below. By the time we have an ensemble of size $M = 10$, there is less than a 1% chance that any given example would have been excluded from the ensemble completely.



It appears that as soon as we have a reasonably sized ensemble, we will almost certainly be using all the training data available to us. So this does not explain the sub-optimal performance. What else can we look at? We can also look at the statistical dependence between classifiers generated. We now learn an ensemble of size $M = 50$, and plot these dependencies . This is illustrated in the figure below, where a white square indicates a statistically significant dependence between classifier i and j (χ^2 test, $\alpha = 0.05$).

In the left figure (Decision trees), every single tree is significantly correlated with at least one other! We have *completely violated* our assumption that the classifiers make independent errors, therefore we cannot possibly get to the theoretical performance predicted by Condorcet. In the second figure, we show the same for Naive Bayes classifiers — even more correlations! Why is this? The answer is that the function class explored by Naive Bayes is much more restricted than the trees — it is a 'high bias' model. Thus,

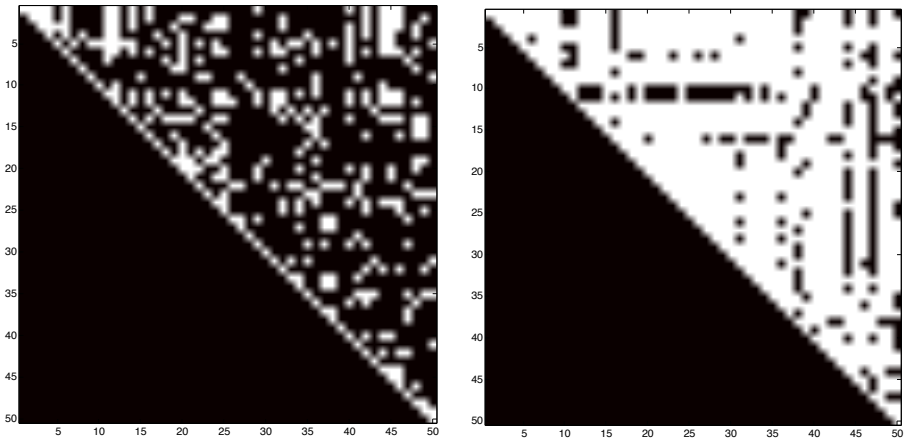


Figure 18: Dependencies between 50 bagged classifiers – trees (left), Gaussian Naive Bayes (right).

the number of possible models that can be learnt from the data is more restricted, hence we end up with near duplicate models, highly correlated. The trees have a larger function class to explore, so we end up with less correlations. Even more interestingly, we can look at the *training errors* of the trees. These are below, indicating that the trees being built from a bootstrap are in fact *overfitting*, and this is helping us!

	Single tree (all data)	Single tree from a bootstrap
Training error	5.8% (±2.9%)	3.6% (±2.1%)
Testing error	16.9% (±4.7%)	20.7% (±4.5%)

Overfitting our trees is helping us? What a paradox! But, not if you consider that it's not the individual tree that makes a prediction.... it is the ensemble. As long as the overfitting patterns of one tree is different from that of another, they will effectively cancel out each others mistakes.

So, we need a way of further reducing dependencies between our classifiers. We will stick with trees for now, and examine a very popular method which produces further differences by randomisation procedures.

5.3.2 Random Forests

Random Forests is an ensemble algorithm only for *decision trees*, hence the name—a forest of randomized trees. The randomisation is generated via *two* mechanisms: a **bootstrap**, (as in Bagging), and a **random selection of features** at each split point. The algorithm is:

Random Forests (input training data+labels $\mathcal{S}_n$, number of trees M)

---

for j = 1 to M **do**
  Take a bootstrap $\mathcal{S}'_n$ from $\mathcal{S}_n$
  Build a tree using $\mathcal{S}'_n$, but, at every split point:
    - Choose a random fraction K of the remaining features.
    - Pick the best feature from that subset.
  Add the tree to the set.
end for
return set of trees
For a test point $\boldsymbol{x}$, get a response from each tree, and take a majority vote.

Commonly, $K = \lceil \sqrt{d} \rceil$ where d is the total number of features. The trees are effectively forced to not choose the best feature at every split, but in a random way. The result is that the trees will be even more different, but still quite accurate. The majority vote ensures that the little drop in accuracy for each tree doesn't matter too much, and the differences between them ensures that they don't make simultaneous mistakes.

The performance of RF on the splice dataset from the previous subsection is shown below, up to 50 trees. Notice that whilst Bagging levelled off in performance around 5.5%, RF continues to decrease test error beyond this. The full run up to 500 trees is shown in Figure 20, zoomed in to show RF levelling off at around 150 trees, with 3.8% average test error (20x 2 fold cross validation).

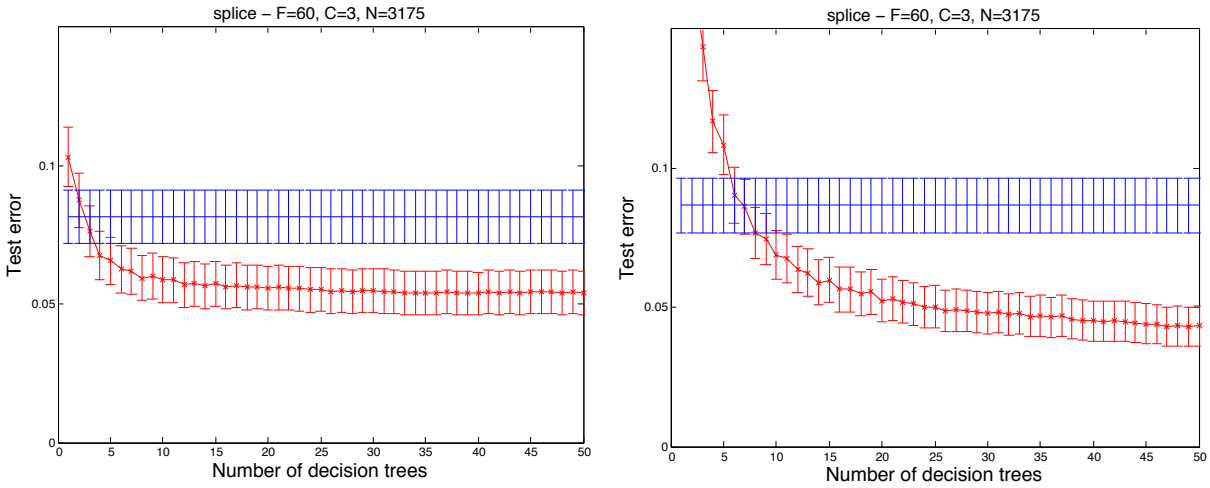


Figure 19: Bagging (LEFT) vs Random Forests (RIGHT) on the Splice dataset.

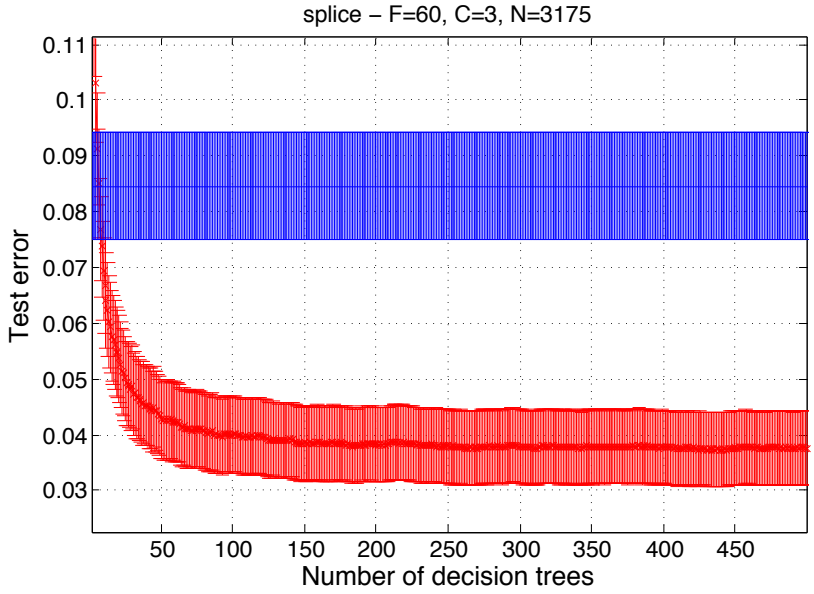


Figure 20: 500 trees on the Splice dataset.

In general RF should be thought of as a *methodology* rather than a precise algorithm. There are various ways to randomise the tree structures, and no single way is the "right" way. However a general point is that RF works best when there are a large number of features — allowing for lots of diversity in the forest.

5.4 A Sequential Algorithm: Boosting

We will now see an ensemble method which works *sequentially*—where each classifier aims to correct the errors of its predecessors. This is called a 'boosting' procedure.

5.4.1 The general idea

Each stage of the procedure involves training one new model, then identifying where it makes mistakes in the training data. Its mis-classified training examples have their emphasis increased, while the correctly classified examples have their emphasis decreased. This generates a new dataset, which is used to train the next model in the boosting chain.

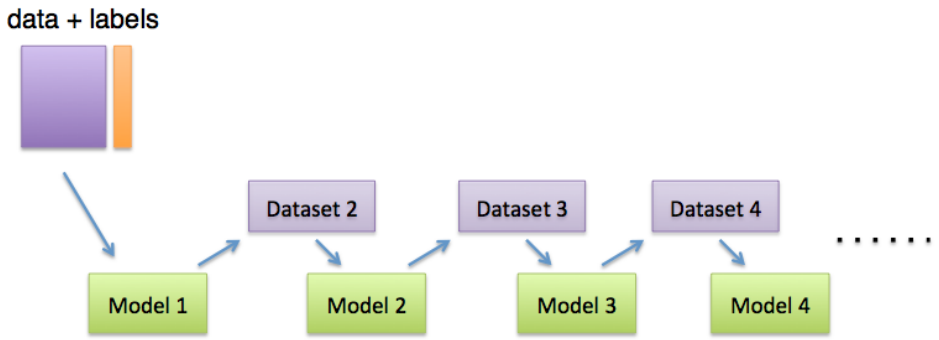


Figure 21: General principle of boosting – each model receives a dataset generated based on the errors made by its predecessor.

The emphasis placed on across the training examples is represented as a probability distribution P. An informal writing of the algorithm is below.

Boosting: input training data+labels $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$, and required number of models M

Define a distribution over the training set, $P_1(i) = \frac{1}{n}$, $\forall i$.
for $t = 1$ to M **do**
  Build a model h_t from the training set, using distribution P_t.
  Update distribution to P_{t+1} :
    Increase the weight on examples that h_t incorrectly classifies.
    Decrease the weight on examples that h_t correctly classifies.
end for
For a new testing point $\boldsymbol{x}$, we take a weighted majority vote from $\{h_1, \ldots, h_M\}$.

It is important to know that Boosting occupies somewhat of a special place in the history of ensemble methods. Though the procedure seems heuristic, the algorithm is in fact grounded in a rich body of literature from computational learning theory. In 1989, Yoav Freund and Rob Schapire addressed a question on the nature of two complexity classes of learning problems. The two classes are *strongly learnable* and *weakly learnable* problems. They showed that these classes were *equivalent*, and had the corollary that if we have a weak prediction model, performing only slightly better than random guessing, *should* be able to be "boosted" into an arbitrarily accurate *strong* model. The original Boosting algorithm (by Freund) was a proof by construction of this equivalence, though had a number of impractical assumptions built-in. Schapire and Freund later improved upon this, producing 'Adaboost', the most well known boosting algorithm. So, let's get into some details of how Adaboost works.

Adaboost (Adaptive Boosting) is the most well known of the boosting family of procedures. Adaboost is naturally a two-class classifier, and uses a particular formalism: assume each classifier is $h_t(\boldsymbol{x}) \in \{-1, +1\}$, then the decision of a weighted majority voting ensemble can be written as $H(\boldsymbol{x}) = \text{sign}(\sum_t \alpha_t h_t(\boldsymbol{x}))$. Here, α_t is the weight assigned to voter t in the weighted vote. The detailed algorithm is shown below.

Adaboost: input training data+labels $\{(\boldsymbol{x}_1, y_1)...(\boldsymbol{x}_n, y_n)\}$, and required number of models M

Define a distribution over the training set, $P_1(i) = \frac{1}{n}$, $\forall i$.
for $t = 1$ to M **do**
  Build a classifier h_t, using distribution P_t.
  Set $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  Update distribution to P_{t+1} :
    Set $P_{t+1}(i) = P_t(i)\exp(-\alpha_t y_i h_t(\boldsymbol{x}_i))$
    Renormalize: $P_{t+1} \leftarrow P_{t+1}/Z_t$.
end for

For a new testing point $(\boldsymbol{x}', y')$, we take a weighted majority vote, $H(\boldsymbol{x}') = sign\Big(\sum_{t=1}^{M} \alpha_t h_t(\boldsymbol{x}')\Big)$

The distribution P_t is used by model h_t. For models that can naturally take into account different emphases on getting certain points correct, like Naive Bayes, the model can directly use the distribution—this is called the *reweighting* method. For others, like maybe simple decision stumps, we *re-sample* n items (with replacement) from the training set according to the distribution P_t. Notice that at the first iteration when P_1 is uniform, this is *exactly* equivalent to Bagging. However once the first model is built, the distribution is updated to become non-uniform, and Bagging/Boosting diverge.

5.4.2 Deriving the Algorithm

The algorithm seems like magic: provide a base classifier just *slightly* better than random guessing, and Adaboost will boost it up into an arbitrarily strong ensemble. But where does it come from? What's so interesting about Adaboost is that it can be derived from many different perspectives. The exact same updates can be derived using game theory, or probabilistic models, or dynamical systems, and others. We will now see one of the most commonly cited derivations, the *greedy minimisation of exponential loss*.

Ideally, we would *like* to create an ensemble that minimises the *average classification error*, that is $\frac{1}{n}\sum_i \delta(H(\boldsymbol{x}_i) \neq y_i)$. However, due to the discontinuity of the δ function, this is hard. So, we will instead find an *upper bound* for the classification error, and minimise that instead. The bound we choose is exponential:

$$\frac{1}{n}\sum_{i=1}^{n} \delta(H(\boldsymbol{x}_i) \neq y_i) \leq \frac{1}{n}\sum_{i=1}^{n} \exp(-y_i \sum_{t=1}^{M} \alpha_t h_t(\boldsymbol{x}_i)) \tag{53}$$

This bound is shown in Figure 22.

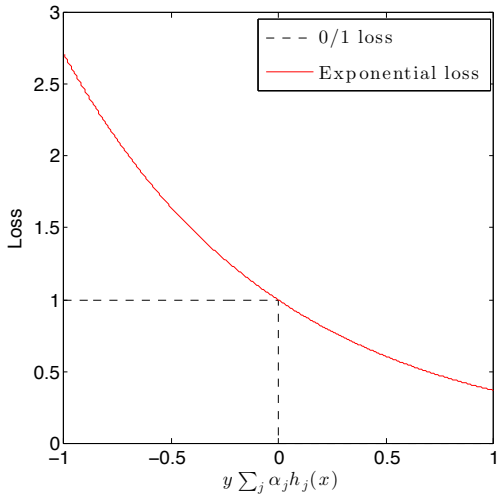


Figure 22: An exponential bound on the classification loss.

The loss function assigns a higher penalty when the ensemble is incorrect, i.e. $y \sum_t \alpha_t h_t(\boldsymbol{x}) < 0$. Since both y and h_t are in $\{-1, +1\}$, then the sign of this product indicates whether the ensemble is correct or not. Take a moment to consider this idea if you do not see it immediately. Notice that the 0/1 loss assigns a value 1 when the ensemble is incorrect, though the exponential loss applies a much harsher penalty. Notice also that when the ensemble is correct ($y \sum_t \alpha_t h_t(\boldsymbol{x}) > 0$), the exponential loss is still not *zero* — this is an important fact that brings Boosting one of its most useful properties, that we will discuss later.

We will see now how we can train an ensemble *sequentially*; i.e. we will greedily minimise the loss of our ensemble, measured by this exponential bound, at each step adding a new classifier to the ensemble. When training the first model h_1, our loss is:

$$E_1 = \frac{1}{n} \sum_{i=1}^{n} e^{-y_i h_1(\boldsymbol{x}_i)} \tag{54}$$

This loss is minimised by the single model h_t getting as many of the n data points correct as it can. Now let's add another model to our ensemble. This means our loss at time step $t = 2$ is:

$$E_2 = \frac{1}{n} \sum_{i=1}^{n} e^{-y_i \sum_{t=1}^{2} \alpha_t h_t(\boldsymbol{x}_i)} \tag{55}$$

where you should notice that we have had to introduce a parameter α_t that says how "important" model t is in the combination of the two classifiers. Intuitively, this should be set proportional to how likely each is to get a datapoint correct, but we will see exactly how to set it shortly — for now just assume it is fixed. The most pertinent point to see here is that due to the properties of the exponential, this loss function decomposes:

$$E_2 = \sum_{i=1}^{n} \underbrace{\frac{1}{n} e^{-y_i \alpha_1 h_1(\boldsymbol{x}_i)}}_{\text{constant}} e^{-y_i \alpha_2 h_2(\boldsymbol{x}_i)} \tag{56}$$

where we have marked the terms that are constant with respect to the parameters h_2, α_2, that we are now optimising. Remembering that we have *already* learnt the parameters of the model h_1, and fixed the α_1 value, we are minimising the following at stage 2:

$$E_2 = \sum_{i=1}^{n} w_2(i) e^{-y_i \alpha_2 h_2(\boldsymbol{x}_i)} \tag{57}$$

where $w_2(i) = \frac{1}{n} e^{-y_i \alpha_1 h_1(\boldsymbol{x}_i)}$. This w_2 term is the relative importance of each datapoint i, and *is constant* when we are learning the parameters of h_2. It quantifies the errors made by the previous classifier on each datapoint; if $w_2(i)$ is large (more precisely if $w_2(i) > 1$) then h_1 got datapoint i incorrect. So in order to

reduce its own loss, h_2 has to focus on data points that h_1 incorrectly classified. If we define $w_1(i) = \frac{1}{n}$ for all i — then $w_2(i)$ is a function of $w_1(i)$, that is $w_2(i) = w_1(i)e^{-y_i\alpha_2 h_2(\boldsymbol{x}_i)}$. This pattern continues when we add a third model:

$$E_3 = \sum_{i=1}^{n} \underbrace{\frac{1}{n} e^{-y_i\alpha_1 h_1(\boldsymbol{x}_i)} e^{-y_i\alpha_2 h_2(\boldsymbol{x}_i)}}_{w_3(i)} e^{-y_i\alpha_3 h_3(\boldsymbol{x}_i)} \tag{58}$$

So, we can see that this is in fact a recursive update:

$$E_3 = \sum_{i=1}^{n} \underbrace{\underbrace{\underbrace{\frac{1}{n}}_{w_1(i)} e^{-y_i\alpha_1 h_1(\boldsymbol{x}_i)}}_{w_2(i)} e^{-y_i\alpha_2 h_2(\boldsymbol{x}_i)}}_{w_3(i)} e^{-y_i\alpha_3 h_3(\boldsymbol{x}_i)} \tag{59}$$

In general form, the relative importance of each datapoint at when training the next model is:

$$w_{t+1}(i) \leftarrow w_t(i)e^{-y_i\alpha_t h_t(\boldsymbol{x}_i)} \tag{60}$$

Notice that we started with a valid distribution for w_1, that is $\sum_i w_1(i) = 1$, but this is not true as we continue adding models, so instead we will normalise the weights at each timestep — but this in no way affects the shape of the loss function. To emphasise that we are normalising the weights, we will call it a distribution, D:

$$P_{t+1}(i) \leftarrow \frac{P_t(i)e^{-y_i\alpha_t h_t(\boldsymbol{x}_i)}}{Z_t} \qquad \textit{where } Z_t = \sum_{i=1}^{n} P_t(i)e^{-y_i\alpha_t h_t(\boldsymbol{x}_i)} \tag{61}$$

which, if you refer back to the algorithm, you will find is the Adaboost update, nice huh? :-) As mentioned, this update means each classifier trains so it corrects the mistakes of its predecessors, and there are *two* things to find: the distribution on which to train each classifier, and the weights α in the majority vote. The α values can be found by considering the loss function that each model is optimising:

$$E_t = \sum_{i=1}^{n} P_t(i)e^{-\alpha_t y h_t(\boldsymbol{x}_i)} \tag{62}$$

where P_t is the distribution we set from the previous time step. By differentiating E_t with respect to α_t, setting it to zero, and solving for α_t, we find that:

$$\alpha_t^* = \arg\min_{\alpha_t} \left\{E_t\right\} = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) \tag{63}$$

where $\epsilon_t = \sum_i P_t(i)\delta(H(\boldsymbol{x}_i) \neq y_i)$, i.e. the weighted error rate of model t. So, with this, we have derived the Adaboost algorithm from scratch.

5.4.3 Summary

Boosting is an incredibly rich family of algorithms to study. There is even a paper published showing how to learn deep networks using boosting algorithms (Huang et al., 2017). One of the reasons Adaboost has been adopted so widely is its flexibility with different classifiers, and the fact that it can be *incredibly* fast when implemented. A seminal paper in computer vision by Viola and Jones (Viola & Jones, 2001) showed how to use it for real-time face detection, and as a result during the early 2000s it quickly became the standard method used in digital cameras.

In summary, ensemble methods are a great tool to have in your arsenal, and a fascinating field to study. Next week we will look deeper into the question of *why* a method works when it does — this is the issue of 'diversity' in ensembles.

6 Ensemble Theory: why do diverse opinions help?

Motivation. We saw last week a family of 'ensemble' algorithms, where we can increase the number of models, seemingly without too much overfitting. One might think that increasing the number of models (i.e. making a bigger overall model) necessarily increases complexity, but this is not the case. The mathematical framework of bias/variance (that we met in week 4) can still be used to understand ensemble algorithms, and why they succeed when they do. This is the topic of this week—the theory of ensemble learning.

Learning Objectives

By the end of this week you should be able to define and use the following concepts correctly in a conversation.

- the bias-variance-diversity trade-off,
- the centroid combiner rule,
- the nature of diversity in squared, cross-entropy, and 0/1 losses

6.1 The Bias-Variance-Diversity decomposition

Last week we met the idea of model 'ensembles', i.e. taking a *set* of models, and combining their predictions. The intuition behind this is similar to a human committee, that the committee will 'average out' the errors of the individuals. For this to happen, the ensemble must be 'diverse' in its predictions. This is an appealing **anthropomorphism**, invoking ideas like the "wisdom of the crowds". However, we can understand this formally, building on the bias/variance concepts we met a few weeks ago. The material for this week is a recent research paper:

> **A Unified Theory of Diversity in Ensemble Learning**
> Journal of Machine Learning Research (vol 24, December 2023)
> https://jmlr.org/papers/volume24/23-0041/23-0041.pdf

The mandatory readings are Sections 1-4, plus the conclusion in Section 8.

Other sections are optional, if you wish to read further.

On the following page are a few tips on reading papers, that may help with this one, or indeed any future reading you may do. These tips themselves are not examinable/assessed, but just presented to help you.

6.2 How to read a research paper

Reading a scientific paper is a very different process than reading a book, a newspaper article about science, or a popular science magazine. They have a very different structure to those other sources you may have read in the past. A typical scientific paper is structured as so:

Abstract: 150-200 words or so, explains the high level achievements and context of the research.
Introduction: Explains the context of the work in a little more detail, but not big technical things.
Background: Provides the necessary explanations and background to the problem tackled. Often very technical, and with lots of pointers to various other papers.
Methods: Explains the core technical achievement—usually very technical.
Experiments: Empirical support to the methods proposed, sometimes blended in with the methods section.
Discussion/Conclusion: Summarises limitations as well as what new challenges are raised by the work.

First, we generally don't read a scientific paper sequentially, in the order it is presented. This may seem odd, but bear with me. The first step is to read the **abstract**. This should summarise the main elements of the paper, and help you decide whether it is really relevant to your work. Then, read the **conclusions**, and to try to figure out what the authors claim they have achieved, and hopefully what evidence they have for their claim. Finally, if the abstract and conclusion have still got your interest, read the main body of the paper, starting with **background** if you're not already familiar. When you encounter bits you don't understand, maybe scribble some notes on the paper, but feel free to **skip over them**. Otherwise, you'll hit a wall and not get past it. Truly understanding a paper takes multiple passes, and can take many hours (or weeks, or months!) of dedicated time, so don't expect to read a paper in one sitting. You'll come back to those difficult bits later, and understand more.

You shouldn't just *read* a paper. You should be reading and, simultaneously, evaluating. Critical thinking is the essence of research, and involves asking yourself a series of questions as you read. I can suggest three criteria you might consider, with a set of questions for each. If you can answer these questions, you will have a deeper understanding of the paper.

Originality: What is really 'novel' in this paper? What do the authors claim they have achieved? Are they addressing a new or complex problem? Are they proposing a new approach to an old problem, or demonstrating a new capability or property that has not been seen before? Maybe they present new perspectives, arguments, or insights about an existing problem or solution? How does it relate to other papers you know? An extension or special case? Does it support other work, or contradict it?

Significance: Is this really an important problem to solve, or an important proposed solution? What is the evidence that other people really care about it? How many subsequent papers have cited[11] this one? Are the results really that surprising? Even if they are addressing an important problem—are they addressing it properly, without making too many simplifying assumptions?

Rigour: How thorough have the authors been in their investigation? Have they considered all the relevant literature—are there papers that they should have cited? Is their chain of reasoning solid—their assumptions justifiable? Are there unwritten assumptions that they may be glossing over, hoping you won't notice? Do they present sufficient evidence to really convince you of each of their claims? In simple language, how easy is it to pull a hole in their conclusions?

This has been a short set of tips that hopefully will help you. There are many other guides out there on the web, that may help in reading papers. Another good one is the following:

- How to read a paper (https://web.stanford.edu/class/ee384m/Handouts/HowtoReadPaper.pdf)

[11] You can use Google Scholar to check this. However, be aware that a large citation track takes a while to build up—papers don't generally get any citations until at least a few months to a year or more after publication.