MFM 703 and 713
Computational Finance I and II

A. Kratsios
Dept. of Mathematics and Statistics
McMaster University
Hamilton,ON, L8S 4K1

Fall 2022

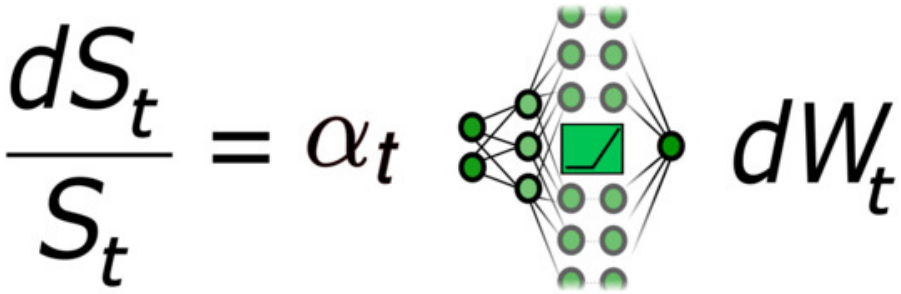

Figure 1: Photo credit due to my dear friend Dr. Wahid Khosrawi.

These notes cover the contents for both halves of the course: the fist half (MFM 703) corresponds to Sections 1 to 8, whereas the second half (MFM 713) corresponds to Section 9 onwards.

Credit: Notes are based on my own material, research with magnificent co-authors, and notes of Josef Teichmann Machine Learning in Finance.

I would like to highlight that Matheus Grasselli has written a large part of these course notes (especially on MFM 713); for which I thank him greatly :)

Warning: *Content will be progressively updated as the semester evolves and not all content covered in class appears in these notes. So be sure to follow the frequent updates!*

Contents

1 Introduction to Machine Learning **6**

1.1 Getting Started 6

1.2 Notation and terminology 7

2 The Regression Problem: with Linear Hypotheses **10**

2.1 Linear regression 10

2.1.1 The Underparameterized Regime ($d < N$); $D = 1$ Case . . 12

2.1.2 The Overparameterized Regime ($d > N$); $D = 1$ Case . . 14

2.1.3 Numerical Optimization: Solving the regression problem in high dimensions, without the Euclidean loss function . 17

2.1.4 The bias-variance trade off 19

2.2 Hyperparameter Selection: Cross Validation 23

2.2.1 Examples 24

3 The Power of Random Features **28**

3.1 Examples of (Random) Feature Maps 30

3.2 Random Neural Networks are Universal 31

3.3 Random Neural Networks Preserve Universality 32

3.3.1 Example of UAP Invariant (Random) Feature Maps . . . 32

3.4 Feature Maps which Generalize Well 33

4 Deep Neural Networks **36**

4.1 What are Neural Networks? 36

4.2 Why do we care about neural networks? 37

4.2.1 The Approximation Theory of Neural Networks 38

4.2.2 How do Neural Networks Approximate? 40

4.3 Breaking the Curse of Dimensionality: From Whitney to Your Laptop 42

4.4 Does depth really Matter? 44

4.5 Generalization/Risk Bounds For Neural Networks 45

4.6 Getting Started IRL 47

5 Classification **48**

5.1 Structure of Classes and Classifiers 49

5.1.1 Open Classes 49

5.1.2 Closed Classes 51

5.2 Borel Classes and The Bayes Classifier . . . 53
5.2.1 Full-Circle: From Bayes Classifiers to High-Probability Classifiers of Open and Closed Sets . . . 57
5.3 Logistic regression . . . 57
5.3.1 Decision boundaries . . . 59
5.3.2 Regularization . . . 59
5.3.3 Logistic regression as a neural network - binary classification 60
5.3.4 Logistic regression - multiple classes . . . 62

6 Machine Learning In Finance 66
6.1 Deep Hedging . . . 66
6.1.1 Deep Hedging exemplified by means of the Black Scholes model . . . 68
6.1.2 Code . . . 70

7 Unsupervised Learning 70
7.1 Anomaly detection . . . 70
7.2 Principal Component Analysis . . . 72
7.2.1 K-means clustering . . . 76
7.2.2 Hierarchical Clustering . . . 77

8 An introduction to Crypto Assets 79
8.1 The economics of Money . . . 79
8.2 A crash course on Cryptography . . . 80
8.2.1 Hash functions . . . 80
8.2.2 Digital Signatures . . . 84
8.3 Toy examples of cryptocurrencies . . . 85

9 Lattice Methods 88
9.1 Models for Stock Prices . . . 88
9.2 One-period binomial model . . . 89
9.3 The binomial tree . . . 91
9.4 The continuum limit and the choice of parameters . . . 93
9.5 American Options . . . 95
9.6 Further Lattice Methods . . . 96
9.7 Examples . . . 97

10 Monte Carlo Methods **102**
10.1 Introduction . . . 102
10.2 Monte Carlo Integration . . . 103
10.3 Random Number Generators . . . 104
10.4 Sampling from a given distribution . . . 105
10.4.1 Inverse Transform Method . . . 106
10.4.2 Acceptance-Rejection Method . . . 107
10.4.3 Box-Muller Method . . . 107
10.5 Monte Carlo Error Estimate . . . 108
10.6 Variance Reduction . . . 109
10.6.1 Antithetic Variates . . . 109
10.6.2 Control Variates . . . 111
10.6.3 Conditioning . . . 112
10.6.4 Computational Efficiency . . . 113
10.7 Pricing European Options . . . 113
10.8 Pricing Exotic Options . . . 114
10.8.1 Barrier Options . . . 115
10.8.2 Asian Options . . . 116
10.8.3 Lookback Options . . . 117

11 Finite-Difference Methods for Partial Differential Equations **118**
11.1 Introduction . . . 118
11.2 Errors . . . 119
11.3 Consistency, convergence, and stability . . . 121
11.4 Methods for the Heat Equation . . . 122
11.4.1 The Explicit Method . . . 122
11.4.2 The Implicit Method . . . 125
11.5 The Crank-Nicolson Method . . . 128
11.6 From Black-Scholes to the Heat Equation . . . 129
11.7 Examples of derivative pricing . . . 131
11.7.1 Explicit method for European put option . . . 131
11.7.2 Implicit method for European call option . . . 133
11.7.3 Barrier option by Crank-Nicolson . . . 133
11.8 American Options . . . 134
11.8.1 Linear complementarity and the heat equation . . . 136
11.8.2 Iterative schemes for solving $Ax = b$ . . . 138
11.8.3 Projected SOR method for American Options . . . 139

A Statistical Classifiers **140**
A.1 Naive Bayes Classifier 141
A.2 Linear and Quadratic Discriminant Analysis 144
A.3 Error Analysis 146
A.4 Examples 148

B Support Vector Machines **151**
B.1 Maximal Margin Classifier 151
B.2 Support Vector Classifier 151
B.3 Support Vector Machines 153
B.4 Relationship with Logistic Regression 153

C Some Notes on Neural Networks in Practice **154**
C.1 Binary classification with one hidden layer 154
C.2 Multiple classes with one hidden layer 156

1 Introduction to Machine Learning

1.1 Getting Started

Let's install Python, anaconda, jupyternotebook, etc ...! I'm working on my Desktop *(which I can get to by entering "cd Desktop").*

```
# Move to directory (set up a GitHub!)
cd Documents/GitHub/Teaching_MachineLearningFinance
conda install -c anaconda python-editor
pip3 install jupyterlab
pip3 install -U numpy
pip3 install -U matplotlib
pip3 install -U scipy
pip3 install -U scikit-learn scipy matplotlib
pip3 install -U scikit-learn
pip3 install -U sklearn
pip3 install -U seaborn
pip3 install -U pandas
pip3 install -U LibTorch
pip install -U pipe
```

Let's open a notebook and get coding!

```
jupyter notebook
```

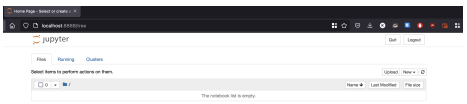

Figure 2: Let's Open Our First Jupyter Notebook

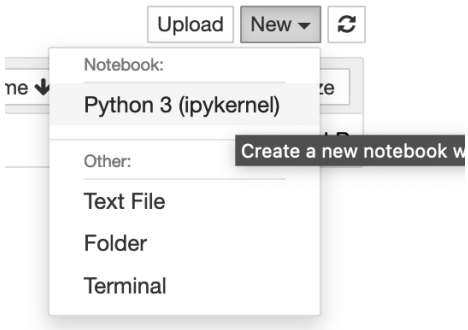

Figure 3: Let's Open Our First Jupyter Notebook

Figure 4: Let's Open Our First Jupyter Notebook

Let's import some packages

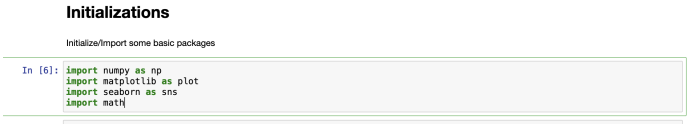

Figure 5: Let's Import some basic packages

1.2 Notation and terminology

We first provide a list of terminologies common in machine learning and used in these lecture notes.

- Input space: $\mathcal{X}$ a set of with distance function $d_{\mathcal{X}}$,
  - Until specified otherwise, we take $\mathcal{X}$ to be a (non-empty) subset of $\mathbb{R}^d$ with distance function $d_{\mathcal{X}}(x, y) := \|x - y\|_2$,

- Output space: $\mathcal{Y}$ a set with distance function $d_{\mathcal{Y}}$
  - Until specified otherwise, we take $\mathcal{Y}$ to be a (non-empty) subset of $\mathbb{R}^D$ with distance function $d_{\mathcal{X}}(x, y) := \|x - y\|_2$.
- Inputs/covariates x: The raw data/inputs; these are elements of $\mathcal{X}$.
- Outputs/targets/labels y: The output value assigned to each example in the problem; these are elements of $\mathcal{Y}$,
- Features: The input variables attributed to a single example and often represented by a real-valued vector.
- Training set $\mathcal{D}$: A set of (input, output) pairs used to train the model.
- Test set $\mathcal{D}'$: The set of examples used to evaluate the performance of a model.
- Probability distribution $\mathbb{P}$: The *true* underlying (population) distribution that samples in $\mathcal{D}$ and $\mathcal{D}'$ are drawn from. $\mathbb{P}$ is a probability distribution on $\mathcal{X} \times \mathcal{Y}$.
- Expectation $\mathbb{E}_{(X,Y)\sim\mathbb{P}} := \mathbb{E}_{\mathbb{P}} := \mathbb{E}$: Expected value over the distribution of $(x, y) \sim \mathbb{P}$.
- Expectation $\mathbb{E}_{\mathcal{D}} := \mathbb{E}_{\mathbb{P}\otimes\cdots\otimes\mathbb{P}}$: Expected value over the joint distribution of the N (random) datapoints $(x_1, y_1), \ldots, (x_N, y_N)$ which make up the training set $\mathcal{D}$.
- Predictor/model f: A function that maps input covariates to output labels; i.e. $f : \mathcal{X} \to \mathcal{Y}$,
- Hypothesis class $\mathcal{H}$: A (non-empty) collection of predictors (i.e. $\mathcal{H}$ is a non-empty set of functions from $\mathcal{X}$ to $\mathcal{Y}$),
- Evaluation metric/cost ℓ: This is the cost related directly to the application at hand.
- (pointwise) loss function ‘: A function that measures the distance between the predicted label and the true label of an example. This loss is used as a surrogate during training. For the purpose of this note, the metric is equal to the pointwise loss.
- Population risk $\mathcal{R}$: Expected loss over the probability distribution ¶.

- Empirical risk $\hat{\mathcal{R}}_{\mathcal{D}}$: Average loss on the training set D.
- Training loss: Consists of empirical risk plus potential penalties.
- Hyperparameters: Free parameters in the model that should be chosen beforehand,
- $\mathbb{N} := \{0, 1, \dots\}$ the non-negative integers,
- $\mathbb{N}_+ := \{1, 2, 3, \dots\}$ the positive integers,
- p-norm on $\mathbb{R}^d$: $\|u\|_p := \sqrt[p]{\sum_{n=1}^{d} u_i^p}$; where $p \in [1, \infty)$.

Machine learning is the field of study of computer programs that *learn* to perform a task T based on experience E, in the sense that a predetermined performance measure P improves with experience. Examples include deciding whether an email message is spam, recognizing an image, predicting the price of a house based on its characteristics, or sorting news into similar topics.

It is common to divide machine learning problems into *supervised* or *unsupervised* learning, depending on whether or not the data used by the computer program includes clearly identified *outputs* based on one or more *inputs*. From the examples mentioned above, spam filtering, image recognition, and house price prediction consist of supervised learning problems, whereas grouping news items into similar topics corresponds to an unsupervised learning problem.

We begin by discussing the (probabilistic) supervised learning; in which case we wish to "best" identify a *predictor* (function) $f : \mathcal{X} \to \mathcal{Y}$ given data $\mathcal{D} := \big\{(x_n, y_n)\big\}_{n=1}^{N}$ (where $N \in \mathbb{N}_+$). We quantify "best" via a (suitably regular) loss function $\ell : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} \to \mathbb{R}$ evaluated on together with our training dataset $\mathcal{D}$ encoded into a numerical quantity called the *empirical risk* and defined for any hypothesis f in our assumed *hypothesis class* $\mathcal{H}$ by

$$\hat{\mathcal{R}}_{\mathcal{D}}(f) := \frac{1}{N} \sum_{n=1}^{N} \ell\big(f(x_n), y_n, f\big).$$

Our "best guess" for the predictor f from within our assumed *hypothesis class* $\mathcal{H}$ is denoted by $\hat{f}$; $\hat{f}$ is obtained by optimizing the empirical risk $\hat{\mathcal{R}}_{\mathcal{D}}$

$$\hat{f} \in \operatorname*{argmin}_{f \in \mathcal{H}} \hat{\mathcal{R}}_{\mathcal{D}}(f).$$

Supervised learning problems can be further subdivided into *regression* or *classification* problems, depending on whether the outputs are numerical values

or categorical types. That is, in a classification problem (for now) the output space $\mathcal{Y}$ is discrete; i.e.

$$\mathcal{Y} = \{0, \dots, C\} \tag{1}$$

where C is a positive integer denoting the *number of classes* which we want to classify any input $x \in \mathcal{X}$ into. From the examples above, predicting the price of a house is a regression problem, whereas spam filtering and image recognition are classification problems. If $\mathcal{Y}$ is not discrete of the form (1) then we the supervised learning problem of best identifying f by optimizing the empirical risk $\hat{\mathcal{R}}_{\mathcal{D}}$ is called *regression*.

Remark 1.1. *For the majority of this course, and until otherwise specified, we will assume that $\emptyset \neq \mathcal{X} \subseteq \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}^D$ for some positive integers d, D.*

Arguably, the ultimate goal of *machine learning*, is to perform best on unseen data using a model trained on seen data in $\mathcal{D}$. That is, a machine learner seeks to optimize the *true/population risk*

$$\mathcal{R}(f) := \mathbb{E}_{(x,y)\sim\mathbb{P}}\big[\ell(f(x), y, f)\big]$$

over the assumed hypothesis class $\mathcal{H}$.

Each of the problems above can be tackled by many different methods or algorithms, including linear and logistic regression, neural networks, clustering algorithms, support vector machines, principal components analysis, and many others. No algorithm can be applied to all problems, and each has their advantages and disadvantages. In these notes, we will focus on the most widely used methods, with particular emphasis on their applications in finance.

2 The Regression Problem: with Linear Hypotheses

In this section, we will familiarize ourselves with two of the main supervised learning problems; namely, regression and classification. We focus on linear methods.

2.1 Linear regression

Linear regression is the most classical example of what is called *statistical learning*. In general, given an output y, inputs $x = (x_1, x_2, \dots, x_p)$ and a relationship

of the form

$$y = f(x) + \epsilon, \tag{2}$$

for some fixed but unknown function f and a random error ϵ, which is assumed to be independent of x and with zero mean, statistical learning then consists of estimating the function f. Thus, in learn regression our *hypothesis class* consists of all affine ("linear") models from $\mathbb{R}^d$ to $\mathbb{R}^D$ and is given by

$$\mathcal{H} := \big\{x \mapsto Ax + b : A \text{ a } D \times d\text{-matrix}, b \in \mathbb{R}^D\big\}. \tag{3}$$

Example 2.1. *If $D = 1$ then $\mathcal{H} = \{f(x) := w^\top x + b : w \in \mathbb{R}^d$ and $b \in \mathbb{R}\}$. Equivalently, in this setting, any hypothesis f is expressed as*

$$f(x) = w_1x_1 + w_2x_2 + \cdots w_px_p + b \tag{4}$$

In the jargon of machine learning, the inputs x_i are called features*, the output y is called a* label*, the coefficients w_i are called* weights *and the coefficient b is called the* bias*. It is common practice to denote the weights as row vectors and the features as column vectors, that is,*

$$\mathbf{w} = [w_1, w_2, \ldots, w_p] \qquad \text{and} \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix},$$

in which case (5) *can be written as*

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w}\mathbf{x} + b, \tag{5}$$

where $\mathbf{wx}$ denotes matrix multiplication of the $1 \times p$ row vector $\mathbf{w}$ by the $p \times 1$ column vector $\mathbf{x}$.

In (classical) linear regression, we seek to optimize the Euclidean loss

$$\ell(x, y, f) := \|f(x) - y\|_2.$$

This means that the empirical and true risks are respectively given by

$$\hat{\mathcal{R}}_{\mathcal{D}}(f) = \frac{1}{N}\sum_{n=1}^{N} \|f(x_n) - y_n\|_2 \text{ and } \mathcal{R}(f) = \mathbb{E}_{(x,y)\sim\mathbb{P}}\big[\|f(x) - y\|_2\big].$$

In the *under-parameterized case*; i.e. when there are fewer model parameters than their are datapoints (under mild conditions) the we may obtain a closed-form optimal solution of the empirical risk $\hat{\mathcal{R}}_{\mathcal{D}}$ over $\mathcal{H}$. To concisely describe this, let us introduce some notation.

If we arrange the labels as the $1 \times n$ row vector $\mathbf{Y} = [y_1, \ldots, y_n]$ and the inputs as the $p \times n$ matrix

$$\mathbf{X} = [\mathbf{x}_1, \cdots, \mathbf{x}_n] = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ x_{p1} & x_{p2} & \cdots & x_{pn} \end{bmatrix}, \tag{6}$$

where x_{ij} denotes the i-th feature for the j-th data sample, it is a straightforward result in linear algebra to show that the minimizer for the empirical risk is given by the solution of the *normal equations*

$$\begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix} \begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix}' \begin{bmatrix} \mathbf{w}' \\ b \end{bmatrix} = \begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix} \mathbf{y}', \tag{7}$$

where $\mathbf{1}_n = [1, 1, \ldots, 1]$ is the $1 \times n$ identity vector. Solving this system in the under-parameterized regime yields the following.

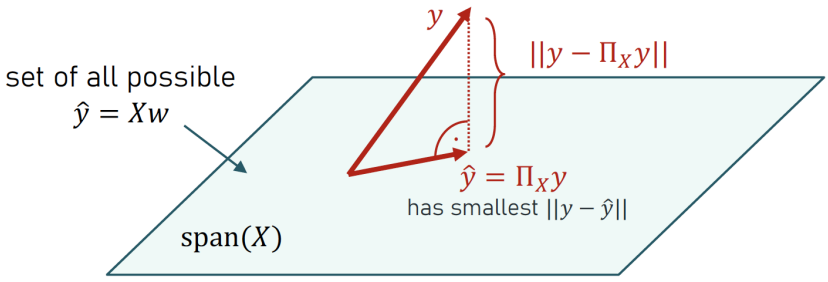

Figure 6: Optimal Linear Regressor (Under-Parameterized Regime)
Taken from Andrea Krause' Notes on Introduction to Machine learning.

2.1.1 The Underparameterized Regime ($d < N$); $D = 1$ Case

We begin by examining the case where the normal equation (6) admits a unique solution. This can only happen if the model has fewer parameters (d) than there are datapoints N.

Proposition 2.2 (Closed-Form Solution: Under-parameterized Linear Regression)**.** *Let $D = 1$, $d \leq N$, and $\mathbf{X}$ be a full-rank tall matrix. Then, the solution to the normal equation* (7) *is given by*

$$\hat{w} := (\mathbf{X}^\top \mathbf{X})^{-\mathbf{1}} \mathbf{X}^\top \mathbf{y}.$$

Moreover, the optimal (empirical) predictor in the hypothesis class (3) *is given by*

$$\hat{f}(x) := \hat{w}^\top x$$

and $\hat{w}$ is the orthogonal projection of $\mathbf{y}$ onto the span of the column space of $\mathbf{X}$ spanned by the vectors $\{x_1, \ldots, x_N\}$ (see Figure 6.)

Why is Overparameterization Typical? (An Example)

Often, in practice, machine learners "extract more information/features" from any set of input data $x_1, \ldots, x_N$ by mapping the data to a *higher-dimensional* "feature space" $\mathbb{R}^F$ ($F \gg d$) and then linear methods are applied therein. Summarized by the (commutative) diagram in Figure 7.

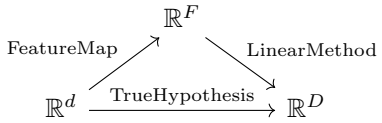

Figure 7: Feature expansion.

We will come back to this problem latter, but for the moment, consider the case where $d = D = 1$. A simple, classical, and not very recommended (imo) *feature map* ϕ, giving us a *polynomial feature representation* is given by

$$\begin{aligned} \phi : \mathbb{R}^1 &\to \mathbb{R}^{1+p} \\ x &\mapsto (1, x, \ldots, x^p), \end{aligned}$$

for a fixed positive integer p. Performing your **linear** regression, not on the data $x_1, \ldots, x_N$ in $\mathbb{R}$, but on their (polynomial) features $\phi(x_1), \ldots, \phi(x_N)$ in $\mathbb{R}^{p+1}$ we

find that our linear hypothesis class has become non-linear an *high-dimensional*

$$w^\top \phi(x) = \sum_{i=0}^{p} w_i x^i.$$

Since p is a *hyperparameter* which we can control, then, by making p very large (so that we can capture many non-linearity in the *True Hypothesis*) we can easily achieve overparameterization; i.e.

$$F = 1 + p > N \text{ for p large enough} \dots$$

We will rigorously come back to feature maps later in these notes.

2.1.2 The Overparameterized Regime ($d > N$); $D = 1$ Case

When have more parameters than the number of examples (aka over-parameterized regime), the equation $y = \mathbf{X}\mathbf{w}$ has infinitely many solutions, and all of them are valid answers to the ERM problem. This is because there are enough parameters to achieve zero empirical risk!

In this case, we *need* to select a "best" solution amongst *all* the candidate solution and this can be done by modifying the loss function. To motivate our modifications, we can choose to instead solve

$$\begin{aligned} &\underset{f_w \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \|f_w(x_n) - y_n\|_2 \\ &\text{s.t. } \|w\|_2 \leq c; \end{aligned} \tag{8}$$

where $c > 0$ is a fixed constant dictating the "maximum distortion" our linear regressor is allowed to apply to its inputs.

That is, we can seek the solution with which the least steep slope amongst all *valid optimal linear regression models*. Using Lagrange duality, it is easy to see that the *constrained optimization problem* (8) is equivalent to the *unconstrained optimization problem* seeking to optimize the loss

$$\underset{f \in \mathcal{H}}{\operatorname{argmin}} \ \frac{1}{N} \sum_{n=1}^{N} \|wx_n - y_n\|_2 \ + \lambda \|w\|_2, \tag{9}$$

for some appropriate *hyperparameter* $\lambda > 0$ (depending only on c). That is, (9)

is an empirical risk minimization problem for the *ridge regression loss* function

$$\ell(x, y, f_w) := \|f_w(x) - y\|_2 + \lambda \|w\|_2, \tag{10}$$

where we use the notation $f_w(x) := w^\top x$.

The empirical risk minimization problem with respect to the *ridge loss function* (defined in (10)) admits a closed-form expression for all $\lambda > 0$; even in the over parameterized regime.

Proposition 2.3 (Closed-Form Solution for Ridge Regressor; $D = 1$ [11, Theorem])**.** *Let $D = 1$; then the empirical risk-minimizer of* (9) *is given by*

$$\begin{aligned} \hat{f}(x) &\mapsto \hat{w}_\lambda x \\ \hat{w}_\lambda &:= \left[(\mathbf{X^T X})^{-1}\mathbf{X} + \lambda \mathbf{I}\right]^\top \mathbf{y}. \end{aligned}$$

Proof. See this Cross-Validated Post [32]. □

Nevertheless, the ridge loss (defined in (10)) can be problematic since it overemphasizes large errors made by our model, and consequentially it can lead to over emphasizing outliers. This naturally, leads to a poorly behaved predictor.

This can be corrected by replacing the Euclidean distance $\|\cdot\|_2$ in (8) by the Huber function

$$\text{Huber}(y_1, y_2) := \begin{cases} \frac{1}{2}\|y_1 - y_2\|_2^2 & : \text{ if } \|y_1 - y_2\|_2 \leq 1 \\ \|y_1 - y_2\|_2 - \frac{1}{2} & : \text{ else .} \end{cases}$$

The Huber loss is illustrated in Figure 8.

```
Huber = function(x, a=1) {
  ifelse(abs(x) <= a, x^2, 2*a*abs(x)-a^2)
}
```

Thus, with the Huber loss function the constrained (overparameterized) re-

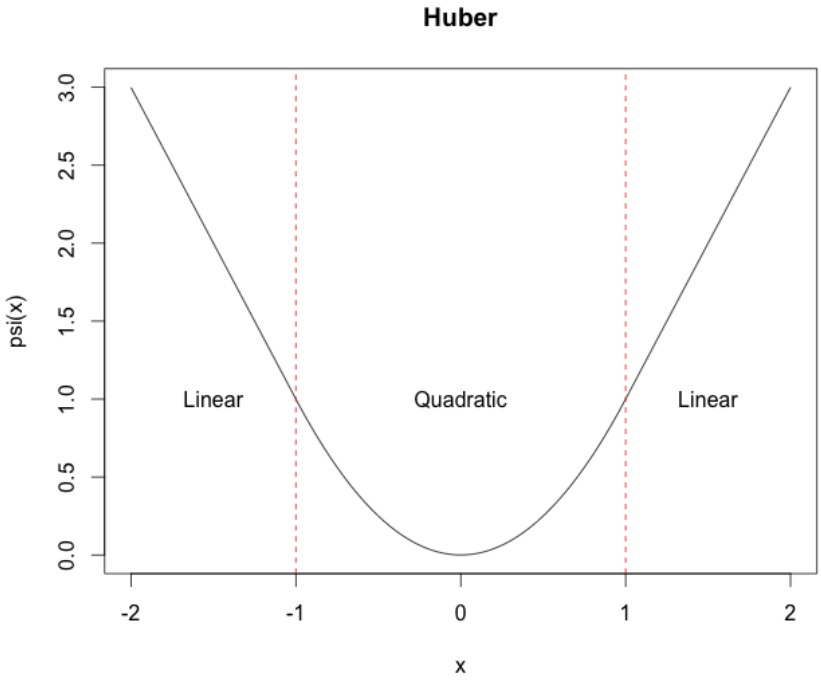

Figure 8: Clipped (Huber) loss function under-emphasizing outliers.

gression problem (8) can be re-written as

$$\begin{aligned} &\underset{f_w \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{N} \sum_{n=1}^{N} \text{Huber}\left(f_w(x_n), y_n\right) \\ &\text{s.t. } \|w\|_2 \leq c; \end{aligned} \tag{11}$$

Naturally, this begs the question:

What is the benefit of overparameterization, since we loose a unique solution?

One possible answer to this question is given by this course's first *risk-bound*; which relies on the ridge formulation of the regression problem (i.e. the overparameterized regime with *positive* hyperparameter λ).

Theorem 2.4 (PAC Bound [15, Corollary 4, Example 3.1])**.** *Let $D = 1$, a "uniform bound on the input space" $r > 0$, suppose that $\mathcal{X} = \{x \in \mathbb{R}^d : \|x\|_2 \leq$*

r} *and fix a "confidence level"* $0 < \delta \leq 1$. *Let* $\hat{f}$ *solve* (11). *Then* $\hat{f}$ *satisfies the Risk-Bound*

$$\mathbb{P}\left(\mathcal{R}(\hat{f}) \leq \hat{\mathcal{R}}_{\mathcal{D}}(\hat{f}) + \frac{2rc}{\sqrt{N}}\left(1 + 2^{-1/2}\sqrt{\log(1/\delta)}\right)\right) \geq 1 - \delta.$$

Proof. Follows from [15, Corollary 4], [15, Example 3.1], and [4, Problem 9.2] (see this post [14] for a solution to Problem 9.2). □

The significance of Theorem 2.4 is that is allows us to give estimates on how *confident we are* that our prediction will remain valid on unseen data; given what it has encountered on *seen data*. At a high-level is our first "machine learning analogue" of classical ideas like confidence intervals where for statistical estimation.

As we move away from simple models optimizing simple loss functions, we may have to give up the ability to have formulae (i.e. closed-form expressions) for our optimal models. This is for example the case with (**??**), where it is more convenient to numerically *approximately obtain* $\hat{f}$ of Theorem (2.4) rather than try to elicit it by hand. Moreover, this is doubly true in the "high-dimensional setting" when the number of our covariates d is "much greater" than 1.

2.1.3 Numerical Optimization: Solving the regression problem in high dimensions, without the Euclidean loss function

Unfortunately, computing optimal regressor $\hat{w}$ is troublesome since it involves inverting the matrix $\mathbf{X}^\top\mathbf{X}$. This is not an issue when N is small; however, when N is large this can be computationally costly. This is because, inverting this matrix takes $O\big(N^{2.373}\big)$ [2] matrix operations (currently) and each subsequent matrix multiplication required to construct $\hat{w}$ takes $O\big(N^{2.3728596}\big)$ elementary operations [2] (currently).

This system of $(1+p)$ equations is relatively easy to solve numerically, even when the number of data points N is large, for example by using an LU factorization algorithm for the $(1+p)\times(1+p)$ matrix appearing on the left-hand side of (7). This *direct* method is not, however, the approach that we will follow, as it becomes very slow when p is large and does not generalize to other machine learning algorithm. Instead, we will use an *iterative* method to approximate the minimizer of the empirical risk. For simplicity, we consider the "vanilly" linear

regression problem of (4) and, we abbreviate

$$\hat{\mathcal{R}}_{\mathcal{D}}(\mathbf{w}, \mathbf{b}) := \hat{\mathcal{R}}_{\mathcal{D}}(\mathbf{f}_{\mathbf{w},\mathbf{b}})$$

where $f_{w,b}(x) := \mathbf{w}^\top \mathbf{x} + \mathbf{b}$.

The most classical/simplest iterative method for minimizing a cost function is the *gradient descent method*. It consists of starting with an initial guess $(\mathbf{w}^{(0)}, b^0)$ and subsequently updated the parameters according to

$$\begin{aligned}
w_1^{(k+1)} &= w_1^{(k)} - \alpha \frac{\partial}{\partial w_1} \hat{\mathcal{R}}_{\mathcal{D}}(w_1^{(k)}, w_2^{(k)}, \ldots, w_p^{(k)}, b) \\
w_2^{(k+1)} &= w_2^{(k)} - \alpha \frac{\partial}{\partial w_2} \hat{\mathcal{R}}_{\mathcal{D}}(w_1^{(k)}, w_2^{(k)}, \ldots, w_p^{(k)}, b) \\
&\vdots \\
w_p^{(k+1)} &= w_p^{(k)} - \alpha \frac{\partial}{\partial w_p} \hat{\mathcal{R}}_{\mathcal{D}}(w_1^{(k)}, w_2^{(k)}, \ldots, w_p^{(k)}, b) \\
b^{(k+1)} &= b^{(k)} - \alpha \frac{\partial}{\partial b} \hat{\mathcal{R}}_{\mathcal{D}}(w_1^{(k)}, w_2^{(k)}, \ldots, w_p^{(k)}, b),
\end{aligned}$$

for a constant α called the *learning rate*. Using the definition of empirical risk, we find that

$$\begin{aligned}
\frac{\partial}{\partial w_i} \hat{\mathcal{R}}_{\mathcal{D}}(w_1, w_2, \ldots, w_p, b) &= \frac{1}{n} \sum_{j=1}^{n} (y_j - f_{\mathbf{w},b}(\mathbf{x}_j)) x_{ij} = \frac{1}{n} \sum_{j=1}^{n} (\mathbf{w}\mathbf{x}_j + b - y_i) x_{ij} \\
\frac{\partial}{\partial b} \hat{\mathcal{R}}_{\mathcal{D}}(w_1, w_2, \ldots, w_p, b) &= \frac{1}{n} \sum_{j=1}^{n} (y_j - f_{\mathbf{w},b}(\mathbf{x}_j)) = \frac{1}{n} \sum_{j=1}^{n} (\mathbf{w}\mathbf{x}_j + b - y_j)
\end{aligned}$$

Using the matrix notation introduced earlier, the cost function and the iterations for gradient descent can be rewritten in vectorized form as

$$\hat{\mathcal{R}}_{\mathcal{D}}(\mathbf{w}, b) = \frac{1}{2n} (\mathbf{w}\mathbf{X} + b\mathbf{1}_n - \mathbf{Y})(\mathbf{w}\mathbf{X} + b\mathbf{1}_n - \mathbf{Y})' \tag{12}$$

$$[\mathbf{w}^{(k+1)}, b^{(k+1)}] = [\mathbf{w}^{(k)}, b^{(k)}] - \frac{\alpha}{n} (\mathbf{w}\mathbf{X} + b\mathbf{1}_n - \mathbf{Y}) \begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix}' \tag{13}$$

The iterations above are repeated until a given tolerance is reached (for example, a sufficiently small improvement in the cost function), at which point the algorithm is deemed to have converged to the minimizers $\widehat{\mathbf{w}}$ and $\widehat{b}$.

Remark 2.5. *To avoid slowing down the gradient descent algorithm, it is im-*

portant to pre-process the features by performing what is called feature scaling*, namely subtracting the mean from each feature and dividing it by its range. In the sequel we assume that all features have been scaled in this way.*

Having found the minimizers $\widehat{\mathbf{w}}$ and $\widehat{b}$, we can predict the output $\widehat{y}$ for any input $\mathbf{x}$ according to

$$\widehat{y} = f_{\widehat{\mathbf{w}},\widehat{b}}(\mathbf{x}) = \widehat{\mathbf{w}}\mathbf{x} + \widehat{b}. \tag{14}$$

2.1.4 The bias-variance trade off

Set $D = 1$. Let us consider the special situation where

$$y = f^\star(x) + \epsilon$$

and ϵ is a random variable independent of the random (input) variable x having

$$\text{Mean:}\mathbb{E}[\epsilon] = 0 \text{ and Variance:}\mathbb{E}[\epsilon^2] = 0.$$

Denote the *law* of ϵ by $\mathbb{Q}$.

With this *structure on* the joint law $\mathbb{P}$ we can begin to interpret the effects of each parameter in the *ridge regression problem* via the conditional laws of x, y, and of the *noise* ϵ.

We find that for any $f \in \mathcal{H}$, the true (ridgeless; i.e. $\lambda = 0$) risk is given by

$$\begin{aligned}
\mathcal{R}(\hat{f}) &= \mathbb{E}_{x,\mathbb{Q}}\big[(y - f(x))^2\big] \\
&= \mathbb{E}_{x,\mathbb{Q}}\big[(y - f(x))^2 + \epsilon\big] \\
&= \mathbb{E}_x\big[(y - f(x))^2\big] + \sigma^2 + 2\mathbb{E}_x\big[f^\star(x) - f(x)\big]\mathbb{E}_{\mathbb{Q}}[\epsilon] \\
&= \mathbb{E}_x\big[(y - f(x))^2\big] + \sigma^2,
\end{aligned}$$

where we have repeatedly used the independence between ϵ and x. Consider the $\hat{f}$ optimizing the ridge regression loss function (9). Since f depends on the *random* training data $\mathcal{D}$ then the true risk $\hat{R}(\hat{f})$ of $\hat{f}$ is a random variable

depending on $\mathcal{D}$. Thus, we may compute

$$\begin{aligned}\mathbb{E}_{\mathcal{D}}[\mathcal{R}(\hat{f})] =& \mathbb{E}_{\mathcal{D}}\left[\left((f^\star(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)]) + (\mathbb{E}_{\mathcal{D}}[\hat{f}(x)] - \hat{f}(x))\right)^2\right] + \sigma^2 \\ =& \underbrace{\mathbb{E}_x\left[(f^\star(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2\right]}_{\text{Bias}} \\ &+ \underbrace{\mathbb{E}_x\left[\left(\mathbb{E}_{\mathcal{D}}[\hat{f}(x)] - \hat{f}(x)\right)^2\right]}_{\text{Variance}} \\ &+ \underbrace{\sigma^2}_{\text{Noise}},\end{aligned} \tag{15}$$

where the last equality follows from the fact that $(f^\star(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2$ is independent of $\mathbb{P}$. The computation in (15) expresses the *expected* population risk $\mathbb{E}_{\mathcal{D}}[\mathcal{R}(\hat{f})]$ into three "parts":

1. The **Bias** term: capture's the model $\hat{f}$'s expressivity,
2. The **Variance** term quantifies how robust our modeling choice $\hat{f}$ is with respect to perturbations in the dataset $\mathcal{D}$.
3. The **Noise** term: cannot be set to 0 even if $\hat{f} = f^\star$ (which is almost never the case).

The linear regression algorithm we just described is extremely versatile and can accommodate very complicated relationships between inputs and outputs. For example, nonlinear functions of the original inputs $x = (x_1, x_2, \ldots, x_p)$ can be easily incorporated into the problem simply by creating new features, say of the form x_1^2, x_1^3, $x_1 x_2$, or other arbitrarily complicated functions. The essential point is that the relationship f between inputs and outputs is linear with respect to the parameters $\mathbf{w}$ and b.

Adding new features, especially higher order polynomials of existing features, has the appeal of drastically reducing the cost function J. This is misleading, however, as what is being reduced is the cost function over the *training dataset* only, namely the data points included in $\mathbf{X}$ and $\mathbf{Y}$ above, which are used in the optimization leading to the minimizers $\widehat{\mathbf{w}}$ and $\widehat{b}$. A better way to assess the performance of the algorithm is to use a different (often smaller) dataset called the *validation dataset* consisting of inputs $\mathbf{X}^{val}$ and outputs $\mathbf{Y}^{val}$ that have not been used in the optimization procedure. One can then compute the

cost function over the validation dataset as

$$\mathcal{R}_{\text{val}}(\hat{f}) := \frac{1}{N_{\text{val}}} \sum_{n=1}^{N} \ell(x_n^{\text{val}}, y_n^{\text{val}}, \hat{f}) \tag{16}$$

where $\{(x_n^{\text{val}}, x_n^{\text{val}})\}_{n=1}^{N_{\text{val}}}$ is a set of N_{val} draws from $\mathbb{P}$ (independent of the training data $\mathcal{D}$) called the *validation set*. The quantity $\mathcal{R}_{\text{val}}(\hat{f})$ is called the *validation risk* or *validation error*.

It can be shown that the true risk function depends on two properties of the model used in the approximation: the bias and the variance. The *bias* of an approximation is the error arising by replacing a complicated real-life relationship with a much simpler model, whereas the *variance* of an approximation refers to how much it would change if we estimated it using a different training dataset. Although each of these concepts can be given a precise mathematical definition, it should be intuitively clear that, generally speaking, more complex models tend to have lower bias and higher variance.

Because the true risk function increases with *both* the bias and the variance of an approximation, ideally we should aim at a model with low bias and low variance. Since the bias is bounded from below (by zero), as the complexity (or flexibility) of the model increases, the variance inevitably takes over. An ideal model for a given problem will be one with enough complexity to have a sufficiently low bias, but not enough to make the variance too big.

Comparing the training cost function $\hat{\mathcal{R}}_{\mathcal{D}}(\widehat{\mathbf{w}}, \widehat{b})$ with the true risk function $J^{val}(\widehat{\mathbf{w}}, \widehat{b})$ can help us determine whether an approximation suffers from high bias or high variance. Namely, if $\hat{\mathcal{R}}_{\mathcal{D}}(\widehat{\mathbf{w}}, \widehat{b}) \approx J^{val}(\widehat{\mathbf{w}}, \widehat{b})$ and both are relatively big, then we are in the presence of high bias. An example of this situation would be the use of a linear function in one variable to approximate a nonlinear relationship (for example a quadratic function). In this case, the model is said to *underfit* the data.

Conversely, if $\hat{\mathcal{R}}_{\mathcal{D}}(\widehat{\mathbf{w}}, \widehat{b}) << \mathcal{R}_{\text{Val}}(\widehat{\mathbf{w}}, \widehat{b})$, then we are in the presence of high variance. An example of this situation would be the use of a high degree polynomial to approximate a relationship given by a relatively small number of data points. In this case, the model is said to *overfit* the data.

So far we have discussed only how the bias and variance change as the flexibility of the models increase, while maintaining the size of the training dataset. A complementary analysis consists of fixing a particular model and varying the size of the training set. In this case, using a larger training set

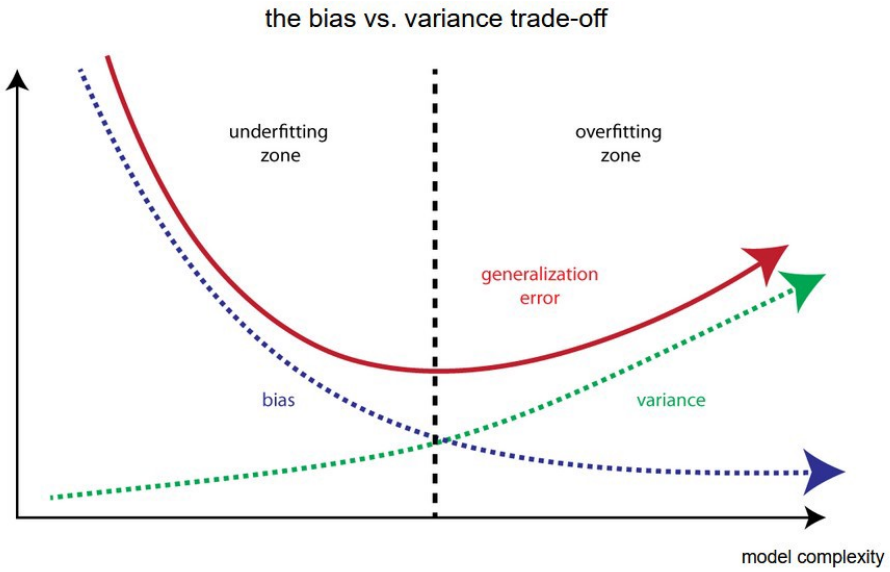

Figure 9: Example of bias (blue), variance (green) and true risk/generalization error (red) as functions of complexity flexibility. We can see that the bias always decreases with flexibility, whereas the variance increases. Because the true risk function depends on both, it eventually increases with flexibility. Source: Figure 9 in Joseph (2017).

typically has the effect of *increasing* the training cost function (as there are now more data points to fit using the same fixed model) and *decreasing* the true risk function (as the optimal model becomes a better approximation of the real-life relationship). What distinguishes a high bias from a high variance situation are the asymptotic values of these costs functions as n becomes large.

In a high bias scenario, both the training and true risk functions converge to the same relatively large value. In this case, adding more samples to the training dataset has little effect. As an example, consider again a real-life quadratic relationship being approximated by a straight line. Conversely, in a high variance scenario, an initial sizeable gap between the training and the true risk functions can typically decreases as the training set gets larger, until both converge to a relatively small value. An example, of this is a high degree polynomial used to approximate a real-life relationship expressed by a small number of data points. As more data points are added, the optimal polynomial eventually converges to a stable relationship.

We conclude this section by mentioning that, recent breakthroughs in *highly*

overparameterized models show that if the number of covariates d is *much larger* than the number of training samples then, the bias-variance trade-off phenomenon from classical statistics (illustrated in Figure 9) may not be correct. Indeed has recently become known that for *sufficiently overparameterized models* the variance, bias, and true risk (generalization error) may start to become better again. Still this "double descent" phenomenon is currently not well understood and is one of the largest open problems in contemporary statistics and statistical learning (illustrated in Figure 2.1.4).

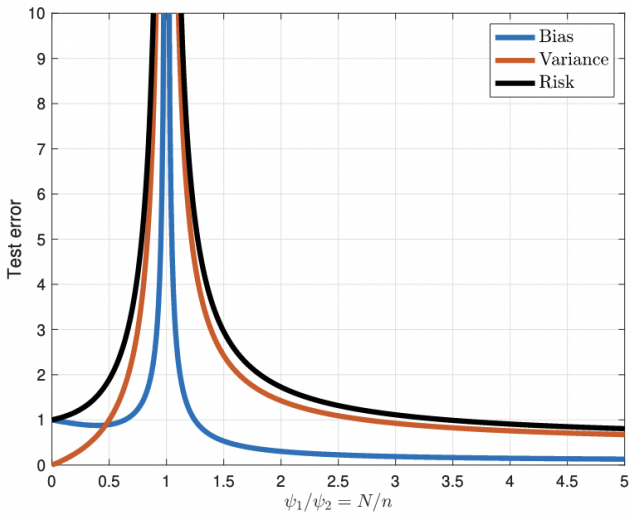

Figure 10: The double-descent phenomenon from random neural networks; taken from [25].

2.2 Hyperparameter Selection: Cross Validation

As we have seen, in addition to the parameters of an individual model, which are found by minimizing the regularized training cost function $\hat{\mathcal{R}}_{\mathcal{D}\lambda}(\widehat{\mathbf{w}}, \widehat{b})$, one can modify the model by changing either the number of features (for example by systematically adding polynomial functions of the existing features) or the regularization parameter λ, or both. A natural idea consists of computing the true risk function for the different models and then picking the model with the lowest cost. However, because the validation dataset was already used to choose the best model, the true risk function for such model is likely to be an underestimate for the true *generalization error*, that is to say, the error arising

by applying the fitted model to entirely new samples.

A better idea consists of designating a subset of the test data as a *cross validation* dataset and use this for model selection. One can then use the cost function on the remaining test data, which has not been use to optimize any of the parameters, to obtain an estimate of the generalization error. As a rule of thumb, given a large data set it is customary to allocate 60% of the data as a training set, 20% as a cross validation set, and the remaining 20% as a test set.

2.2.1 Examples

The following examples are adapted from *An Introduction to Statistical Learning: with Applications in R* by Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani (Springer, 7th edition, 2017), which we refer to as ISLR in the sequel. For brevity, we omit the R output following each command.

Example 2.6. *Simple linear regression - median house value versus percentage of households with low socioeconomic status in the Boston dataset.*

```
> library(MASS)
> install.packages("ISLR")
> library(ISLR)
> View(Boston)
> attach(Boston)
> plot(lstat,medv)         # x = low socioeconomic status, y = median house value
> lm.fit=lm(medv~lstat)
> summary(lm.fit)        # coefficients confirm a decreasing relationship
> abline(lm.fit)           # best linear relationship
> predict(lm.fit,data.frame(lstat=(c(5,10,15))), interval ="confidence")
> plot(predict(lm.fit), residuals(lm.fit))    # residuals should be symmetric
```

Example 2.7. *Multiple linear regression - median house value versus all variables in the Boston dataset.*

```
> install.packages("scatterplot3d")
> library(scatterplot3d)
> s3d<-scatterplot3d(lstat,age,medv)  # x1 = low socioeconomic status, x2 = age
> lm.fit2=lm(medv~lstat+age)  # y = median house value
> summary(lm.fit2)   # dependence on age is less significant
> s3d$plane3d(lm.fit2)
```

```
> plot(predict(lm.fit2), residuals(lm.fit2))   # residuals still not ok
> lm.fit3=lm(medv~.,data=Boston) # y = median house value, x = all variables
> summary(lm.fit3)     # age is not significant
> plot(predict(lm.fit3), residuals(lm.fit3))   # getting better
> lm.fit4=lm(medv~.-age,data=Boston) # excludes the age variable
> plot(predict(lm.fit4), residuals(lm.fit4))  # not much better
> lm.fit5=lm(medv~lstat*age,data=Boston) # includes lstat, age and lstat times age
> plot(predict(lm.fit5), residuals(lm.fit5))   # made it worse
> lm.fit6=lm(medv~lstat+I(lstat^2))  # adds lstat^2 to predictors
> summary(lm.fit6)
> plot(predict(lm.fit6), residuals(lm.fit6))  # much better
```

Example 2.8. *Qualitative Predictors*

```
> View(Carseats)
> contrasts (ShelveLoc ) # examines the dummy variables for ShelveLoc
> lm.fit=lm(Sales~.+Income:Advertising+Price:Age,Carseats)
> summary(lm.fit)
```

The next example is adapted from *Data Science: Theories, Models, Algorithms, And Analytics* by Sanjiv Ranjan Das (S. R. Das, 2016), which we refer to as DAS in the sequel.

Example 2.9. *Linear regression for stock market*

```
> library (quantmod)
> getSymbols(c("^GSPC","AAPL","CSCO","IBM"))
> sp = as.matrix(GSPC[,6])
> aapl = as.matrix(AAPL[ ,6])
> csco = as.matrix(CSCO[,6])
> ibm = as.matrix(IBM[,6])
> stockdata = cbind(sp,aapl,csco,ibm)
> n = length(stockdata[,1])
> returns = log(stockdata[2:n,]/stockdata[1:(n-1),])
> daily_returns = colMeans(returns)
> yearly_returns = 365*daily_returns
> covariance = cov(returns)
> Y = as.matrix(returns[,1])
> X = as.matrix(returns[,2:4])
```

```
> lm.fit = lm(Y~X)
> plot(predict(lm.fit), residuals(lm.fit))
```

The next example is adapted from ISLR.

Example 2.10. *Logistic regression for stock market data*

```
> library(ISLR)
> View(Smarket)  # daily percentage returns of S&P 500
> dim(Smarket)
> cor(Smarket[,-9])    # which correlations appear substantial?
> attach(Smarket)
> plot(Volume)
> glm.fit=glm(Direction~Lag1+Lag2+Lag3+Lag4+Lag5+Volume , data=Smarket ,family=binomial)
> summary(glm.fit)    # glm stands for generalized linear model, including logistic
> glm.probs=predict(glm.fit,type="response")
> contrasts (Direction)
> glm.pred=rep("Down",1250)   # creates a vector of "Down" elements
> glm.pred[glm.probs >.5]="Up"   # changes to "Up" if predicted prob is higher than 0.5
> table(glm.pred,Direction)   # confusion matrix
> mean(glm.pred==Direction)
> train =( Year <2005)    # excludes the year 2005 from training set
> Smarket.2005= Smarket [!train,]   # test data
> dim(Smarket.2005)
> Direction.2005=Direction[!train]
> glm.fit=glm(Direction~Lag1+Lag2+Lag3+Lag4+Lag5+Volume , data=Smarket ,family=binomial,subs
> glm.probs=predict(glm.fit,Smarket.2005,type="response")
> glm.pred=rep("Down",252)
> glm.pred[glm.probs >.5]="Up"
> table(glm.pred,Direction.2005)   # confusion matrix for test data
> mean(glm.pred!=Direction.2005)
> glm.fit=glm(Direction~Lag1+Lag2,data=Smarket ,family=binomial, subset=train)
> glm.probs=predict(glm.fit,Smarket.2005,type="response")
> glm.pred=rep("Down",252)
> glm.pred[glm.probs >.5]="Up"
> table(glm.pred,Direction.2005)
> mean(glm.pred==Direction .2005)   # better confusion matrix once irrelevant variables are
```

3 The Power of Random Features

Clearly, no member of the *linear* (affine) hypothesis class $\{x \mapsto ax + b\}$ is capable of perfectly *approximating* the quadratic function $x \mapsto 3\,x^2$. However, upon inspection, we immediately notice that it we *transform* the input data "x" via the map $x \mapsto x^2$ then, in the transformed coordinates ($u := x^2$) the linear model $u \mapsto 3u$ is capable of exactly pressing the quadratic function $x \mapsto 3\,x^2$. We call the directions in these "new (transformed) coordinates" features.

This leads us the the questions:

> Can be (approximately) implement any function using the right changes of coordinates (feature map)?

Our first answer comes from this Weierstrass approximation theorem.

Theorem 3.1 (Weierstrass Approximation Theorem: Traditional Version)**.** *If $f : \mathbb{R}^d \to \mathbb{R}^D$ is continuous and if $\mathcal{X} \subseteq \mathbb{R}^d$ is compact then, for every "approximation error" $\epsilon > 0$ there is a polynomial function $p_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ satisfying the uniform approximation condition*

$$\max_{x \in \mathcal{X}} \|p_\epsilon(x) - f(x)\|_2 < \epsilon.$$

There is a lot to learn and think about from Weierstrass's approximation result. The first is it introduces us to the notion of universality, which (partially) answers our above question.

Definition 3.2 (Universal Hypothesis Classes)**.** *Let $\mathcal{H}$ be a hypothesis class of functions from $\mathbb{R}^d$ to $\mathbb{R}^D$. The hypothesis class $\mathcal{H}$ is universal if for:*

- *every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$,*
- *every compact subset of inputs $\mathcal{X} \subseteq \mathbb{R}^d$, and*
- *every approximation error $\epsilon > 0$*

there is some hypothesis $\hat{f} \in \mathcal{H}$ satisfying the uniform estimate

$$\sup_{x \in \mathcal{X}} \|f(x) - \hat{f}(x)\|_2 < \epsilon.$$

Since part of the philosophy of machine learning is to assume as little as possible (i.e. have "generic methods") then, we of course want to *construct* and

work with universal hypothesis classes. Not some rigid and inflexible objects like linear hypothesis classes.

Still, we like linear hypothesis classes since they are easy to analysis and to work with (i.e. we want to have our cake and eat it too!!). But how to do that?

A first answer comes by thinking again about Weierstrass' result *(nb, philosophy is important for developing good models)*. For simplicity, let us assume that $d = D = 1$; in this setting, Weierstrass' Approximation Theorem can be restated as follows.

Corollary 3.3 (Weierstrass Approximation Theorem: Feature Map Version)**.** *Set $d = D = 1$. If $f : \mathbb{R}^d \to \mathbb{R}^D$ is continuous and if $\mathcal{X} \subseteq \mathbb{R}^d$ is compact then, there is some positive integer p and some weight $w \in \mathbb{R}^{1+p}$ such that*

$$\max_{x \in \mathcal{X}} \|w^\top \phi(x) - f(x)\|_2 < \epsilon,$$

where the (deterministic + polynomial) feature map $\phi : \mathbb{R} \to \mathbb{R}^{1+p}$ is given by

$$\phi(x) := \begin{pmatrix} 1 \\ x \\ \vdots \\ x^p \end{pmatrix}.$$

Under this new interpretation of Weierstrass' result, we are therefore asking if one can obtain a generic (set of) transformations of the input data $\phi : \mathbb{R}^d \to \mathbb{R}^D$; such that after applying a simple linear regression to the transformed data (called features); the resulting model universal.

We have therefore come full circle to the overparameterized linear regression problem of Section 2.1.2. Let us therefore, restate our little diagram re-illustrate in Figure 11.

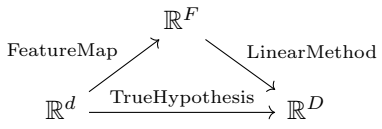

Figure 11: Feature expansion.

Clearly the polynomial feature map of Corollary 3.3 was not the best choice,

since (upon thinking a bit) we would notice that the multi-variate polynomial's degree is an exponential function of time input space's dimension. This is of course problematic, since then the feature space's dimension (F) would be an exponentially large in d; which means that it is not *computationally tractable*. That is, any program which *deterministically* generates polynomial features will not halt in polynomial time!
So we arrive at our new problem:

Q: How can one choose a feature map?

A: What about doing it *randomly*?

The advantage of this randomization procedure is that, as we will see, one can ensure that with almost surely the feature map has "good properties" (i.e. it does not destroy any information in the input space). Moreover, with high probability the feature map can make a simple linear model "generically expressive".

However, before turning our attention to properties of "good feature maps", let us look at some examples of (random) feature maps.

3.1 Examples of (Random) Feature Maps

A few examples and ideas to try out are the following. We being by a few examples and then more towards a sophisticated example which we will often be deploying.

Example 3.4 (Random Fourier Features)**.** *Fix a positive integer k and suppose that $d = 1$. Generate a set of "frequencies" $\alpha_1, \ldots, \alpha_k$ by sampling from the standard normal distribution $N(0, 1)$. Using these set of landmarks we can induce a Fourier-like (augmented) feature map $\phi : \mathbb{R}^d \to \mathbb{R}^{d+k}$ given by*

$$\phi(x) := \big(x, (\sin(\alpha_i \cdot x))_{i=1}^k, (\cos(\alpha_i \cdot x))_{i=1}^k\big).$$

This feature map augments the information in x by adding a set of random "artificial frequencies" to each input x.

Since the projection map sending any $(x, u) \in \mathbb{R}^{d+2k}$ to $x \in \mathbb{R}^d$ is a left-inverse of the feature map ϕ (on it's image!) then ϕ is injective.

Example 3.5 (Random Fréchet Features)**.** *Fix a positive integer k. We generate a finite set of "reference/landmark points" $p_1, \ldots, p_k$ in the input space $\mathbb{R}^d$, by sampling from the standard normal distribution $N(0, 1)$. Using these set of*

landmarks we can induce a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^{d+k}$ *given by*

$$\phi(x) := \big(x, (\|x - p_i\|_2)_{i=1}^k\big).$$

This feature map, encodes the relative geometric information of x*, relative to the landmarks* $p_1, \ldots, p_k$ *as input features.*

Since the projection map sending any $(x, u) \in \mathbb{R}^{d+k}$ *to* $x \in \mathbb{R}^d$ *is a left-inverse of the feature map* ϕ *(on it's image!) then* ϕ *is injective.*

The following example is taken from [18, Corollary 3.20]. It is a particularly beautiful combination of deep random matrix theory results [29] and the straightforward characterization of UAP invariance given in Theorem 3.7 (i).

[(Unnormalized) Random Shallow Neural Network Layer] Fir an integer $F \geq d$. Let A be a (random) $F \times d$ matrix with i.i.d. *columns* $A = (A_1, \ldots, A_F)$ with law

$$A_i \sim \text{Uniformly Distributed on } \{u \in \mathbb{R}^d : \|u\| \leq 1\}.$$

Consider a random bias b, independent of A, with i.i.d. entries with law

$$b_i \sim \text{Uniformly Distributed on } \{u \in \mathbb{R}^d : \|u\| \leq 2\}.$$

Define the (random) feature map $\phi : \mathbb{R}^d \to \mathbb{R}^F$ by

$$\phi(x) := \sigma \bullet (Ax + b),$$

where $\bullet$ denotes the *component-wise* composition of the univariate function σ; i.e.

$$\sigma \bullet (u_i)_{i=1}^F := (\sigma(u_i))_{i=1}^F.$$

3.2 Random Neural Networks are Universal

The first important property of random feature maps is that they genuinely do improve a linear model's *expressive*. A fortiori, they *nearly maximize* it.

Unfortunately, the precise regularity conditions on the target functions are beyond the scope of this course's background *(they require a tad of complex analysis but not much)*. Therefore, we will only state a corollary to the main result of [12], which is strong enough for our purpose.

Theorem 3.6 (Universality of Random Neural Networks)**.** *Let* $D = 1$ *and* P *be a probability measure on* $\mathbb{R}^d$ *with finite variance. For any target function*

$f : \mathbb{R}^d \to \mathbb{R}$, any "approximation error" $\epsilon > 0$ and any "confidence level" $0 < \delta \leq 1$ there is some positive integer N and some weight $w \in \mathbb{R}^N$, such that the random feature map of Example 3.1 satisfies

$$P\left(\mathbb{E}\left[(w^\top \phi(x) - f(x))^2\right]^{1/2} < \epsilon\right) \geq 1 - \delta$$

3.3 Random Neural Networks Preserve Universality

The next property which we seek for a "good feature map" is that it preserves the universality of any hypothesis class (which possesses it). I.e. it does not reduce a hypothesis class's genericness. This invariance property is called UAP-invariance in the machine learning literature (see [19]).

Theorem 3.7 (Characterization of UAP-Invariance [18, Theorem 3.4])**.** *Let $\mathcal{X}$ be a metric space and suppose that the feature map $\phi : \mathcal{X} \to \mathbb{R}^d$ is continuous. Then the following are equivalent:*

(i) ***Injectivity:*** *ϕ is injective,*

(ii) ***UAP-Invariance:*** *For every universal hypothesis class $\mathcal{H}$ of (continuous) functions from $\mathbb{R}^d$ to $\mathbb{R}^D$ the class*

$$\mathcal{H}_\phi := \{f \circ \phi : \ f \in \mathcal{H}\},$$

is a universal hypothesis class from $\mathcal{X}$ to $\mathbb{R}^D$.

We do not need to use only one feature map, but often iteratively combining many may become advantageous. Let's examine the differences between the feature maps designed to have each of these respective properties. NB, we can combine both types of feature maps by composition!

3.3.1 Example of UAP Invariant (Random) Feature Maps

Example 3.8 (Normalized Random Shallow Neural Network Layer)**.** *Fix a positive integer F, at-least as large as d. Let B be a $d \times F$-matrix with entries independently sampled from $\{-1, 1\}$ with equal probabilities; i.e.*

$$P(B_{i,j} = 1) = P(B_{i,j} = -1) = \frac{1}{2}.$$

Surprisingly, as shown in [29], the matrix A has full-rank with P-probability 1. *Therefore, with P-probability* 1, *the following matrix is well-defined*

$$A := \frac{1}{\|B\|_{op}} \cdot B.$$

Moreover, by construction is P-a.s. of operator norm equal to 1. *Recall that $\|B\|_{op}$ is the largest singular value of B computed in Python by*

```
u, s, vh = np.linalg.svd(B, full_matrices=True)
specnormB = np.max(s)
```

Let σ denote the PReLU activation function

$$\sigma(z) := \begin{cases} z & : z \geq 0 \\ \frac{z}{2} & : z < 0. \end{cases} \tag{17}$$

We define a random feature map $\phi : \mathbb{R}^d \to \mathbb{R}^F$ by

$$\phi(x) := \sigma \bullet (Ax).$$

Note that, since B (and therefore A) is P-almost surely injective, the PReLU function σ defined in (17) *is in injective, and since the composition of injective functions is again injective then, the random feature map ϕ is injective with P-probability* 1.

Example 3.9 (Augmented Random Shallow Neural Network Layer)**.** *Consider the setting of Example 3.8. Define the feature map $\tilde{\phi} : \mathbb{R}^d \to \mathbb{R}^{d+F}$ by*

$$\tilde{\phi}(x) := (x, \phi(x)).$$

Q: *By arguing as in Example 3.4 can you show that ϕ is injective?*

3.4 Feature Maps which Generalize Well

So far, we have only asked that our feature map be injective and we hinted that we would like it to be as regular as possible (which in all our examples

amounted) to it having a small Lipschitz constant. However, a more detailed analysis of model expressibility (given in [20]) shows that if we apply a feature map to a universal model, then the resulting model relies on the smallest number of parameters if ϕ and its inverse have very small Lipschitz constants.

This condition is formalized by the distortion of a *bi-Lipschitz* function; defined as follows.

Definition 3.10 (Bi-Lipschitz Functions)**.** *Let $(\mathcal{X}, \text{dist})$ be a metric space. A function $f : \mathcal{X} \to \mathcal{R}^d$ is called bi-Lipschitz if there is a constant $\text{Dist}(f) \geq 1$ (called f's distortion) such that:*

$$\frac{1}{\text{Dist}(f)}\ \text{dist}(x_1, x_2) \leq \|f(x_1) - f(x_2)\|_2 \leq\ \text{Dist}(f)\ \text{dist}(x_1, x_2)$$

for every $x_1, x_2 \in \mathcal{X}$.

Our interest in bi-Lipschitz maps are that they are continuous invertible (whence UAP-invariant) and, that we have *quantitative* (explicit) control on the regularity of their inverses.

Lemma 3.11 (Bi-Lipschitz Maps are Invertible)**.** *If $f : \mathcal{X} \to \mathbb{R}^d$ is bi-Lipschitz with distortion $\text{dist}(f) \geq 1$ then, f is injective and f^{-1} is Lipschitz with Lipschitz constant*

$$\text{Lip}(f^{-1}) = \text{dist}(f).$$

Proof. See this post for a simple proof. □

Example 3.12 (Bounds on Lipschitz Regularity of "Square" Linear Maps)**.** *Let A be a full-rank $d \times d$ matrix. Then, A is invertible, the map $x \mapsto Ax$ is bi-Lipschitz and*

$$\|A^{-1}\|_{op} \leq \text{Lip}(x \mapsto Ax).$$

If, moreover, $\|A - I_d\|_{op} < 1$ then we have the upper-bound (quantitative control) also

$$\text{Lip}([x \mapsto Ax]^{-1}) \leq \frac{1}{1 - \|I_d - T\|_{op}}.$$

Proof. See this post. □

Using these observations let us return to examining the regularity of our main feature map; of Example 3.13.

Example 3.13 (Random Shallow Neural Network Layer (Continued))**.** *Let us evaluate the "regularity" or stability of this feature map. First, we recall that the Lipschitz constant of a composition of functions is itself Lipschitz and its Lipschitz constant is at-most the product of the Lipschitz constant of its constituents. Therefore,*

$$\begin{aligned}\mathrm{Lip}(\phi) \leq & \mathrm{Lip}(\sigma) \cdot \mathrm{Lip}(x \mapsto Ax) \\ = & \mathrm{Lip}(\sigma) \cdot \|A\|_{op} \\ = & \frac{1}{2} \cdot 1 \\ = & \frac{1}{2}.\end{aligned}$$

... TBD in class ...

From the point of view of risk bounds, our interest in bi-Lipschitz feature maps are that they generalize no worse that the original model!

... TBD in class ...

Remark 3.14 (Closing Remarks)**.** *So far, we've used one neural network layer at a time and we have generated our neural network randomly (for the most part). This says little about how well we can do if we optimize all our neural network parameter simultaneously and if we leverage many of these special layers together.... One would expect that doing so yields much better results.*

Indeed this is the case, and it is the subject of the next section. Still the moral of this section is that when initializing a neural network randomly, we are already in good shape.

4 Deep Neural Networks

Neural networks are a surprising (approximate) answer to Hilbert's 13^{th} which (essentially) asks if:

> *Can all (continuous) functions from $\mathbb{R}^d$ to $\mathbb{R}^D$ be (approximately) implemented by "basic functions" + "simple operations"?*

Surprisingly, early insights and developments [24] in an independent area of research; namely artificial intelligence, provided an astounding affirmative answer to this version of Hilbert's question.

4.1 What are Neural Networks?

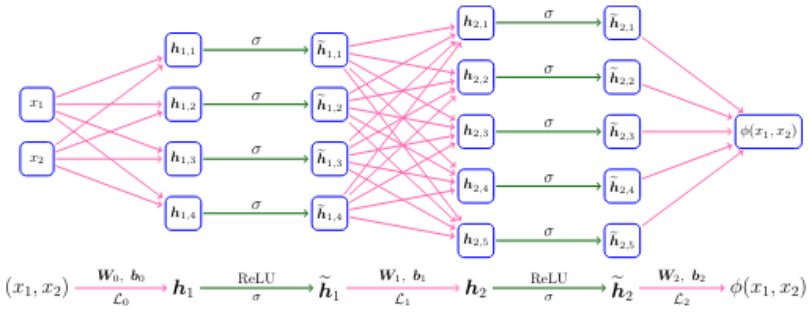

Figure 12: An example of a ReLU network with width 5 and depth 2; from[28].

Neural networks are iterative object, which have the power to *approximately implement* any continuous function (in an appropriate sense), while relying on very *few parameters*. Briefly, a neural network is a very "computerish" function which is defined algorithmically. That is, a (feed-forward) neural network is a function $f : \mathbb{R}^d \to \mathbb{R}^D$ with iterative representation

$$\begin{aligned} f(x) &:= A^{(L)}x^{(L)} + b^{(L)} \\ x^{l+1} &:= \sigma \bullet \left(A^{(l)}x^{(l)} + b^{(l)}\right) \qquad l = 0, \ldots, L-1 \\ x^{(0)} &:= x, \end{aligned} \tag{18}$$

where each $A^{(l)}$ is a $d_{l+1} \times d_l$-dimensional matrix, $b^{(l)}$ is a vector in $\mathbb{R}^{d_{l+1}}$, σ is a continuous function from $\mathbb{R} \to \mathbb{R}$ (which we apply component-wise), $d_0 := d$,

and $d_{L+1} := D$. By component-wise composition we mean

$$\sigma \bullet (x_i)_{i=1}^d := \big(\sigma(x_i)\big)_{i=1}^d.$$

The function σ is called an activation function and typical choices are

- ReLU (Rectified Linear Unit): $\sigma(z) = \max(0, z)$, $\sigma'(z) = \begin{cases} 1 \text{ if } x > 0 \\ 0 \text{ if } x < 0 \end{cases}$,
- Swish: $\sigma(z) = \frac{z}{1+e^{-z}}$ and $\sigma' = \sigma(x) + \frac{1}{1+e^{-x}}(1 - \sigma(x))$,
- Leaky ReLu: $\sigma(z) = \max(\varepsilon z, z)$, for $\varepsilon > 0$, $\sigma'(z) = \begin{cases} 1 \text{ if } x > 0 \\ \varepsilon \text{ if } x < 0 \end{cases}$,
- Sigmoid: $\sigma(z) = \frac{e^z}{1+e^z}$, $\sigma'(z) = g(z)(1 - g(z))$

The hyperparameter L is the *hidden layers* of the neural network $\hat{f}$ and $L+1$ is called $\hat{f}$'s *depth*. The width of the neural network $\hat{f}$ is

$$\max_{0 \le l \le L+1} d_l.$$

That is, $\hat{f}$'s width is the largest domain or co-domain of any of the matrices $A^{(1)}, \ldots, A^{(L+1)}$. The number of *trainable/active parameters* determining the neural network $\hat{f}$ with representation (18) is

$$\text{Param}(\hat{f}) := \sum_{0 \le l \le L+1} \|A^{(l)}\|_0 + \|b^{(l)}\|_0,$$

where $\|\cdot\|_0$ counts the number of non-zero entries in the matrix or vector it acts on. A simple computation then shows that the neural network $\hat{f}$ with representation (18) has at-most

$$\text{Param}(\hat{f}) \le \sum_{0 \le l \le L+1} d_l(d_{l+1} + 1) \le \text{Depth}(\hat{f}) \times \text{Width}(\hat{f})$$

trainable/active parameters.

4.2 Why do we care about neural networks?

Let's begin by studying the noiseless case where every output y is simply the evaluation of the true function $f^\star : \mathcal{X} \to \mathbb{R}^D$ at any in put $x \in \mathcal{X}$. Moreover,

in this scenario, we additionally assume that we have access to *all data*; i.e. we know all pairs $\{(x, f^\star(x)) : x \in \mathcal{X}\}$.

4.2.1 The Approximation Theory of Neural Networks

The first surprising fact about neural networks is that they can approximate any (continuous) function. This was first discovered in [7, 13], where the authors showed the following (qualitative) *universal approximation theorem* for neural networks with one hidden. We begin by reporting the characterization of [26] for neural networks with one hidden layer (called shallow neural networks). Our current discussion operates under the noiseless regime; i.e. "$y = f(x)$".

Theorem 4.1 (Universal Approximation Theorem: Shallow Neural Networks)**.** *Let σ be continuous. Fix a continuous target function $f : \mathbb{R}^d \to \mathbb{R}^D$, every approximation error $\epsilon > 0$, and every radius $r > 0$, there is a neural network $\hat{f} : \mathbb{R}^D \to \mathbb{R}^D$ with activation function σ and with one hidden layer satisfying the uniform estimate*

$$\max_{\|x\| \leq r} \|f(x) - \hat{f}(x)\| < \epsilon.$$

Still, in practice, machine learners have empirically verified that *depth* is more important than *width* when determining a model's expressive power. Recently, in [33] that this is indeed the case for the ReLU activation function and then in 2020 it was shown in [17] that approximation is indeed possible by neural networks which have large depth but whose width does go above a certain (dimension-dependant) level. Those results form the *qualitative* counterpart of Theorem 4.1; this can be informally thought of as a "deep universal approximation thoerem".

Still, those results do not explain the advantage which deep neural networks have over their shallow counterparts in practice. For the popular ReLU activation function, those results where derived only recently by [34] with optimal constants obtained in [28] for the case where $\mathcal{X} = [0, 1]^d$ and $D = 1$ and in [22, Lemma 4] for the general case where $\mathcal{X}$ is an arbitrary compact subset of $\mathbb{R}^d$ and where $D \in \mathbb{N}_+$. For the case of general continuous activation functions, the precise depth and width required to perform a uniform approximation of a given error has been derived in [21].

To fully appreciate the approximation power of feedforward networks, for differentiable functions, we need to recall some basic d

Recap of Basic Analysis

For this, we need to recall the definition of a *Lipschitz function*. Given a constant $L > 0$ and $\alpha \in (0, 1]$, a function $f : \mathbb{R}^d \to \mathbb{R}^D$ is called (L, α)-*Hölder* if the

$$\|f(x_1) - f(x_2)\|_2 \leq L \|x_1 - x_2\|^\alpha,$$

for every $x_1, x_2 \in \mathbb{R}^d$. In the special case where $\alpha = 1$, f is called L-Lipschitz. If, moreover, f is L-Lipschitz (resp. (L, α)-Hölder) for *some* $L > 0$ then we simply say that f is Lipschitz (resp. Hölder).

A subset $\mathcal{X} \subseteq \mathbb{R}^d$ is said to be *compact* if it is closed[1] and bounded[2] The *diameter* of a compact set $\mathcal{X}$ quantifies the maximum possible distance between any two points in $\mathcal{X}$ and it is formalized by the optimization problem

$$\max_{x_1, x_2 \in \mathcal{X}} \|x_1 - x_2\|.$$

Back to the point

We begin by quoting the optimal results for ReLU networks.

Theorem 4.2 (Uniform approximation of Lipschitz maps)**.** *Let $X \subseteq \mathbb{R}^d$ be non-empty and compact and let $f : X \to \mathbb{R}^D$ be Lipschitz. For every "depth parameter" $L \in \mathbb{N}_+$ and "width parameter" $N \in \mathbb{N}_+$ there exists a $\hat{f} \in \mathrm{NN}$ satisfying the uniform estimate*

$$\max_{x \in X} \left\| f(x) - \hat{f}(x) \right\| \lesssim C_d \operatorname{diam}(\mathrm{X}) \operatorname{Lip}(f) \frac{D^{3/2} d^{1/2}}{N^{2/d} L^{2/d} \log_3(N+2)^{1/d}},$$

where $\lesssim$ hides an absolute positive constant independent of X, d, D, and f and where C_d is a constant depending only on $\mathbb{R}^d$. Furthermore, $\hat{f}$ satisfies

1. ***Width**: $\hat{f}$'s width is at-most $d(D+1) + 3^{d+3} \max\{d\lfloor N^{1/d} \rfloor, N+2\}$*
2. ***Depth**: $\hat{f}$'s depth is at-most $D(11L + 2d + 19)$.*

Remark 4.3 ("Low-Dimensional Manifold Hypothesis")**.** *In fact, if X is a $\tilde{d} < d$ dimensional C^0-submanifold of $\mathbb{R}^d$ then, the constant C_d can instead be taken to be $\tilde{d}$.*

[1] The limit of any convergent sequence of elements all in $\mathcal{X}$ must converge in $\mathcal{X}$.
[2] There is some constant $M > 0$ such that $\|x_1 - x_2\| \leq M$ for any $x_1, x_2 \in \mathcal{X}$.

4.2.2 How do Neural Networks Approximate?

Let's visually sketch how neural networks approximate target functions (always working in the noiseless regime). We follow the grown-breaking optimal, and first, arguments of [28] using their pictorial representation. We sketch the proof in the case where $\mathcal{X} = [0,1]^d$ and $D = 1$; noting simply that the general case requires tools from the theory of Lipschitz/uniform extensions of uniformly continuous functions which is beyond the scope of these notes (see [22], [1], and [21] for the introduction of these tools in "theoretical" deep learning). For illustrative reasons, we consider the case where $d = 2$.

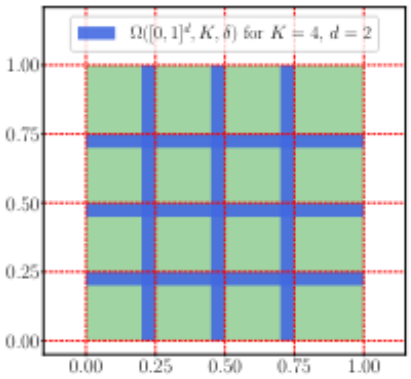

Figure 13: Step 1: Quantize the domain $[0,1]^2$ into little (cubes) with a tiny bit (of space (much less than $\frac{\epsilon}{2}$) between each cube); from [28].

We first subdivide the domain into little cubes, each of length $\frac{\epsilon}{2}$ (figure 13). We keep a bit of space in between each cube.

Next, a representative point in the middle of each little cube is chosen an a function, implemented by a ReLU neural network, implements the "representative values" at the middle of each cube $\{x_1, \ldots, x_K\}$; subdividing our input space $\mathcal{X} = [0,1]^d$ (K is the number of these cubes). In this step, also we linearly interpolate the function while fixing the level of the linear interpolator within each little cube and linearly interpolating between theses constant functions in the little crevices between the little cubes; thus we "discretize" the relevant part of the output space $f(\mathcal{X}) \subset \mathcal{Y} = \mathbb{R}$ (Figure 14). Let's denote the value of each $f(x_n)$ by y_n; so the relevant part of $f(\mathcal{X})$ has *approximately* been reduced to the discrete set of values $\{y_1, \ldots, y_K\}$.

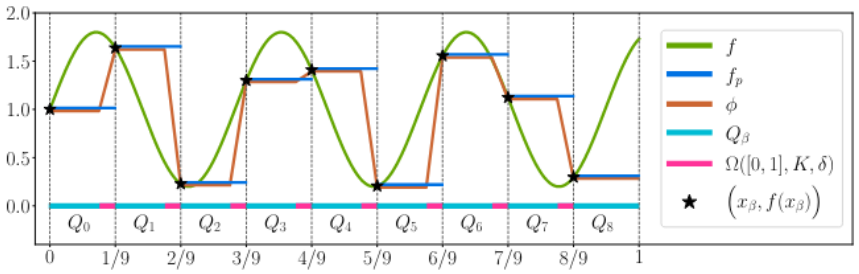

Figure 14: Step 2: Piecewise linear interpolation of target function $f^\star$ where each "level" of the function $f^\star$ is (fixed inside each little sub-cubes) and the linear interpolator can "vary" in the little (spaces between each of cubes.); from [28].

Finally, in the last step, a function implementing the problem implement a function which maps each of the representative points in $\{x_1, \ldots, x_K\}$ to $\{y_1, \ldots, y_K\}$. That is, we seek a neural network which can memorize and implement the assignment

$$\text{Little Cube containing } x_k \mapsto y_k.$$

Combining both neural networks in steps 2 and 3 yields our approximating neural network (Figure 15).

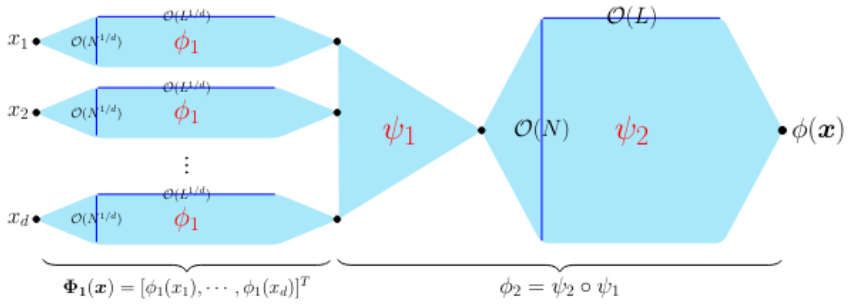

Figure 15: Step 3: Combining the "discretizing" neural network in step 2 with the "piecewise constant interpolating neural network" of step 3; from [28].

4.3 Breaking the Curse of Dimensionality: From Whitney to Your Laptop

Upon examining the *approximation rate* derived in Theorem 4.2 one may still realize that the deep neural network required to achieve ϵ accuracy may still depends on many parameters. Moreover, this dependence is an exponential function of d for a fixed reciprocal approximation error $\frac{1}{\epsilon}$.

A first explanation for this phenomenon passes through a connection with one of the deepest theorems in mathematical analysis. Namely, the *Whitney Extension Theorem* solved after almost half a decade by [9] (see [10] for an interesting related discussion on Whitney's Extension Theorem).

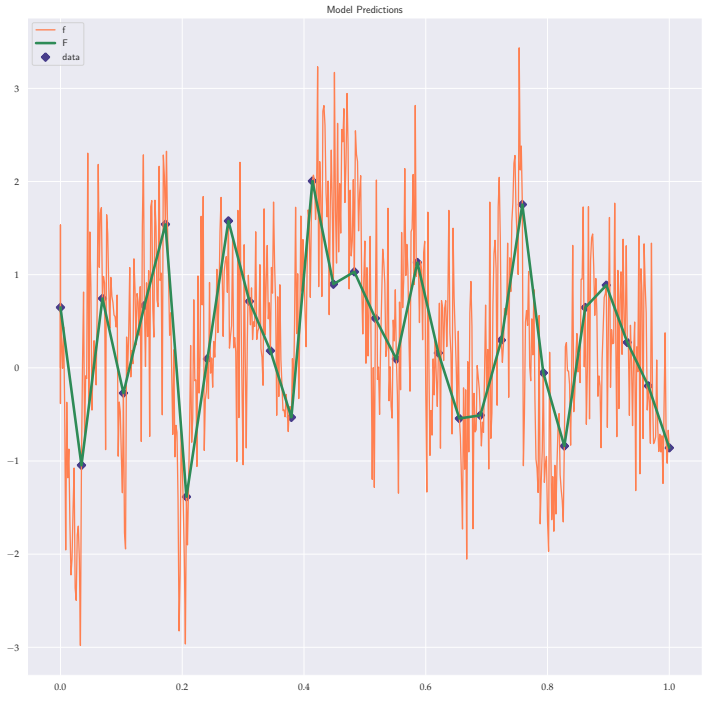

Figure 16: Efficient Approximation by Whitney Extension

Informally, this results states gives conditions on the dataset $\mathcal{D}$ guaranteeing

that $f^\star$ can be replaced by the approximation of a *smooth function* $F : \mathbb{R}^d \to \mathbb{R}^D$ which by chance happens to coincide with $f^\star$ on the dataset $\mathcal{D}$. Now, the regularity of this coincidental function F will completely dictate how efficiently it can be approximated by a deep feedforward neural network. However, the more regular (i.e. the smoother with smaller higher-order partial derivatives) F is then the smaller our approximating neural networks needs to be!

This connection, first established in [21], is concisely summarized by Figure 16 wherein see that the green function F coincides with the target function $f^\star$ on the dataset $\mathcal{D}$ but F is much more regular. Thus, if we instead approximate the more regular function F by a neural network on all of the input space, then we can do so with a neural network which avoids the curse of dimensionality and simultaneously obtain an equally accurate approximation of target function f on dataset $\mathcal{D}$ since both target function f and the green function F coincide thereon.

Theorem 4.4 (Polynomial Approximation Rates On Efficient Datasets)**.** *Fix $n \in \mathbb{N}_+$, let $f : \mathcal{X} \to \mathbb{R}^D$, σ be a non-affine piecewise linear activation function (e.g.* ReLU*), and let the dataset $\mathcal{D} \subseteq \mathbb{R}^d$ (possibly infinite!) be such that there is an nd-times continuously differentiable function $F : \mathbb{R}^d \to \mathbb{R}^D$ satisfying*

$$F(x) = f(x) \qquad \forall x \in \mathcal{X}$$

and such that F's np^{th} partial derivatives are all Lipschitz.

Then, for each $\epsilon > 0$, there is a deep feedforward neural network with activation function σ and a constant $\kappa > 0$ (not depending on ϵ, d, or on D), such that:

$$\max_{x \in \mathcal{D}} \|f^\star(x) - \hat{f}(x)\| \leq \kappa D^{\frac{1}{2}} \epsilon. \tag{19}$$

Moreover, $\hat{f}$ satisfies the following sub-exponential complexity estimates:

(i) ***Width (W):*** *satisfies $D \leq W \leq D(4d + 10)$,*

(ii) ***Depth (D):*** *is $O\left(D + D\epsilon^{\frac{2p}{3(nd+1)} - \frac{p}{nd+1}}\right)$,*

(iii) ***Number of trainable parameters:*** *is $O\left(D(D^2 - 1)\epsilon^{-\frac{2d}{3(nd+1)}}\right)$.*

We close this section by noting that, the dataset $\mathcal{D}$ in Theorem 4.4 wsa not required to be finite. Indeed, if $\mathcal{D}$ was a true *training dataset; i.e. if it were finite* then we can do much better.

4.4 Does depth really Matter?

One of the beauties of having optimal approximation *rates* in therms of a networks hyperparameters, applicable to arbitrary functions of a given regularity (e.g. as in Theorem 4.2) is that we may understand the role/importance of each of these "knobs" when deploying a model in real-life. Looking at the rates in our approximation theorems above, it seems that depth is more expressive than width (i.e. it has a larger impact on the approximation quality) but only marginally (i.e. only up to a logarithmic factor; provided that the network $\hat{f}$ has a minimum depth and width). These observations beg the question:

Does depth give that much of an edge over network width?

Practitioners and experiments indicate that depth does greatly matter; so why does it not show up in our estimates? Well, this is because the above theorems concerns generic classes of functions which we only knew about their regularity and nothing more about their structure. However, if more is known about the target function $f^\star$'s structure then we can begin to see a difference in the effects of depth and width in approximation quality/rates of uniform convergence (on compact sets). A fascinating answer to this question can be found in the wonderful survey paper of [27].

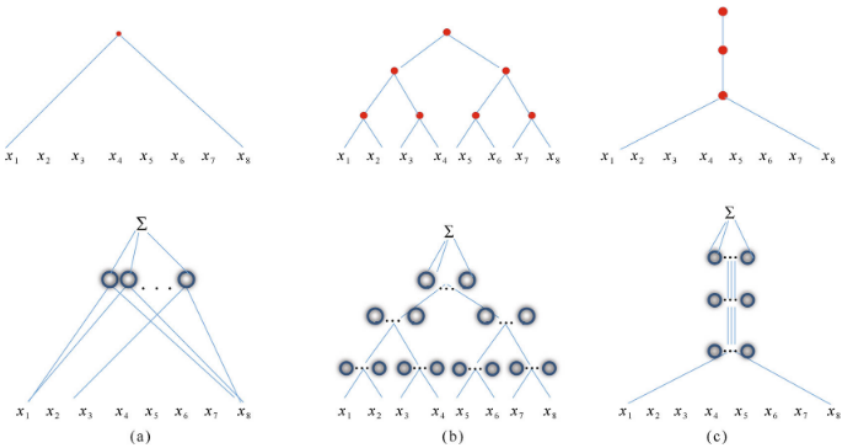

Figure 17: Graphical representation of compositional functions.

Briefly, a function $f^\star : \mathbb{R}^d \to \mathbb{R}^D$ is of compositional type if it can be written as the composition of several independent functions (of possibly lower

regularity) which only depend on few covariates. Since shallow neural networks can approximate each of these functions efficiently, as they are inherently low-dimensional, then we can approximately implement each of them using a shallow neural network. Composing each of these shallow neural networks and forming a *deep neural network* with the same structure as the original compositional functions lets us approximate $f^\star$ to high precision by leveraging our approximation theorems and $f^\star$ "deep structure".

Further details of this type of construction, can be found in [6] and in [27].

4.5 Generalization/Risk Bounds For Neural Networks

In practice, the power of deep neural networks to perform so well on unseen data remains one of the largest open problems in statistical/machine learning theory. Though many approaches have attempted to explain this phenomenon, all answers seem to fall short of what is observed in practice. The following is a simple consequence of a deep and very result at the intersection of statistical learning theory and optimal transport.

We need only recall some simple facts about matrices. For an (Lebesgue) almost-everywhere differentiable activation function $\sigma : \mathbb{R} \to \mathbb{R}$ we define its *Lipschitz constant* by

$$\mathrm{Lip}(\sigma) := \sup_{x_1 \in \mathbb{R}} |\sigma'(x)|$$

where σ' is computed at all points where σ is differentiable.

Example 4.5. *If* $\sigma = \mathrm{ReLU}$ *then* $\mathrm{Lip}(\sigma) = 1$.

For an affine map $x \mapsto Ax + b$, where A is a matrix and b is a vector, the Lipschitz constant is given by the *spectral norm* of the matrix A; which we denote by $\|A\|_{\mathrm{spec}}$. That is, $\|A\|_{\mathrm{spec}}$ is the largest singular value in A's SVD decomposition; in Python this can be computed using the numpy package via

```
u, s, vh = np.linalg.svd(A, full_matrices=True)
specnormA = np.max(s)
```

Assumption 4.6. *The domain* $\mathcal{X}$ *is a compact subset of* $\mathbb{R}^d$ *the* d*-dimensional Euclidean space, and the image* $\mathcal{Y}$ *is a compact subset of* $\mathbb{R}$.

Assumption 4.7. *There exists a constant $L_{\hat{f}} \geq 0$ such that the deep feedforward network $\hat{f}$ with representation* (18) *(trained on the data $\mathcal{D}$) $\mathbb{P}$-almost surely satisfies*

$$\mathrm{Lip}(\sigma)^{L+1} \prod_{i=0}^{L+1} \|A^{(l)}\|_{\mathrm{spec}} \leq L_{\hat{f}}$$

Assumption 4.8. *The loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is L_ℓ-Lipschitz.*

Theorem 4.9 (Generalization error of Lipschitz estimators)**.** *Set $L_\ell, L_{\hat{f}} > 0$ and let $\hat{f}$ have representation* (18) *and which can depend on the sample $\mathcal{D}^N$. Suppose Assumptions 4.6, 4.7, and 4.8 hold.*

For any "confidence level" $0 < \delta \leq 1$ it holds that

$$\mathcal{R}(\hat{f}) - \hat{\mathcal{R}}_{\mathcal{D}}(\hat{f}) \leq \left[\mathrm{cost}_{\mathrm{transport}}(\mathbb{P}) + \mathrm{err}_{\mathrm{transport}}(\mathbb{P})\right] \times \mathrm{diam}(\mathcal{X} \times \mathcal{Y})$$

with probability greater than $1 - \delta$, where

$$\mathrm{cost}_{\mathrm{transport}}(\mathbb{P}) := C_{d+1,1} L_\ell N^{\frac{d}{d+1}-1} \max\left\{1, \mathrm{Lip}(\sigma)^{L+1} \prod_{i=0}^{L+1} \|A^{(l)}\|_{\mathrm{spec}}\right\},$$

$$\mathrm{err}_{\mathrm{transport}}(\mathbb{P}) := \sqrt{\frac{\ln(4/\delta)}{N}} L_\ell \max\{1, L_{\hat{f}}\}.$$

The constant $C_{d+1,1}$ is recorded in Table 1.

Dimension	Constant ($C_{d+1,\alpha}$ or $C_{d+1,s}$)
$d+1 < 21$	$C_{d+1,1} = \frac{d+1^{1/2} 2^{d+1/2-2}}{1-2^{d+1/2-1}}$
$d+1 = 2$	$C_{d+1,1} = \frac{d+1^{1/2}}{2^2}$
$d+1 > 2$	$C_{d+1,1} = 2\left(\frac{\frac{d+1}{2}-1}{2(1-2^{1-d+1/2})}\right)^{2/d+1} \left(1 + \frac{1}{2^1(\frac{d+1}{2}-1)}\right) d+1^{1/2}$

Table 1: Concentration rates and constants for thm:concentration$_m ain$

The power of Theorem 4.9 is that is is an *instant dependant bound*. That is, it is computer for the actual neural network produced from *any* training algorithm on a given *training dataset*. This is completely different from previous bounds which estimated the worst-case out-of-sample performance of *any* neural network model on *any* training dataset. As you can imagine, those classical approaches produced *enormous* estimates which made the generalization bounds truly unusable! This is not the case for the beautiful bound we have just seen.

Remark 4.10 (The compactness of $\mathcal{Y}$ in practice)**.** *Note that $\mathcal{Y} = \{0, \dots, C\}$ for a classification task, and for regression we simply can take $\mathcal{Y} = [-M, M]$ for suitably large $M > 0$.*

Remark 4.11 (Further Improvements on the Bound)**.** *One can greatly improve the bound further, and this is what is done in the paper it is taken from; however, this requires (randomly/adaptively/or smartly) partitioning the input-output space $\mathcal{X} \times \mathcal{Y}$. Time permitting we can discuss such partitioning algorithms but I fear this is beyond the scope of this introductory course.*

4.6 Getting Started IRL

Let's put our discussion on pause, and, before we get starting; make sure all our Python packages are installed.

```
% Install Tensorflow
tensorflow_version 1.x
```

Every time we'll be doing some deep learning, we'll be loading the same "basic" packages. Let's get into the habit:

```
% Import Packages
import numpy as np
import tensorflow as tf
import sys
import keras
(x_train, y_train),
(x_test, y_test) = tf.keras.datasets.mnist.load_data()
print('Python version ', sys.version)
print('Tensorflow version ', tf.__version__)
print('Keras version', keras.__version__)
% Import Plotting Packages
import matplotlib.pyplot as plt
```

A Note: There are Other Types of Neural Networks

The neural networks described so far are called Feed-Forward Neural Networks (FFNN). Other examples include:

- Convolution Neural Networks (CNN): nodes in one layers are only connected to a subset of nodes in the previous layer.
- Recurrent Neural Networks (RNN): each node in a hidden layer can be connected to itself.
- Long-Short Term Memory (LSTM) networks: each node is a complex unit consisting of gates that process the information flow, that is, decide how many previous inputs should be remembered or forgotten at each step. A special case consists of a Gated Recurrent Unit (GRU).
- Autoencoder: the network tries to learn the identity map, that is, the output layer is a copy of the input layer.

5 Classification

Let $\mathcal{X} \subseteq \mathbb{R}^d$ be a non-empty compact subset, C be a positive integer, and define the (ordinal) set $[C] \stackrel{\text{def.}}{=} \{1, \dots, C\}$. Every point in $\mathcal{X}$ is "labelled" via an (a priori) unknown function

$$\mathcal{C} : \mathcal{X} \to [C]. \tag{20}$$

The map $\mathcal{C}$ in (20) is called a *classifier*. Each $c \in [C]$ is called a *label*, and the pre-images of each label c under the map $\mathcal{C}$; i.e.

$$C_i \stackrel{\text{def.}}{=} \mathcal{C}^{-1}[\{c\}] = \big\{x \in \mathcal{X} : \mathcal{C}(x) = c\big\}$$

is the *class* labeled by c. Note that every element of $\mathcal{X}$ is labeled (possibly non-uniquely) since

$$\bigcup_{c=1}^{C} \mathcal{C}^{-1}\big[C_i\big] = \mathcal{C}^{-1}\big[\cup_{c=1}^{C}\ C_i\big] = \mathcal{C}^{-1}\big[\cup_{c=1}^{C}\ \{c\}\big] = \mathcal{C}^{-1}\big[[C]\big] = \mathcal{X};$$

where this derivation uses elementary facts about pre-images of sets.

The goal of the classification problem is to *learn* the map $\mathcal{C}$ by selecting a learner/model from our hypothesis class of *possible classifiers*, given a finite set

of noisy data. The (finite) noisy data is a (non-empty) set $\{(X_n, C_n)\}_{n=1}^N$ of in dependant and identically distributed random variables taking almost-surely values in $\mathcal{X} \times [C]$. NB, in this case our *hypothesis class* $\mathcal{H}$ of possible classifiers are maps from $\mathcal{X}$ to $[C]$.

At this level, the classification problem is too general since the map $\mathcal{C}$ in (20) can be any function. Equivalently, the classes $\{C_c\}_{c=1}^C$ can be *any subsets* of $\mathcal{X}$. Let's gain some insight about what a "reasonable classifier" should look like by studying "reasonable" classes. After which, we can build and study reasonable hypothesis classes of classifiers.

5.1 Structure of Classes and Classifiers

For the moment, we will assume that there is no noise! That is, x_n are points in $\mathcal{X}$ and $C_n = \tilde{C}(x_n)$.

We will consider two classes of subsets of $\mathcal{X}$ which are "well-behaved", open sets and closed sets. For the unfamiliar reader, the ideas are summarized in Figure 18

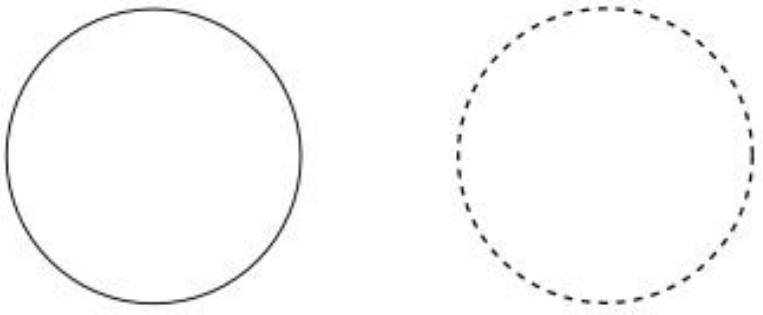

Figure 18: A prototypical **closed set** (left) and a prototypical **open** set (right).

5.1.1 Open Classes

We recall that a subset U of $\mathcal{X}$ is called *open* if for every $x \in U$, there is some radius $r > 0$ such that $\{u \in \mathcal{X} : \|u - x\| < r\} \subseteq U$. If each class $C_1, \ldots, C_C \subseteq \mathcal{X}$ is open then we know exactly how the classifier $\mathcal{C}$ in (20) must "look". Specifically a consequence of Uryson's Lemma, for each $c \in [C]$, [8,

Corollary 1.5.13] implies that there is a *continuous function* $\tilde{C}_c : \mathcal{X} \to [0,1]$ with the property that: for every $x \in \mathcal{X}$ we have

$$x \in C_c \text{ if and only if } 1/2 < \tilde{C}_c(x). \tag{21}$$

Therefore, (21) implies that

$$x \in C_c \text{ if and only if } I_{(1/2,1]} \circ \tilde{C}_c(x) = 1. \tag{22}$$

Therefore, (22) implies that if each class is open then the classifier $\mathcal{C}$ in (1) can be represented as

$$\mathcal{C}(x) = \left(I_{(1/2,1]} \circ \tilde{C}_c(x)\right)_{c=1}^{C}, \tag{23}$$

for some continuous functions $\tilde{C}_1, \ldots, \tilde{C}_C : \mathcal{X} \to [0,1]$.

The representation (23) suggests that, if we have a *universal hypothesis class* $\tilde{\mathcal{H}}$ with inputs in $\mathbb{R}^d$ and outputs $[0,1]$, then we could build a "universal classifier" by thresholding outputs above $1/2$. That is, a viable "universal" class of classifiers can be built by first re-normalizing and then thresholding the outputs of our set of deep feedforward networks. More precisely, our universal hypothesis class of classifiers $\mathcal{H}$ can be the set of functions

$$\hat{C} : \mathcal{X} \to [C]$$

with representation

$$\hat{C} \overset{\text{def.}}{=} (I_{(1/2,1]} \circ \sigma) \bullet \hat{f}, \tag{24}$$

where $\hat{f} : \mathbb{R}^d \to \mathbb{R}^C$ be a deep feedforward neural network, $\bullet$ denotes component-wise composition and σ is the sigmoid function defined by

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

Remark 5.1 (Universal Classification)**.** *The reader versed in topology is referred to [18, Theorem 3.11] for the rigorous statement guaranteeing the universality of the hypothesis class defined by* (24)*, when* $\{0,1\}^C$ *is equipped with products of the Sierpinski topology (i.e. the universal classifying space for open sets).*

5.1.2 Closed Classes

Next, we'll consider the case where the classes $C_1, \ldots, C_C$ are all non-empty *closed* subsets of the *compact* input space $\mathcal{X}$. In this case, there is a well-defined notion of *distance* between any two (non-empty) closed subsets K and $\tilde{K}$ of $\mathcal{X}$. This notion of distance is the *Hausdorff distance* defined as a type of worst-case distance which makes use of the fact that, since K is closed (and non-empty) then for any other $x \in \mathcal{X}$, the distance of x to K

$$d(x, K) \stackrel{\text{def.}}{=} \inf_{u \in K} \|u - x\|$$

is achieved; i.e. $d(x, K) \stackrel{\text{def.}}{=} \min_{u \in K} \|u - x\|$. This means that there is *at-least one point* $u^x \in K$ for which

$$d(x, K) = \|x - u^x\| \text{ and } u^x \in K. \tag{25}$$

For our problem, the significance of (25) is that u^x does not *lie outside* K. **This is not the case if,** K **were only open as in the RHS of Figure** 18.

Lemma 5.2 (Properties of the Distance Function)**.** *If $\mathcal{X}$ is compact and K is a closed and non-empty subset, then the map $x \mapsto d(x, K)$ has the following properties:*

- *It is Lipschitz with Lipschitz constant* 1*,*
- *There is a (finite) constant M such that $d(\cdot, K)$ takes values in $[0, M]$.*

In particular, if the $\sup_{x, \tilde{x} \in \mathcal{X}} \|x - \tilde{x}\| \leq 1$ then $M \leq 1$.

Drawing upon Lemma 5.2, [35] proposed to approximate any such closed K by approximating the distance function $d(\cdot, K)$ to K using ReLU neural networks and a Universal Approximation Theorem (e.g. Theorem 4.4). Rigorous formulations of these ideas were formalized shortly after in [31, 23].

Briefly, given $\epsilon \in (0, M]$, we can find a ReLU neural network $\hat{f} : \mathbb{R}^d \to \mathbb{R}$ satisfying

$$\max_{x \in K} \|\sigma \circ \hat{f}(x) - d(x, K)\| < \epsilon. \tag{26}$$

where as before σ is the sigmoid activation function. Moreover, the estimate (26) implicitly defines a so-called *deep (level) set* as the collection of points in $\mathcal{X}$ for which the "surrogate distance function" $\sigma \circ \hat{f}$ is at-most ϵ i.e.

$$K_{\hat{f}, \epsilon} \stackrel{\text{def.}}{=} \left\{ u \in \mathcal{X} : d(u, \sigma \circ \hat{f}(\mathcal{X})) \leq \epsilon \right\} = \sigma \circ \hat{f}^{-1}[[0, \epsilon]].$$

The connection between the sets $K_{\hat{f},\epsilon}$ and K and the uniform approximation bound in (26) is that (26) implies that *Hausdorff distance* between K and $K_{\hat{f},\epsilon}$ is at-most ϵ. To understand this, we need some definitions. The Hausdorff distance, illustrated in Figure 19, between any two *closed* (and non-empty) subsets K and $\tilde{K}$ of $\mathcal{X}$ is defined as

$$d(K, \tilde{K}) \stackrel{\text{def.}}{=} \max \left\{ \sup_{x \in K} d(x, \tilde{K}), \sup_{y \in \tilde{K}} d(y, K) \right\}.$$

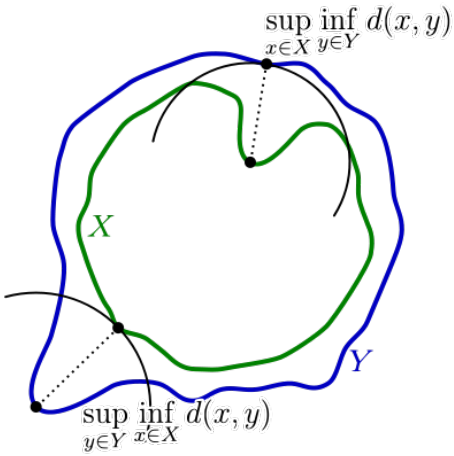

Figure 19: Hausdorff distance between closed sets X and Y.

The estimate (26) implies that

$$d(K, K_{\hat{f},\epsilon}) \leq \epsilon, \tag{27}$$

and this is due to the fact that (since $\mathcal{X}$ is compact and K and $K_{\hat{f},\epsilon}$ are closed) then

$$d(K, K_{\hat{f},\epsilon}) = \max_{x \in \mathcal{X}} \left| d(x, K) - d(x, K_{\hat{f},\epsilon}) \right|.$$

These considerations and Lemma 5.2 suggest that, up to a shift and scaling, if the classes $C_1, \ldots, C_C$ are closed (and non-empty) in $\mathcal{X}$, then the classifiers $\tilde{C}$ in (1) must be representable as

$$\mathcal{C} \stackrel{\text{def.}}{=} ((I_{[1]} \circ d(x/M, C_c))_{c=1}^{C}, \tag{28}$$

and each class C_c is approximable in Hausdorff distance by the set closed sets

$$I_{[1-\epsilon,1]} \circ \sigma \circ \hat{f},$$

where $0 < \epsilon < 1$, σ is the sigmoid function, and $\hat{f}$ is a suitably chosen ReLU feedforward neural network.

Remark 5.3 (Further Reading On Set-Valued Convergence)**.** *The interested reader is referred to [3] for further details on the Hausdorff distance.*

5.2 Borel Classes and The Bayes Classifier

One may induce structure on the classes through the classifier $\mathcal{C}$'s structure. This type of structure arises naturally in the presence of noise!

Let X and Y are random variables taking values in $\mathcal{X}$ and in $[C]$, respectively. Suppose that $\mathbb{P}$ is their joint law. By the Disintegration Theorem[3], the *regular conditional distribution of* Y *given* X, denoted by $\mathbb{P}(Y|X = \cdot)$ is a well-defined (Borel) measurable function taking values in the space $\mathcal{P}([C])$ of probability measures on $[C]$. Before discussing further, let's convince one another that $\mathcal{P}([C])$ is an extremely concrete object; a "visual preview" of our discussion is given in Figure 20.

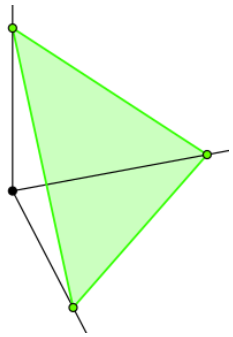

Figure 20: $\mathcal{P}([3])$ is "essentially" just the 3-Simplex.

[3]See [16, Theorem 6.4].

Probability Measures on $[C]$ Though the space $\mathcal{P}([C])$ sounds abstract, in this context, it really isn't since each of its elements $\mathbb{Q}$ are of the form

$$\mathbb{Q} = \sum_{c=1}^{C} w_c \, \delta_c, \tag{29}$$

where $w \in [0,1]^C$ is such that $\sum_{c=1}^{C} w_c = 1$ and the "point-mass/Dirac measure" δ_c is defined on any subset B of $[C]$ by

$$\delta(B) \stackrel{\text{def.}}{=} \begin{cases} 1 & \text{if } c \in B \\ 0 & \text{else} \end{cases}.$$

Therefore, (29) implies that, as a set, $\mathcal{P}([C])$ can be identified with the *standard C-Simplex* defined by

$$\Delta_C \stackrel{\text{def.}}{=} \Big\{ w \in [0,1]^C : \sum_{c=1}^{C} w_c = 1 \Big\}.$$

Next, we equip $\mathcal{P}_1([C])$ with the *total variation distance* d_{TV} defined for any $\mathbb{Q}_1$ and $\mathbb{Q}_2$ in $\mathcal{P}([C])$ as the largest difference which the two probability measures place on any subset B of $[C]$

$$d_{TV}(\mathbb{Q}_1, \mathbb{Q}_2) = \max_{B} \left| \mathbb{Q}_1(B) - \mathbb{Q}_2(B) \right|.$$

Now the identification of $\mathcal{P}([C])$ with Δ_C which sends any probability measure $\mathbb{Q}$ in $\mathcal{P}([C])$ with representation (29) to its "vector of weights" $w \in \Delta_C$; i.e.

$$\begin{aligned} \mathcal{P}([C]) &\to \Delta_C \\ \sum_{c=1}^{C} w_c \, \delta_c &\mapsto w, \end{aligned}$$

is not only continuous but it actually preserves the total-variation distance up to some constant re scaling. That is, there are constants $0 < \kappa_1 < \kappa_2$ such that: for every pair of probability measures $\mathbb{Q}_1 \stackrel{\text{def.}}{=} \sum_{c=1}^{C} w_c^{(1)} \, \delta_c$ and $\mathbb{Q}_2 \stackrel{\text{def.}}{=} \sum_{c=1}^{C} w_c^{(2)} \, \delta_c$ in $\mathcal{P}([C])$ one has

$$\kappa_1 \left\| w^{(1)} - w^{(2)} \right\|_2 \leq d_{TV}(\mathbb{Q}_1, \mathbb{Q}_2) \leq \kappa_2 \left\| w^{(1)} - w^{(2)} \right\|_2.$$

The Bayes Classifier Again, the Disintegration Theorem[4], the *regular conditional distribution of Y given X*, together with our identification of $\mathcal{P}([C])$ with the simplex Δ_C implies that

$$\begin{aligned} \mathbb{P}(Y|X=\cdot) : \mathcal{X} &\to [C] \\ x &\mapsto \big(\mathbb{P}(Y=c|X=x)\big)_{c=1}^{C}, \end{aligned} \tag{30}$$

defines a *Borel-measurable* function.

Note that, in this case every realization of Y can only occupy one state c in $[C]$. Therefore, one typically *needs* to make a choice about which state Y is in, given $X = x$. We would like to take the most likely state (which is the same as the risk-minimizing state), however, on can immagine that Y is equally likely to take two states.

Example 5.4. *Let $(M_t)_{t\geq 0}$ be a real-valued martingale on some filtered probability space $(\Omega, \mathcal{F}, (\mathcal{F}_t)_{t\geq 0}, \mathbb{P})$, $X \overset{\text{def.}}{=} M_0$, and $Y \overset{\text{def.}}{=} M_1$. Let $C = 2$, and define the classes*

$$C_1 \overset{\text{def.}}{=} \{Y > 0\} \text{ and } C_2 \overset{\text{def.}}{=} \{Y < 0\}$$

and assume that $\mathbb{P}(M_t = x) = 0$ for any $x \in \mathbb{R}$ (e.g. this is the case for Brownian motion). Then, by the martingale property, we have that

$$\mathbb{P}(Y = 1|X = x) = \mathbb{P}(Y = 2|X = x).$$

In this case, there is not unique class "c" maximizing

$$\mathbb{P}(Y = c|X = x) = \max_{\tilde{c}=1,2} \mathbb{P}(Y = \tilde{c}|X = x). \tag{31}$$

To avoid problems like in (5.4) we will always make a *measurable (selection)* choice of argmaximizer for problems such as (31).

Remark 5.5 (Notation: Argmax Selection/Choice)**.** *For simplicity, we "abuse notation" and write* **argmax** *for such a choice.*

A Bayes classifier is defines precisely as a *choice* of maximizer as in (31), or equivalently as a risk-minimizer. Formally,

$$\begin{aligned} \mathcal{C}^{\text{Bayes}} : \mathcal{X} &\to [C] \\ x &\mapsto \operatorname{argmax}_{c\in[C]} \mathbb{P}(Y = c|X = x). \end{aligned} \tag{32}$$

[4]See [16, Theorem 6.4].

In particular, a Bayes classifier is a Borel-measurable function and therefore, the classes which it defines

$$C_c \stackrel{\text{def.}}{=} \mathcal{C}^{\text{Bayes}^{-1}}\big[\{c\}\big]$$

are *by definition* Borel subsets of $\mathcal{X}$.

Approximating A Bayes Classifier Clearly, in this context, one cannot *directly observe* **A** Bayes classifier $\mathcal{C}^{\text{Bayes}}$ since this requires intimate knowledge of Y and X. Therefore, the original goal of the classification problem was to infer or learn $\mathcal{C}^{\text{Bayes}}$ using various hypothesis classes of classifiers.

However, we do not have any universal approximation theorems for Borel functions, since those objects can be *too poorly behaved*. Worry-not however, we do have access to Lusin's theorem.

Theorem 5.6 (Lusin's Theorem (In Our Context))**.** *Let $\boldsymbol{P}$ be any prior (Borel) probability measure on $\mathcal{X}$. If $f : \mathcal{X} \to [C]$ is a Borel-measurable function then, for every "confidence level" $0 < \delta \leq 1$, there is a compact subset $\mathcal{X}_\delta$ of $\mathcal{X}$ such that:*

1. *The restriction of f to $\mathcal{X}_\delta$, denoted by $f|_{\mathcal{X}_\delta}$, is continuous,*
2. *Inputs in $\mathcal{X}_\delta$ are "common"; i.e.*

$$\boldsymbol{P}\big(\mathcal{X}_\delta\big) \geq 1 - \delta.$$

In this sense, Lusin's theorem states that, on an arbitrarily high probability region on $\mathcal{X}$, where we *get to choose what high-probability means*, the map f is continuous. Since the universal approximation theorem[5] (Theorem 4.1) allowed us to uniformly approximate continuous functions, then we can conclude the following.

Corollary 5.7 (Universal Probability Approximately Correct (PAC) Deep Classification)**.** *Let X and Y be random variables taking values in $\mathcal{X}$ and in $[C]$ respectively, let C be a positive integer, and let $\boldsymbol{P}$ be a prior probability measure on $\mathcal{X}$. Let $\mathcal{C}^{Bayes}$ be a Bayes classifier for this problem. Then, for any confidence level $0 < \delta \leq 1$ and any "approximation error" $\epsilon > 0$, there is a deep*

[5] Technically, one should use [18, Theorem 3.10] but this is beyond the scope of this course.

ReLU feedforward network satisfying

$$\boldsymbol{P}\big(\|\mathcal{C}^{Bayes}(x) - \sigma \circ \hat{f}(x)\| < \epsilon\big) \geq 1 - \delta,$$

where σ is the sigmoid activation function.

5.2.1 Full-Circle: From Bayes Classifiers to High-Probability Classifiers of Open and Closed Sets

Next, Lusin's Theorem implied that $\mathcal{C}^{\text{Bayes}}|_{\mathcal{X}_\delta}$ was continuous as a function into $[C]$. Since every point $c \in [C]$ is closed (and also open) then the continuity of $\mathcal{C}^{\text{Bayes}}|_{\mathcal{X}_\delta}$ implies that "restricted" the classes defined by

$$\tilde{C}_c \stackrel{\text{def.}}{=} \mathcal{C}^{\text{Bayes}^{-1}}[\{c\}] \cap \mathcal{X}_\delta,$$

are both open and closed. Therefore, our intuitions about classification of open and closed sets, from the previous section, apply here! *Feel free to interpret :)*

We wrap-up this section, by considering a classical logistic regression problem; which is an analogue of the linear regressor we studied in Section 2.

5.3 Logistic regression

Consider now a *binary classification* problem, namely deciding whether an output y belongs to one of two classes, traditionally denoted by 0 and 1, based on inputs $x = (x_1, \ldots, x_p)$. The *logistic regression* method consists of assuming that the probability that $y = 1$ (namely, that y belongs to the class associated with the label 1) is given by

$$f_{\mathbf{w},b}(\mathbf{x}) = \frac{e^{\mathbf{w}\mathbf{x}+b}}{1 + e^{\mathbf{w}\mathbf{x}+b}} = g(\mathbf{w}\mathbf{x} + b) \tag{33}$$

where, as before, $\mathbf{w}$ is a $1 \times p$ row vector of *weights* and b is a bias, and $g : \mathbf{R} \to [0, 1]$ is the *sigmoid function*

$$g(z) = \frac{e^z}{1 + e^z}. \tag{34}$$

Similar to linear regression, given data of the form $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)$, where each $y_j \in \{0, 1\}$, we seek to *estimate* the function f by finding the pa-

rameters $\mathbf{w}$ and b that minimize the cost function

$$J(\mathbf{w}, b) = -\frac{1}{n}\sum_{j=1}^{n}\big(y_j \log f_{\mathbf{w},b}(\mathbf{x}_j) + (1-y_j)\log(1 - f_{\mathbf{w},b}(\mathbf{x}_j))\big) \tag{35}$$

It is straightforward to see that this is equivalent to maximizing the *likelihood function*:

$$\mathcal{L}(\mathbf{w}, b) = \prod_{j:y_j=1} f_{\mathbf{w},b}(\mathbf{x}_j) \prod_{\tilde{j}:y_{\tilde{j}}=0} (1 - f_{\mathbf{w},b}(\mathbf{x}_{\tilde{j}})). \tag{36}$$

In other words, we want to find parameters $\mathbf{w}$ and b such that the function $f_{\mathbf{w},b}(\mathbf{x}_j)$ is a number close to one whenever $y_j = 1$ and close to zero whenever $y_j = 0$.

Using the fact that $\frac{dg}{dz}(z) = g(z)(1 - g(z))$, we find that

$$\frac{\partial}{\partial w_i} J(w_1, w_2, \ldots, w_p, b) = \frac{1}{n}\sum_{j=1}^{n}(f_{\mathbf{w},b}(\mathbf{x}_j) - y_j)x_{ij} \tag{37}$$

$$\frac{\partial}{\partial b} J(w_1, w_2, \ldots, w_p, b) = \frac{1}{n}\sum_{j=1}^{n}(f_{\mathbf{w},b}(\mathbf{x}_j) - y_j), \tag{38}$$

which is identical to what we found in the linear regression case, but with the function $f_{\mathbf{w},b}$ given by (33) instead of (5). Denoting by $g : \mathbf{R}^n \to [0,1]^n$ the map that implements the sigmoid function on each component of an n-dimensional vector, we can write the derivatives above in vector form as

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{n}[g(\mathbf{w}\mathbf{X} + b\mathbf{1}_n) - \mathbf{Y})]\mathbf{X}' \tag{39}$$

$$\frac{\partial}{\partial b} J = \frac{1}{n}[g(\mathbf{w}\mathbf{X} + b\mathbf{1}_n) - \mathbf{Y})]\mathbf{1}_n' \tag{40}$$

so that the iterations of gradient descent for logistic regression can be written as

$$[\mathbf{w}^{(k+1)}, b^{(k+1)}] = [\mathbf{w}^{(k)}, b^{(k)}] - \frac{\alpha}{n}\big[g(\mathbf{w}\mathbf{X} + b\mathbf{1}_n) - \mathbf{Y}\big]\begin{bmatrix}\mathbf{X}\\ \mathbf{1}_n\end{bmatrix}' \tag{41}$$

As before, the iterations above are repeated until a given tolerance is reached (for example, a sufficiently small improvement in the cost function), at which point the algorithm is deemed to have converged to the minimizers $\widehat{\mathbf{w}}$ and $\widehat{b}$. Having found the minimizers $\widehat{\mathbf{w}}$ and $\widehat{b}$, we can predict the output $\widehat{y}$ for any input

$\mathbf{x}$ according to

$$\widehat{y} = \begin{cases} 0, \text{ if } f_{\widehat{\mathbf{w}},\widehat{b}}(\mathbf{x}) < p_0 \\ 1, \text{ if } f_{\widehat{\mathbf{w}},\widehat{b}}(\mathbf{x}) \geq p_0, \end{cases} \tag{42}$$

where p_0 is a threshold value.

5.3.1 Decision boundaries

Setting the value $p_0 = 0.5$ in (42) is equivalent to predicting that $y = 1$ if, and only, if

$$\widehat{\mathbf{w}}\mathbf{x} + \widehat{b} \geq 0, \tag{43}$$

that is to say, a linear decision boundary. For example, suppose that we have two features (that is, $p = 2$) and find that the optimal parameters for logistic regression are $\widehat{w}_1 = 1, \widehat{w}_2 = 1, \widehat{b} = -2$. In this case, we are going to predict that $y = 1$ whenever the feature vector $\mathbf{x} = [x_1, x_2]'$ satisfies

$$x_1 + x_2 - 2 \geq 0,$$

that is to say, whenever the point (x_1, x_2) lies to the right of the line $x_2 = 2 - x_1$. Similarly to linear regression, we can introduce more complicated relationships between the features by considering higher order polynomials of existing features. For example, we can extend the previous example by considering the additional features $x_3 = x_1^2$ and $x_4 = x_2^2$. Suppose that the new optimal parameters in this case are now $\widehat{b} = -4, \widehat{w}_1 = \widehat{w}_2 = 0, \widehat{w}_3 = \widehat{w}_4 = 1$, so that (43) becomes

$$x_1^2 + x_2^2 - 4 \geq 0.$$

This corresponds to a non-linear decision boundary, whereby we predict that $y = 1$ whenever the point (x_1, x_2) lies outside the circle $x_1^2 + x_2^2 = 4$.

5.3.2 Regularization

The possibility of creating complicated decision boundaries for logistic regression leads to the same issues of overfitting that we encountered with linear regression. Namely, having more flexibility in the features (for example, by adding nonlinear functions of existing features) reduces the cost function over the training set but can increase the cost function in more general data sets (say either a cross validation or a test set). The idea of using regularization to constraint the size fo the parameters can also be applied to prevent overfitting in logistic regression.

Specifically, we consider the modified cost function

$$J_\lambda(\mathbf{w},b) = -\frac{1}{n}\sum_{j=1}^{n}\big(y_j \log f_{\mathbf{w},b}(\mathbf{x}_j)+(1-y_j)\log(1-f_{\mathbf{w},b}(\mathbf{x}_j))\big)+\frac{\lambda}{2n}\sum_{i=1}^{p} w_i^2, \quad (44)$$

leading to a modified gradient descent of the form

$$[\mathbf{w}^{(k+1)}, b^{(k+1)}] = \left[\mathbf{w}^{(k)}\left(1-\alpha\frac{\lambda}{n}\right), b^{(k)}\right] - \frac{\alpha}{n}\big[g(\mathbf{w}\mathbf{X}+b\mathbf{1}_n)-\mathbf{Y}\big]\begin{bmatrix}\mathbf{X}\\ \mathbf{1}_n\end{bmatrix}' \quad (45)$$

5.3.3 Logistic regression as a neural network - binary classification

We can interpret logistic regression for binary classification as a very simple neural network consisting of one input layer with p nodes and a single node in the output layer. Given the $p \times 1$ input vector $\mathbf{x}_j$ for the j-th training point, the $1 \times p$ vector of weights $\mathbf{w}$ and the bias b, the first step in the neural network consists of computing the linear combination

$$z_j = \mathbf{w}\mathbf{x}_j + b. \quad (46)$$

Next we apply the nonlinear *activation function* $g : \mathbf{R} \to [0,1]$ and compute the value of the single node in the output layer as

$$a_j = g(z_j) \quad (47)$$

In the language of neural networks, this is called the *forward propagation* of the input $\mathbf{x}_j$. We can then compute the cost associated with this training point according to (35), namely

$$J_j(\mathbf{w},b) = -\big(y_j \log a_j + (1-y_j)\log(1-a_j)\big). \quad (48)$$

In order to implement gradient descent, we need to compute the effect that a change in the weights $\mathbf{w}$ and the bias b has in this cost. This is achieved by what is called *backward propagation*, and essentially consists of repeated applications of the chain rule for derivatives. Start with

$$\frac{\partial J_j}{\partial a_j} = -\frac{y_j}{a_j} + \frac{1-y_j}{1-a_j}. \quad (49)$$

Next compute

$$\frac{\partial J_j}{\partial z_j} = \frac{\partial J_j}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \left(-\frac{y_j}{a_j} + \frac{1-y_j}{1-a_j}\right) a_j(1-a_j) = a_j - y_j \tag{50}$$

Finally, compute

$$\frac{\partial J_j}{\partial w_i} = \frac{\partial J_j}{\partial z_j}\frac{\partial z_j}{\partial w_i} = \frac{\partial J_j}{\partial z_j} x_{ij} = (a_j - y_j)x_{ij} \tag{51}$$

$$\frac{\partial J_j}{\partial b} = \frac{\partial J_j}{\partial z_j}\frac{\partial z_j}{\partial b} = \frac{\partial J_j}{\partial z_j} 1 = (a_j - y_j)1 \tag{52}$$

which are the analogues of (37) and (38) for a single data point. Handling all the data points simultaneously is straightforward using the matrix notation introduced earlier. Namely, we first arrange the inputs as the columns of the $p \times n$ matrix $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x}_n]$ and the outputs as the $1 \times n$ row vector $\mathbf{Y} = [y_1, \ldots, y_n]$. Forward propagation then consists of computing the $1 \times n$ row vector of linear combinations

$$\mathbf{Z} = \mathbf{w}\mathbf{X} + b\mathbf{1}_n, \tag{53}$$

where $\mathbf{1}_n = [1, \ldots, 1] \in \mathbf{R}^n$, followed by computing the $1 \times n$ row vector of values for the single node in the output layer as

$$\mathbf{A} = g(\mathbf{Z}), \tag{54}$$

where $g : \mathbf{R}^n \to [0,1]^n$ applies the activation function to each component of the vector $\mathbf{Z}$. For backward propagation, we start with a cost function for all data points given by

$$J(\mathbf{w}, b) = \frac{1}{n}\sum_{j=1}^{n} J_j(\mathbf{w}, b) = -\frac{1}{n}\sum_{j=1}^{n} \big(y_j \log a_j + (1-y_j)\log(1-a_j)\big) \tag{55}$$

and compute the corresponding derivatives, namely

$$\frac{\partial J}{\partial \mathbf{Z}} := \left[\frac{\partial J}{\partial z_1}, \ldots, \frac{\partial J}{\partial z_n}\right] = \frac{1}{n}(\mathbf{A} - \mathbf{Y}) \tag{56}$$

$$\frac{\partial J}{\partial \mathbf{w}} := \left[\frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_p}\right] = \frac{\partial J}{\partial \mathbf{Z}}\mathbf{X}' \tag{57}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial \mathbf{Z}}\mathbf{1}'_n \tag{58}$$

An iteration of gradient descent can then be written as

$$[\mathbf{w}^{(k+1)}, b^{(k+1)}] = [\mathbf{w}^{(k)}, b^{(k)}] - \alpha \left[\frac{\partial J}{\partial \mathbf{w}}, \frac{\partial J}{\partial b}\right] = [\mathbf{w}^{(k)}, b^{(k)}] - \frac{\alpha}{n}(\mathbf{A} - \mathbf{Y}) \begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix}', \tag{59}$$

which coincides with (41).

5.3.4 Logistic regression - multiple classes

Consider now the problem of classifying an output y into one of q classes. This can be achieved by a generalization of the logistic regression known as *one-versus-all* algorithm, whereby we estimate q different functions $f^1_{\mathbf{w},b}, f^2_{\mathbf{w},b}, \ldots, f^q_{\mathbf{w},b}$ of the form (33) using data of the form $(\mathbf{x}_1, \mathbf{y}_1), \cdots, (\mathbf{x}_n, \mathbf{y}_n)$ where, if the j-th output is in class k, we have that $y_{kj} = 1$ and $y_{ij} = 0$ for all $i \neq k$. For example, in a problem with three classes, if the j-th output belongs to second class we would represent it as

$$\mathbf{y}_j = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}. \tag{60}$$

In other words, we perform q separate logistic regressions on the same inputs, each using a separate row of the output vector. Having found the optimal parameters for each function, we predict that the output associated with an input $\mathbf{x}$ belongs to the class k that maximizes

$$\max_k f^k_{\widehat{\mathbf{w}},\widehat{b}}(\mathbf{x}). \tag{61}$$

The algorithm we just described can be seen as a neural network consisting of one input layer with p nodes as before and one output layer with q nodes, each performing a separate logistic regression on the inputs. As before, we arrange the inputs as $p \times 1$ column vectors $\mathbf{x}_j$, for $j = 1, \ldots, n$, and the weights and biases for each logistic regression as $1 \times p$ row vectors $\mathbf{w}_k$ and scalars b_k, for $k = 1, \ldots q$. In addition, the outputs are now $q \times 1$ column vectors $\mathbf{y}_j$ where, if the j-th output is in class k, we have that $y_{kj} = 1$ and $y_{ij} = 0$ for all $i \neq k$.

In this case, forward propagation for a single input $\mathbf{x}_j$ consists of

$$\mathbf{z}_j = \mathbf{W}\mathbf{x}_j + \mathbf{b} \tag{62}$$

$$\mathbf{a}_j = g(\mathbf{z}_j) \tag{63}$$

where

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_q \end{bmatrix} \qquad \text{and} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_q \end{bmatrix} \tag{64}$$

are the $q \times p$ matrix and the $q \times 1$ column vector whose rows are the weights and the bias for each separate logistic regression. Notice that both $\mathbf{z}_j$ and $\mathbf{a}_j$ are $q \times 1$ column vectors. The cost function associated with this single training sample is now

$$J_j(\mathbf{W}, \mathbf{b}) = -\sum_{k=1}^{q} \big(y_{kj} \log a_{kj} + (1 - y_{kj}) \log(1 - a_{kj}) \big) \tag{65}$$

In other words, this is the sum of the cost functions for each separate logistic regression. It then follows that

$$\frac{\partial J_j}{\partial a_{kj}} = -\frac{y_{kj}}{a_{kj}} + \frac{1 - y_{kj}}{1 - a_{kj}} \tag{66}$$

and therefore

$$\frac{\partial J_j}{\partial z_{kj}} = \frac{\partial J_j}{\partial a_{kj}} \frac{\partial a_{kj}}{\partial z_{kj}} = \left(-\frac{y_{kj}}{a_{kj}} + \frac{1 - y_{kj}}{1 - a_{kj}} \right) a_{kj}(1 - a_{kj}) = a_{kj} - y_{kj} \tag{67}$$

$$\frac{\partial J_j}{\partial w_{ki}} = \frac{\partial J_j}{\partial z_{kj}} \frac{\partial z_{kj}}{\partial w_{ki}} = (a_{kj} - y_{kj}) x_{ij} \tag{68}$$

$$\frac{\partial J_j}{\partial b_k} = \frac{\partial J_j}{\partial z_{kj}} \frac{\partial z_{kj}}{\partial b_k} = (a_{kj} - y_{kj}), \tag{69}$$

where we used the fact that

$$z_{kj} = (\mathbf{W}\mathbf{x}_j + \mathbf{b})_k = w_{k1} x_{1j} + w_{k2} x_{2j} + \cdots w_{kp} x_{pj} + b_k. \tag{70}$$

We can then rewrite these equations in vector notation as

$$\frac{\partial J_j}{\partial \mathbf{z}_j} := \begin{bmatrix} \frac{\partial J_j}{\partial z_{1j}} \\ \vdots \\ \frac{\partial J_j}{\partial z_{qj}} \end{bmatrix} = \mathbf{a}_j - \mathbf{y}_j \tag{71}$$

$$\frac{\partial J_j}{\partial \mathbf{W}} = \begin{bmatrix} \frac{\partial J_j}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J_j}{\partial \mathbf{w}_q} \end{bmatrix} = \frac{\partial J_j}{\partial \mathbf{z}_j} \mathbf{x}_j' \tag{72}$$

$$\frac{\partial J_j}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial J_j}{\partial b_1} \\ \vdots \\ \frac{\partial J_j}{\partial b_q} \end{bmatrix} = \frac{\partial J_j}{\partial \mathbf{z}_j} \tag{73}$$

To handle all data points simultaneously, we first arrange the inputs as the columns of the $p \times n$ matrix $\mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x}_n]$ and the outputs as the columns of the $q \times n$ matrix $\mathbf{Y} = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$. Forward propagation then consists of computing the $q \times n$ matrices

$$\mathbf{Z} = \mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}_n \tag{74}$$

$$\mathbf{A} = g(\mathbf{Z}), \tag{75}$$

where $g : \mathbf{R}^{q \times n} \to [0,1]^{q \times n}$ applies the activation function to each component of the matrix $\mathbf{Z}$. For backward propagation, we start with a cost function for all data points given by

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{j=1}^{n} J_j(\mathbf{w}, b) = -\frac{1}{n} \sum_{j=1}^{n} \sum_{k=1}^{q} \left(y_{kj} \log a_{kj} + (1 - y_{kj}) \log(1 - a_{kj}) \right) \tag{76}$$

and compute the corresponding derivatives, namely

$$\frac{\partial J}{\partial \mathbf{Z}} := \left[\frac{\partial J}{\partial \mathbf{z}_1}, \ldots, \frac{\partial J}{\partial \mathbf{z}_n}\right] = \frac{1}{n}(\mathbf{A} - \mathbf{Y}) \tag{77}$$

$$\frac{\partial J}{\partial \mathbf{W}} := \begin{bmatrix} \frac{\partial J}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J}{\partial \mathbf{w}_q} \end{bmatrix} = \frac{\partial J}{\partial \mathbf{Z}}\mathbf{X}' \tag{78}$$

$$\frac{\partial J}{\partial \mathbf{b}} = \begin{bmatrix} \frac{\partial J}{\partial b_1} \\ \vdots \\ \frac{\partial J}{\partial b_q} \end{bmatrix} \frac{\partial J}{\partial \mathbf{Z}}\mathbf{1}_n', \tag{79}$$

from which we can compute an iteration of gradient descent as

$$[\mathbf{W}^{(k+1)}, \mathbf{b}^{(k+1)}] = [\mathbf{W}^{(k)}, \mathbf{b}^{(k)}] - \alpha \left[\frac{\partial J}{\partial \mathbf{W}}, \frac{\partial J}{\partial \mathbf{b}}\right] = [\mathbf{W}^{(k)}, \mathbf{b}^{(k)}] - \frac{\alpha}{n}(\mathbf{A} - \mathbf{Y}) \begin{bmatrix} \mathbf{X} \\ \mathbf{1}_n \end{bmatrix}' \tag{80}$$

6 Machine Learning In Finance

6.1 Deep Hedging

Credit 6.1. *Josef Teichmann and Wahid Khosrawi: Lectures on Machine Learning in Finance.*

Deep Hedging goes back to the following paper [5] by Hans Bühler, Lukas Gonon, Josef Teichmann and Ben Wood.

The main idea is to parametrize the hedging strategies (at each time) via neural networks which can depend on input variables chosen by the user, for instance the current price, the past strategy, etc. This then allows to solve a potentially high dimensional hedging problem for many assets whose dynamics are described by an arbitrary given arbitrage free model even in the presence of transaction costs.

Let us exemplify first the idea by the Black Scholes model in one dimension.

Let T be a finite time horizon and consider on a filtered probability space $(\Omega, (\mathcal{F}_{0\leq t\leq T}), \mathcal{F}_T, P)$ a standard Black Scholes model with interest rate $r = 0$ and the price of the risky asset S being described by

$$dS_t = S_t \mu dt + S_t \sigma dW_t^{\mathbb{P}}, \quad S_0 = S_0$$

under the physical measure $\mathbb{P}$. Here $\mu \geq 0$, $\sigma \neq 0$, $S_0 > 0$ and $W^{\mathbb{P}}$ is a Brownian motion (under $\mathbb{P}$).

Under the unique risk neutral probability measure, denoted by $\mathbb{Q}$, the dynamics are then given by

$$dS_t = S_t \sigma dW_t, \quad S_0 = S_0$$

where W is a $\mathbb{Q}$ Brownian motion

We consider here the problem of hedging a $\mathcal{F}_T$-measurable claim $f(S_T)$. In the case of the Black Scholes model the hedging strategy can be found by the Delta hedge, i.e.

$$\Delta(t, s) = \partial_s \mathbb{E}_Q[f(S_T)|S_t = s].$$

In more involved models this is no longer possible. In particular in incomplete models not every claim can be hedged and we thus need to optimize a hedging criterion. We here consider a *quadratic hedging criterion* but other risk

measures are of course also possible.

Let π denote the price of the option, i.e. $\mathbb{E}_Q[f(S_T)]$. Then the goal is solve the following optimization problem

$$\inf_{H \text{ predictable}} \mathbb{E}[(f(S_T) - \pi - (H \bullet S)_T)^2],$$

where (H_t) ranges over all predictable process and $(H \bullet S)_T = \int_0^T H_t dS_t$ denotes the stochastic Ito integral. Optimizing over all predictable processes is infeasable.

Therefore we choose to specify H_t in a smaller set: for each t as a neural network whose input can be specified. In complete Markovian models, as it is the case of the Black Scholes model, we know from the delta hedging strategy that it makes sense to parameterize H_t as a function of the current price S_t. In the current setting we therefore choose that the input of each neural network in the implementation below depends only on the current price, i.e.

$$H_t = g_t(S_t)$$

and g_t denotes a neural network.

We can view the above as supervised learning problem: the input data x_i correspond to trajectories of $(S_t(\omega_i))_{0 \leq T}$, the output y_i should be 0 and the the loss function is given by

$$\mathcal{L} = \frac{1}{K} \sum_i \left(f(S_T(\omega_i)) - \pi - \int_0^T g_t(S_t(\omega_i)) dS_t(\omega_i) \right)^2.$$

To implement this we need to generate input data which will be our training data set. Consider the log price of S_t under $\mathbb{Q}$, i.e.

$$d \log(S_t) = -\frac{\sigma^2}{2} dt + \sigma dW_t.$$

The practical implementation requires a time discretization. If we discretize our time interval $[0, T]$ in N time steps of length T/N we can write

$$\log(S_i) = \log(S_{i-1}) - \frac{\sigma^2}{2} \frac{T}{N} + \sigma \sqrt{\frac{T}{N}} Z_i, i = 1, \ldots, N$$

where Z_i are independent $N(0, 1)$ distributed random variables. The discretized price $(S_0, S_1, \ldots S_N)$ is obtained by exponentiation.

In this disretized form the whole trajetory of the price $(S_0, S_1, \ldots S_N)$ is

therefore determined by $(S_0, Z_1, \ldots, Z_N)$ or in other words $(S_0, X_1, \ldots, X_N)$ where X_i are independent $N(-\frac{\sigma^2}{2}\frac{T}{N}, \sigma^2\frac{T}{N})$ distributed random variables. Considering K many samples thereof constitutes the input training data set. The outputs are simply K zeros.

In the above loss function we also need to disretize the stochastic integral

$$\int_0^T g_t(S_t(\omega))dS_t(\omega).$$

We do this by choosing N neural networks $g_0, \ldots, g_{N-1}$ and disretizing the integral as follows:

$$\sum_{i=0}^{N-1} g_i(S_i(\omega))(S_{i+1}(\omega) - S_i(\omega)).$$

6.1.1 Deep Hedging exemplified by means of the Black Scholes model

Let T be a finite time horizon and consider on a filtered probability space $(\Omega, (\mathcal{F})_{0\leq t\leq T}, \mathcal{F}_T, P)$ a standard Black Scholes model with interest rate $r = 0$ and the price of the risky asset S being described by

$$dS_t = S_t\mu dt + S_t\sigma dW_t^{\mathbb{P}}, \quad S_0 = S_0$$

under the physical measure $\mathbb{P}$. Here $\mu \geq 0$, $\sigma \neq 0$, $S_0 > 0$ and $W^{\mathbb{P}}$ is a Brownian motion (under $\mathbb{P}$).

Under the unique risk neutral probability measure, denoted by $\mathbb{Q}$, the dynamics are then given by

$$dS_t = S_t\sigma dW_t, \quad S_0 = S_0$$

where W is a $\mathbb{Q}$ Brownian motion

We consider here the problem of hedging a $\mathcal{F}_T$-measurable claim $f(S_T)$. In the case of the Black Scholes model the hedging strategy can be found by the Delta hedge, i.e.

$$\Delta(t, s) = \partial_s \mathbb{E}_Q[f(S_T)|S_t = s].$$

In more involved models this is no longer possible. In particular in incomplete models not every claim can be hedged and we thus need to optimize a hedging criterion. We here consider a *quadratic hedging criterion* but other risk

measures are of course also possible.

Let π denote the price of the option, i.e. $\mathbb{E}_Q[f(S_T)]$. Then the goal is solve the following optimization problem

$$\inf_{H \text{ predictable}} \mathbb{E}[(f(S_T) - \pi - (H \bullet S)_T)^2],$$

where (H_t) ranges over all predictable process and $(H \bullet S)_T = \int_0^T H_t dS_t$ denotes the stochastic Ito integral. Optimizing over all predictable processes is infeasable.

Therefore we choose to specify H_t in a smaller set: for each t as a neural network whose input can be specified. In complete Markovian models, as it is the case of the Black Scholes model, we know from the delta hedging strategy that it makes sense to parameterize H_t as a function of the current price S_t. In the current setting we therefore choose that the input of each neural network in the implementation below depends only on the current price, i.e.

$$H_t = g_t(S_t)$$

and g_t denotes a neural network.

We can view the above as supervised learning problem: the input data x_i correspond to trajectories of $(S_t(\omega_i))_{0 \leq T}$, the output y_i should be 0 and the the loss function is given by

$$\mathcal{L} = \frac{1}{K} \sum_i \left(f(S_T(\omega_i)) - \pi - \int_0^T g_t(S_t(\omega_i)) dS_t(\omega_i) \right)^2.$$

To implement this we need to generate input data which will be our training data set. Consider the log price of S_t under $\mathbb{Q}$, i.e.

$$d \log(S_t) = -\frac{\sigma^2}{2} dt + \sigma dW_t.$$

The practical implementation requires a time discretization. If we discretize our time interval $[0, T]$ in N time steps of length T/N we can write

$$\log(S_i) = \log(S_{i-1}) - \frac{\sigma^2}{2} \frac{T}{N} + \sigma \sqrt{\frac{T}{N}} Z_i, i = 1, \ldots, N$$

where Z_i are independent $N(0, 1)$ distributed random variables. The discretized price $(S_0, S_1, \ldots S_N)$ is obtained by exponentiation.

In this disretized form the whole trajetory of the price $(S_0, S_1, \ldots S_N)$ is

therefore determined by $(S_0, Z_1, \ldots, Z_N)$ or in other words $(S_0, X_1, \ldots, X_N)$ where X_i are independent $N(-\frac{\sigma^2}{2}\frac{T}{N}, \sigma^2\frac{T}{N})$ distributed random variables. Considering K many samples thereof constitutes the input training data set. The outputs are simply K zeros.

In the above loss function we also need to disretize the stochastic integral

$$\int_0^T g_t(S_t(\omega))dS_t(\omega).$$

We do this by choosing N neural networks $g_0, \ldots, g_{N-1}$ and disretizing the integral as follows:

$$\sum_{i=0}^{N-1} g_i(S_i(\omega))(S_{i+1}(\omega) - S_i(\omega)).$$

6.1.2 Code

For code, please see this GitHub repository [30].

7 Unsupervised Learning

We now focus on machine learning methods for which only the $p \times n$ matrix $\mathbf{X}$ of inputs is available, that is to say, for which there are no labels associated with each input.

7.1 Anomaly detection

Anomaly detection is a type of machine learning method that falls somewhere between supervised and unsupervised learning and is particular important in the banking industry and insurance industries, where it typically takes the form of *fraud detection*. As before, let $\mathbf{X}$ be the $p \times n$ matrix representing the n inputs, each with p features. Assume that the feature vector can be modelled by a multivariate Gaussian distribution with mean μ and covariance Σ, that is, assume that probability density for the random variable $\mathbf{x} \in \mathbb{R}^p$ is

$$f(\mathbf{x}; \mu, \Sigma) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)'\mathbf{\Sigma}^{-1}(x - \mu)\right), \tag{81}$$

where μ is a $p \times 1$ vector and Σ is a $p \times p$ matrix. We can then estimate the parameters of this distribution using the training data $\mathbf{X}$ as follows:

$$\widehat{\mu} = \frac{1}{n} \sum_{j=1}^{n} \mathbf{x}_j \tag{82}$$

$$\widehat{\Sigma} = \frac{1}{n-1} (\mathbf{X} - \mu \mathbf{1}_p')(\mathbf{X} - \mu \mathbf{1}_p')'. \tag{83}$$

Given this estimates, we declare a test point $\mathbf{x}^{val}$ to be an anomaly if

$$f(\mathbf{x}^{val}; \widehat{\mu}, \widehat{\Sigma}) < \epsilon \tag{84}$$

for some predetermined small value $\epsilon > 0$. This method works well provided p is relatively small. For applications where the number of features is large (say $p = 10,000$), then the number of parameters in Σ becomes too large, leading to overfitting as well as severe limitations in calculating the inverse Σ^{-1}. In these cases, it is generally preferable to assume that the features are independent and estimate a covariance matrix of the form $\Sigma = \text{diag}[\sigma_1^2, \ldots, \sigma_p^2]$ instead.

So far we have treated anomaly detection purely as an unsupervised learning algorithm. However, if we have a dataset with labels that separate the normal from anomalous cases, we can first designate a larger portion of the dataset as the training set as above, that is, used for the purpose of estimating the parameters $\widehat{\mu}$ and $\widehat{\Sigma}$, and then use smaller portions of the data containing some labelled anomalies for the purpose of cross validation and testing. Specifically, the cross validation dataset can be used to adjust the parameter ϵ so that all the anomalies in this portion of the data are correctly detected, whereas the validation dataset can then be used to compute the overall test error of the algorithm.

This should be contrasted with other supervised learning methods that could be used for the same problem. For example, we could try and train a neural network on the same set of features using the labelled dataset with normal cases and anomalies. In many applications, however, the number of anomalies is disproportionately small compared to normal cases, and do not follow any clear pattern that can be reliable "learned" by a neural network. In these cases, the simpler algorithm presented in this section tends to produce much better results at a fraction of the computational cost.

7.2 Principal Component Analysis

This unsupervised learning method consists of finding linear combinations of the original features that explain large parts of the variability of the data. Given an input matrix $\mathbf{X}$ that has already been pre-processed so that each feature (that is, each row) has zero mean, by a *first principal component* we mean a linear combination

$$z_{1j} = \sum_{i=1}^{p} \phi_{i1} x_{ij} = \phi_{11} x_{1j} + \phi_{21} x_{2j} + \cdots \phi_{p1} x_{pj} \tag{85}$$

such that the variance

$$\frac{1}{n-1} \sum_{j=1}^{n} z_{1j}^2 = \frac{1}{n-1} \sum_{j=1}^{n} \left(\sum_{i=1}^{p} \phi_{i1} x_{ij} \right)^2 \tag{86}$$

is as large as possible, subject to the constraint

$$\sum_{i=1}^{p} \phi_{i1}^2 = 1. \tag{87}$$

The solution

$$\widehat{\phi}_1 = \begin{bmatrix} \widehat{\phi}_{11} \\ \vdots \\ \widehat{\phi}_{p1} \end{bmatrix} \tag{88}$$

to this optimization problem is called the *loading* vector of the first principal component, whereas the linear combinations

$$z_{1j} = \widehat{\phi}_{11} x_{1j} + \widehat{\phi}_{21} x_{2j} + \cdots \widehat{\phi}_{p1} x_{pj} = \widehat{\phi}_1' \mathbf{x}_j, \qquad j = 1, \ldots, n \tag{89}$$

are called the *scores* for the first principal component, which reduces the p dimensional vector $\mathbf{x}_j$ corresponding to the j-th data point to a single number z_{1j}. In other words, the vector $\widehat{\phi}_1$ defines a direction in the p–dimensional feature space with the property that the projections $\widehat{\phi}_1' \mathbf{x}_j$ onto this direction have the largest possible variance. Alternatively, we can characterize the direction defined by $\widehat{\phi}_1$ as the line with the smallest average Euclidean distance to all data points.

Having found the first principal component loading vector $\widehat{\phi}_1$, we can then

look for a linear combination

$$z_{2j} = \sum_{i=1}^{p} \phi_{i2} x_{ij} = \phi_{12} x_{1j} + \phi_{22} x_{2j} + \cdots \phi_{p2} x_{pj} \tag{90}$$

such that the variance

$$\frac{1}{n-1} \sum_{j=1}^{n} z_{2j}^2 = \frac{1}{n-1} \sum_{j=1}^{n} \left(\sum_{i=1}^{p} \phi_{i2} x_{ij} \right)^2 \tag{91}$$

is as large as possible, subject to the constraints

$$\sum_{i=1}^{p} \phi_{i2}^2 = 1 \qquad \text{and } {\phi_2}' \widehat{\phi}_1 = 0. \tag{92}$$

In other words, we seek to maximize the variance among all directions that are orthogonal to the first principal component. As before, we can characterize the plane defined by the vectors $\widehat{\phi}_1$ and $\widehat{\phi}_2$ as the plane with the smallest average Euclidean distance to all data points.

For $p < n$, this process can continue until we find exactly p orthogonal directions, each with the property of maximizing the variance while begin orthogonal to the previously found directions. It can be shown that the solutions to these successive optimization problems are related to the eigenvalues and eigenvectors of the $p \times p$ covariance matrix

$$\mathbf{C} = \frac{1}{n-1} \mathbf{X}\mathbf{X}'. \tag{93}$$

Specifically, because $\mathbf{C}$ is a symmetric matrix, it can be expressed as

$$\mathbf{C} = \mathbf{U}\mathbf{D}\mathbf{U}' \tag{94}$$

where $\mathbf{U}$ is a $p \times p$ orthonormal matrix whose columns are the eigenvectors of $\mathbf{C}$ and $\mathbf{D}$ is a $p \times p$ diagonal matrix whose entries are the eigenvalues of $\mathbf{C}$. Moreover, because $\mathbf{C}$ is positive definite, its non-negative eigenvalues can be represented as $\sigma_1^2 \geq \sigma_2^2 \geq \ldots \geq \sigma_p^2$. It then follows that the sum of the variances of each original feature over the entire the data set is given by

$$\sum_{i=1}^{p} \mathrm{Var}(x_i) = \sum_{i=1}^{p} \left(\frac{1}{n-1} \sum_{j=1}^{n} x_{ij}^2 \right) = \sum_{i=1}^{p} C_{ii} = \sum_{i=1}^{p} \sigma_i^2 \tag{95}$$

Using this decomposition, it is not difficult to show that the principal component vectors $\widehat{\phi}_1, \widehat{\phi}_2, \ldots, \widehat{\phi}_p$ are exactly the orthonormal eigenvectors of the symmetric matrix $\mathbf{C}$, ordered from the largest to the smallest eigenvalue, that is

$$\mathbf{U} = [\widehat{\phi}_1, \widehat{\phi}_2, \ldots, \widehat{\phi}_p]. \tag{96}$$

It then follows from (93) and (94) that

$$\frac{1}{n-1}\mathbf{U}'\mathbf{X}\mathbf{X}'\mathbf{U} = \mathbf{D} = \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_p^2 \end{bmatrix} \tag{97}$$

But we recognize the i-th row of the matrix $\mathbf{U}'\mathbf{X}$ as the scores corresponding to the i-th principal component, namely

$$(\mathbf{U}'\mathbf{X})_{ij} = \widehat{\phi}_1'\mathbf{x}_j. \tag{98}$$

Therefore, what (97) states is that

$$\mathrm{Var}(z_i) = \frac{1}{n-1}\sum_{j=1}^{n} z_{ij}^2 = \sigma_i^2, \tag{99}$$

that is, the variance of the scores associated with the i-th principal component is equal to σ_i^2.

It follows that the *proportion of variance explained* (PVE) by the i-th principal component is given by

$$\frac{\mathrm{Var}(z_i)}{\sum_{i=1}^{p}\mathrm{Var}(x_i)} = \frac{\sigma_i^2}{\sum_{i=1}^{p}\sigma_i^2}, \tag{100}$$

whereas the cumulative PVE for the first m principal components is

$$\frac{\sum_{i=1}^{m}\mathrm{Var}(z_i)}{\sum_{i=1}^{p}\mathrm{Var}(x_i)} = \frac{\sum_{i=1}^{m}\sigma_i^2}{\sum_{i=1}^{p}\sigma_i^2}, \tag{101}$$

which clearly equals to one when $m = p$. Plotting the PVE against the number of principal components leads to a decreasing function that can be used in the so-called "elbow method" to determined the number of components to use for further analysis of the dataset.

Having selected a number $M \leq p$ of principal components, we obtain a

reduced representation of the data as the $M \times n$ matrix whose columns are the scores of the first M principal components, that is

$$\mathbf{Z} = \mathbf{U}_M' \mathbf{X} = [\widehat{\phi}_1, \ldots, \widehat{\phi}_M]' \mathbf{X}. \tag{102}$$

Conversely, given the $M \times 1$ vector $\mathbf{z}_j$ we can recover an approximation of the original data point as

$$\widetilde{\mathbf{x}}_j = \mathbf{U}\mathbf{z}_j. \tag{103}$$

The first example below is adapted from ISLR.

Example 7.1. *Principal component analysis for US arrests data*

```
> library(ISLR)
> View(USArrests)
> apply(USArrests , 2, mean)
> apply(USArrests , 2, var)
> pr.out=prcomp(USArrests, scale=TRUE)
> names(pr.out)
> pr.out$center   # mean of the features
> pr.out$scale    # std of the features
> pr.out$rotation
> biplot(pr.out, scale=0)
> pr.out$rotation=-pr.out$rotation
> pr.out$x=-pr.out$x
> pr.out$rotation
> biplot(pr.out, scale=0)
> pr.out$sdev
> pr.var=pr.out$sdev ^2
> pve=pr.var/sum(pr.var)
> plot(pve, xlab="Principal Component", ylab="Proportion of Variance Explained ",
ylim=c(0,1),type='b')
> plot(cumsum(pve), xlab="Principal Component ", ylab=" Cumulative Proportion of
Variance Explained ", ylim=c(0,1), type='b')
```

The next two examples are adapted from DAS.

Example 7.2. *Principal component analysis for NCAA data.*

```
> ncaa = read.table("ncaa.txt",header=TRUE)
```

```
> x = ncaa [4:14]
> result = prcomp(x, scale=TRUE)
> screeplot (result , type="lines")
> biplot ( result )
```

Example 7.3. *Principal component analysis for Interest Rate data*

```
> tryrates = read.table("tryrates.txt",header=TRUE)
> View(tryrates)
> rates = as.matrix(tryrates[2:9])
> result = prcomp(rates)
> biplot(result)
```

7.2.1 K-means clustering

Whereas PCA attempts to summarize data by finding combinations of the features that explain most of its variability, clustering attempts to aggregate data according to similarity. As before, we represent data points as $\mathbf{x}_j \in \mathbb{R}^p$, for $j = 1, \ldots, n$. By *clusters* we mean sets of indices $C_1, \ldots, C_K$ such that $j \in C_k$ if the j-th data point belongs to the k-th cluster. We further impose that $C_1 \cup C_2 \cup \ldots \cup C_K = \{1, \ldots, n\}$ and that $C_k \cap C_{\tilde{k}} = \emptyset$ for all $k \neq \tilde{k}$, that is, each observation is assigned to one, and only one, cluster.

Furthermore, let us define the *within-cluster variation* as the average *squared Euclidean distance* between points in the cluster, that is,

$$W(C_k) = \frac{1}{|C_k|} \sum_{j,\tilde{j} \in C_k} \|\mathbf{x}_j - \mathbf{x}_{\tilde{j}}\|^2 = \frac{1}{|C_k|} \sum_{j,\tilde{j} \in C_k} \sum_{i=1}^{p} (x_{ij} - x_{i\tilde{j}})^2. \tag{104}$$

We can then express K-means clustering as trying to find clusters with the smallest possible total within-cluster variation, that is,

$$\min_{C_1,\ldots,C_K} \sum_{k=1}^{K} W(C_k). \tag{105}$$

This is a computationally hard problem, if only because there are K^n different ways to assign the n data points to K clusters. In what follows, we present a simple algorithm that is guaranteed to find a *local* minimum for (105). The algorithm is based on the straightforward observation that (104) can be rewritten

as

$$W(C_k) = 2 \sum_{j \in C_k} \sum_{i=1}^{p} (x_{ij} - \bar{x}_{ik})^2, \tag{106}$$

where

$$\bar{x}_{ik} = \frac{1}{|C_k|} \sum_{j \in C_k} x_{ij} \tag{107}$$

is the mean of the i-th feature among observations in the k-th cluster. In other words, minimizing the average Euclidean distance between all pairs of observations in a cluster is equivalent to minimizing the sum of the Euclidean distances from each observation to the mean of all observations in the cluster. This remark leads to the following algorithm for K-means clustering:

1. Assign a random cluster from 1 to K to each observation
2. Iterate until the cluster assignments do not change:
   (a) Compute the *centroid* vector $\overline{\mathbf{x}}_k = [\bar{x}_{1k}, \bar{x}_{2k}, \cdots, \bar{x}_{pk}]'$ for each $k = 1, \ldots, K$.
   (b) Reassign each observation fo the cluster with the closest centroid vector.

Because of (106), it is clear that each iteration of this algorithm can only decrease the total within-cluster variation, so that the algorithm is guaranteed to converge to a local minimum. To improve the performance, we can repeat the algorithm with different random initial assignments and pick the final clustering with the lowest total within-cluster variation, which is then a better approximation for the global minimum.

To decide how many clusters to use, we can plot the resulting total within-cluster variation as a function of the number of clusters and use the same type of "elbow method" mentioned earlier for PCA. Another delicate issue associated with the algorithm just presented, and with all clustering methods in general, is that applying a transformation to the features (for example normalizing them by subtracting the mean and dividing by the range) can have a large impact on the final assignment of clusters.

7.2.2 Hierarchical Clustering

A deficiency of the K-means clustering algorithm is that it requires the number of clusters to be determined in advance. As an alternative, we consider the

following *hierarchical clustering* algorithm:

1. Initialize each observation as its own cluster
2. For $k = n, n-1, \ldots, 2$: compute all pairwise inter-cluster dissimilarities among the k clusters and fuse the pair that are most similar.

The function used to compute the dissimilarity between clusters depends on the *linkage* that is assumed for members of a cluster. The most commonly used linkages are the following:

- Centroid linkage: dissimilarity between the centroid for one cluster and the centroid for another.
- Average linkage: compute all pairwise dissimilarities between observations in one cluster and observations in the other cluster and record the *average* dissimilarity.
- Single linkage: compute all pairwise dissimilarities between observations in one cluster and observations in the other cluster and record the *smallest* dissimilarity.
- Complete linkage: compute all pairwise dissimilarities between observations in one cluster and observations in the other cluster and record the *largest* dissimilarity.

Once the algorithm is completed, it results in a *dendrogram*, or an inverted tree-like structure graph exhibiting the links between clusters at each stage in the algorithm, with the first step shown at the bottom, corresponding to each separate observation in a single leaf, and each subsequent step moving up, with the height indicating the value of dissimilarity at which two clusters were fused. We can then select an appropriate vertical level of the dendrogram and readily identify the clusters associated with that level. Figure 22 shows the resulting dendrogram for an example dataset using different linkages.

Figure 21: Example of a dendrogram (left) for hierarchical clustering of 9 points on the plane (right) using complete linkage for inter-cluster dissimilarity. Source: Figure 10.10 in ISLR.

Figure 22: Dendrograms in hierarchical clustering for an example dataset using three different linkage measures. Source: Figure 10.12 in ISLR.

8 An introduction to Crypto Assets

The topic of cryptocurrencies, and more generally crypto assets, lies in the intersection of economics, cryptography, and computer science. In this section, we briefly review the portions of these three huge subjects that are relevant for an introduction to the subject. The discussion and examples are heavily based on the downloadable version of the book *Bitcoin and Cryptocurrency Technologies* by Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, Steven Goldfeder, which we refer to as BCT in the sequel.

8.1 The economics of Money

In its traditional form, cash is an *anonymous*, *permissionless* asset issued by the state that can be used as a *store of value* and *medium of exchange*. The simplest use of cash consists of a payment, for example in the form of a $100 bill, made by one agent, say Alice, to another, say Bob, in exchange of goods. Nowadays, a more common way to proceed would be for Alice to make an electronic transfer for the same amount to Bob. But this typically introduces one layer of intermediation in the form of a bank as follows.

Assuming that Alice and Bob use the same bank, an electronic transfer from Alice to Bob consists of a decrease in the amount of deposits held by Alice and a simultaneous increase in the amount of deposits held by Bob. Notice that this is neither anonymous, in the sense that the bank knows the identity of both parties in the transaction, nor permissionless (in the sense that both Alice and Bob had to first be approved to have deposit accounts in the bank. Moreover, the deposits themselves were not issued by the state, but are instead liabilities of the bank, which were acquired by Alice and Bob when they deposited cash with the bank (or received from another source, say their employer). That is to say, the deposit accounts used in the electronic transfer are treated as *money*, but only have some of the properties of actual cash.

The situation is even more complicated when Alice and Bob use different banks. In this case, the decrease in Alice's deposit account (a liability for her bank) is matched by a decrease in the amount of reserves that the bank holds with the Central Bank (an asset for the bank), while the increase in Bob's

deposit account (a liability for his bank) is matched by an increase in the amount of reserves of his bank. That is to say, the transfer of reserves between banks is what plays the role of an actual transfer of physical cash from one vault to another. In other words, here the reserves also have some of the properties of money (for example being a liability of the state), while not being the same thing as cash either (for example neither anonymous nor permissionless).

The main idea behind a cryptocurrency consists of achieving the same conveniences of an electronic transfer while maintaining as many characteristics of physical cash as possible. In particular, a strong motivation in the development of cryptocurrencies was to eliminate the need for a bank to intermediate the transactions and use the properties of a network of users instead.

An illustrative precursor of this notion consists of the system of stone money known as Rai used in Yap, an island group in the Pacific Ocean. Because of their size (up yo 4 meters in diameter) when these stones are used as payment for goods what is actually transferred between agents the their *ownership*, rather than the stones themselves, which are not physically moved when a transaction occurs. The Yapese keep record of who owns which stones through an elaborate network representing the oral history of the island. When a payment is made using Rai, the oral history is update in a way that prevents *double spending*, that is to say, the same stone being used for payment more than once. This is made possible because of the relatively simple nature of the network, with all stone owners effectively knowing each other and agreeing on a common record. The challenge of cryptocurrencies is to reproduce this feature in the much more complex and larger network of all potential users around the world. In order to do this, they rely on rather sophisticated concepts in cryptography and a lot of computer power.

8.2 A crash course on Cryptography

8.2.1 Hash functions

The first concept needed to understand cryptocurrencies is that of a *hash function*, namely a function that accepts input in the form of any numerical string of arbitrary size n and computes an output of fixed size in time of order $\mathcal{O}(n)$. For concreteness, let us consider outputs that consist of binary strings of length $n = 256$. As an example, consider the hash function

$$H(x) = x \bmod 2^{256}, \tag{108}$$

that is to say, the remainder after division by 2^{256}.

We then say that a hash function H is *collision resistant* if it is infeasible to find values $x \neq y$ such that $H(x) = H(y)$. It is easy to see that the function in 108 is not collision resistant, as for example $H(5) = H(5 + 2^{256}) = 101$ (recall that 101 is 5 in binary format, where we omitted the first 253 digits that are equal to zero in the output string).

For a different example, consider the so-called mid-squares hash function, namely a function that squares an arbitrary input and then extracts a fixed number of middle digits, say 4. For example, the square of $x = 123,456$ is 15,241,383,936 from which we can extract the 4 middle digits (ignoring the high digit) 1,383, so that $H(x) = 1383$ (notice that in this example we are not representing the output string in binary format). It is now much harder to come up with another input y such that $H(y) = 1383$ (try it!), although it can be shown that this particular example is not collision resistant either. As a matter of fact, no known hash function has been rigorously proved to be collision resistant, while some that were used in earlier applications of cryptography, such as the MD5 function, were later shown to fail to be collision resistant and subsequently phased out.

A popular application of collision resistant hash functions is what is called a *message digest*, namely a representation of an input message x by a hash function $H(x)$ that can be used to verify that there was no tampering with the contents of the message during transmission. To see this, suppose that a very large file with content x is uploaded to the cloud, while the much smaller hash function $H(x)$ is sent to someone by email. After downloading the supposedly transmitted file with content $\hat{x}$ from the cloud, all that the recipient has to do in order to verify that there was no tampering is to calculate $H(\hat{x})$ and check if $H(x) = H(\hat{x})$, since for a collision resistant hash function H, any tampering would most likely result in $H(x) \neq H(\hat{x})$

Moving on, we would like to say that a hash function H is *hiding* if, given an output $c = H(x)$, it is infeasible to figure out the input x. The problem with this potential definition is that if the inputs can only take a few values, then it is always very easy to figure out which of them produced a given output simply by computing the corresponding hash function. To remedy this, we should first *concatenate* the input x with a secret value r to form the modified input $r \,||\, x$. Here r should be chosen from a probability distribution with *high min-entropy*, meaning that no particular value is very likely to occur. For example, think of r as chosen uniformly from all strings that are 256 digits long. We then say

that a hash function H is *hiding* if, when a secrete value r is chosen from a probability distribution with high min-entropy, then it is infeasible to find x given an output $c = H(r \mid\mid x)$.

An application of such hash functions is what is known as a *commitment scheme*, namely a commit function of the form

$$c = \text{commit}(r, x) \tag{109}$$

and a verify function of the form

$$\text{verify}(c, r, x) = \begin{cases} \text{TRUE if } c = \text{commit}(r, x) \\ \text{FALSE otherwise} \end{cases} \tag{110}$$

In other words, the commitment scheme is the digital analogue of sealing a value x in an envelope and putting it on the table where everyone can see (namely the hash value c). At a later point, the value x can be revealed and verified that it was indeed the message committed to earlier. In this context, the random value r is referred to as a *nonce*, meaning that it is a value that can only be used once. Using a hiding, collision resistant hash function, we can implement a commitment scheme simply by setting

$$\text{commit}(r, x) = H(r \mid\mid x) \tag{111}$$

and the verify function as being true provided $c = H(r \mid\mid x)$ and false otherwise.

The third property that we are going to require is that the hash function should be *puzzle friendly*, meaning that for every possible n-bit output z, if k is chosen from a distribution with high min-entropy, then it is infeasible to find x such that $z = H(k \mid\mid x)$ in time significantly less than 2^n.

As an application of this property, consider what is called a *search puzzle*, namely a hash function H, a puzzle-ID k chosen from a high min-entropy distribution, and a target set Z. We say that a solution to this puzzle is a value x such that

$$H(k \mid\mid x) \in Z. \tag{112}$$

The larger the set Z, the easier the puzzle. By making Z restricted enough, for example by imposing that $H(k \mid\mid x)$ should be smaller than a given threshold (which means that at least a certain number of its first digits should be equal to zero), the puzzle friendliness property ensures that there is no effective way to

solve the puzzle other than brute force trial of a large number of random values of x.

As a final application of hash functions, let us now define a *hash pointer*, namely a pointer to where a particular block of information is stored in a data structure, together with the value of a hash function of the information. Next consider what is called a regular *linked list*, namely a series of blocks of information together with a pointer to the previous block in the list. Putting the two concepts together we arrive at a *block chain*, namely a series of blocks of information together with a hash-pointer to the previous block.

Figure 23: An example of a hash pointer, consisting of the location of some information, plus a digest of its content. Figure 1.4 in BCT.

Figure 24: An example of a block chain, where not only we know the location of the previous block, but also a digest of its content. Figure 1.5 in BCT.

Suppose now that a malicious actor tampers the data in block k of the chain. Then simply by reading the value of the hash pointer in block $k+1$, which is part of the information contained in the block, we are able to detect the tampering. If the malicious actor further alters the hash pointer in block $k+1$ to hide their actions, this altered data will then lead to an inconsistent value for the hash pointer in the next block. Continuing in this way, we are able to detect tampering in *any* previous block simply by computing the hash value of the last block.

A simple modification of this idea leads to the *Merkle tree* shown in Figure 25, which is also a way to detect tampering of the data by simply observing the value of the hash function at the root of the tree. Moreover, such tree allows one to verify *proof of membership* for a particular block of data simply by exhibiting the hash function of the path from the block to the root, which consists of only about $\log(n)$ items. If we then sort the blocks at the bottom of the tree using some agreed-on order (say alphabetical, numerical, etc), we can use the same mechanism to verify proof of nonmembership in the tree.

The hash function used in the Bitcoin protocol is called *SHA-256* (Secure Hash Algorithm-256) and outputs fixed-length 256-bit strings for arbitrary lengths inputs in a way that is believed to be collision-resistant, hiding, and

puzzle friendly. For example, the SHA256 hash for the string "hello" is[6]

$$H(\text{hello}) = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e730433629 38b9824 \tag{113}$$

whereas the SHA256 hash for the very similar string "Hello" is

$$H(\text{Hello}) = 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969 \tag{114}$$

In these example, observe that the output is a 64-bit string in hexadecimal format, which can easily be converted into a 256-bit string in binary format.

Figure 25: Merkle tree. Figure 1.7 in BCT.

8.2.2 Digital Signatures

Start by observing that the two distinct properties of a physical signature are that only the person signing should be able to make it, but anyone can verify it, and that the signature should be tied to a particular document. With this in mind, we define a *digital signature scheme* as consisting of a method to generate a secret ket sk, which is kept privately, and a public key pk provided for verification, that is,

$$(sk, pk) := \text{generateKeys}(keysize), \tag{115}$$

together with a method to sign messages using the private key, that is,

$$sig := \text{sign}(sk, message), \tag{116}$$

followed by a method to verify the validity of the signature, that is,

$$isValid := \text{verify}(pk, message, sig). \tag{117}$$

These methods should have the properties that signatures should be *existentially unforgeable* and that

$$\text{verify}(pk, message, \text{sign}(sk, message)) == \text{true}. \tag{118}$$

[6]See https://emn178.github.io/online-tools/sha256.html

The *unforgeability* property is usually characterized in the form of a game between an attacker and a challenger as shown in Figure 26. We say that the signature is existentially unforgeable if the chances of the attacker winning the game are extremely small. Observe that by combining digital signatures with a hash function one is able to sign messages of arbitrary length by simply signing the digest of its content. Moreover, by signing a hash pointer, one is able to effectively sign an entire data structure, such as a block chain or a Merkle tree. Moreover, observe that the public key generated in a digital signature scheme can be used a a proxy for the *identity* of the person using it. Because anyone can generate such keys, this leads to a decentralized identity management system, where anyone can register in the system simply by creating new keys. In the context of Bitcoin, for example, these identities are called *addresses*, which simply consists of the hash of a public key.

Figure 26: The unforgeability game: an attacker sends a polynomial number of messages to the challenger and receives signed documents and then signs a previously unseen message M. If this signature is verified, the attacker wins. Figure 1.9 in BCT.

8.3 Toy examples of cryptocurrencies

Instead of dealing directly with the complexities of a real cryptocurrency like Bitcoin, let us illustrate the main ideas with two toy examples called *Goofycoin* and *Scroogecoin*. Goofycoin operates with two simple rules:

1. An actor called Goofy can create new coins with a unique ID by generating the string `CreateCoin[uniqueCoinID]` and then sign it using his secret key.

2. Whoever owns a coin (say Alice) can transfer it to someone else (say Bob) by generating the string `Pay H( ) to` pk_{Bob} and sign it, where $H()$ is a hash pointer to the data structure representing the ownership history of the coin.

These rules are illustrated in Figure 27. An obvious flaw in this system is that Alice can transfer the same coin to two different actors, say Bob and Charlie, who now both possess equally valid data structures representing the same coin, which corresponds to a *double spending attack* by Alice.

Figure 27: Transfer of ownership of Goofycoins. Figure 1.10 in BCT.

Our next toy example, Scroogecoin, is designed to solve this problem. It builds upon Goofycoin by adding an actor called Scrooge who publishes an *append-only ledger* containing the history of all transactions in the system in the form of a block chain as shown in Figure 28 and signs it. The transactions in this system take the form of `CreateCoins`, signed by Scrooge only, and `PayCoins`, signed by everyone who is transferring a coin, as shown in Figures 29 and 30 below. Once Scrooge verifies that a `PayCoin` transaction is valid, in particular by checking that no double spending has occurred, he appends it to the block chain and signs the last hash pointer.

Figure 28: Transaction history of Scroogecoins.

Figure 29: An example of a CreateCoins transaction in Scroogecoins. Figure 1.12 in BCT.

Figure 30: An example of a PayCoins transaction in Scroogecoins. Figure 1.13 in BCT.

The obvious problem with this system is that it depends too heavily on Scrooge! The solution to this *centralization* problem offered by Bitcoin consists of eliminating Scrooge from the system and make a transaction acceptable only when a puzzle associated with it has been solved. For example, given `data` corresponding to a new block, one looks for a hash such that

$$H(\text{data} \,||\, \text{nonce}) = 000\,000\,000\,0\ldots \tag{119}$$

We recognize this as an example of a search puzzle with $k = \text{data}$ being the puzzle-ID and $x = \text{nonce}$ being the random values that need to be found by brute force. This is a difficult problem to solve, requiring lots of computing (and energy!) resources, but once it is solved it can be easily verified by anyone in the system, at which point the new block (and all of its transactions) can be added to the chain, in addition to one more `CreateCoins` transaction to reward the first person who solves the puzzle and broadcasts it to the network. At

this point, the strategy of all other actors updating their own network with the validate information becomes a Nash equilibrium: if everyone believes that this is what others are doing, there is no inventive to deviate (for example continue searching for a solution so that one can get the reward instead). As an example, for the Bitcoin block #69785 that was added to the block chain at 12:09:36 CET on July 23rd, 2010, the hash value was

$$000000000293b78a2833b45d78e97625f6484ddd1accbe0067c2b8f98b57995$$

9 Lattice Methods

9.1 Models for Stock Prices

Stock prices observed in typical financial markets arise from the combined decisions of a large number of traders who are meant to act (somewhat) rationally based on the information available at the time of trade. The result of such complex interactions are prices that change rapidly and unpredictably in time, as can be observed in any financial news service.

One way to model these prices mathematically is to assume that they are given by a continuous-time *stochastic processes* S_t. For example, the popular Geometric Brownian Motion model proposed by Samuelson in 1965 is described by the stochastic differential equation

$$dS_t = \mu S_t dt + \sigma S_t dW_t \tag{120}$$

and attempts to capture the fact that logarithmic returns on stock prices are roughly normally distributed with constant mean μ and variance σ. It is well documented that (120) does not account for the statistical properties of real stock prices, such heavy-tailed distributions, jumps and non-constant volatility. Accordingly, several other models for continuous time stochastic processes have been proposed in the past decades.

Whatever the model for S_t might be, one should also take into account that data from financial time series is necessarily presented as a discrete time realization of the underlying stochastic process. For example, for the ultra-high-frequency time series available from financial data sources, the observations are presented as *tick-data*, that is, whenever a transaction occurs. This leads to delicate issues for dealing with time series observed at random times, since transactions generally do not occur at pre-specified times. A simpler alternative is to consider only observations over equally spaced time intervals, say daily or monthly.

For modeling purposes, we can achieve further simplification if we assume that the state space for S_t is also discrete, that is, we take the underlying sample space to be $\Omega = \{\omega_1, \omega_2, \ldots\}$, so that asset prices can only take a discrete set of values $S_t(\omega_1), S_t(\omega_2), S_t(\omega_3), \ldots$ at each instant t. The combined effect of discretizing both time and state space is to cover the set $\mathbb{R}_+ \times \mathbb{R}_+$ by a *lattice*. Financial time series then corresponds to specific paths on the lattice. The most popular choice of lattice structure is the binomial model described next.

9.2 One-period binomial model

Consider times $t_0 = 0$ and $t_1 = T$ and two processes B_t and S_t representing a cash account and a stock. We assume that the value for the cash account is deterministic and given by

$$B_0 = 1, \qquad B_T = e^{rT}, \tag{121}$$

where r is the continuously compounded annualized interest rate. For the stock price, we assume that

$$S_0 = s, \qquad S_T = \begin{cases} us, & \text{with probability } p \\ ds, & \text{with probability } 1-p \end{cases}, \tag{122}$$

with $d < u$. That is, we implicitly consider a sample space $\Omega = \{\omega_1, \omega_2\}$ with $P(\omega_1) = p > 0$ and $P(\omega_2) = 1 - p > 0$.

A *portfolio* in this model is a vector $H = (x, y) \in \mathbb{R}^2$, where x and y represents the number of units invested in the cash account and the number of shares held at time $t = 0$, respectively. Therefore, the value of the portfolio H at any time $t = 0, T$ is the process

$$V_t^H = xB_t + yS_t.$$

An *arbitrage* in this model is a portfolio H such that

1. $V_0^H = 0$
2. $V_T^H(\omega_1) \geq 0$ and $V_T^H(\omega_2) \geq 0$.
3. At least one of $V_T^H(\omega_1)$ or $V_T^H(\omega_2)$ is strictly positive.

A probability measure Q on the sample space $\Omega = \{\omega_1, \omega_2\}$ is called an *equivalent martingale measure* if $Q(\omega_1) > 0$ and $Q(\omega_1) > 0$, and

$$S_0 = e^{-rT} E^Q[S_T]. \tag{123}$$

In other words, the expected growth rate of S under the measure Q is the risk free interest rate r, which explains why equivalent martingale measures are also called *risk-neutral measures*.

The next proposition is the simplest version of the Fundamental Theorem of Asset Pricing (FTAP) and is left to be proved as an exercise.

Proposition 9.1. *The one-period binomial model is arbitrage free if and only if there exists a martingale measure Q given by*

$$Q(\omega_1) = q := \frac{e^{rT} - d}{u - d}, \quad Q(\omega_2) = 1 - q = \frac{u - e^{rT}}{u - d} \tag{124}$$

We now turn to the problem of pricing options in the one period binomial model. We define a *contingent claim* as a random variable of the form $C = \Phi(S)$ for some function $\Phi : \mathbb{R} \to \mathbb{R}$. The claim is said to be *replicable* if there exist a portfolio H with $V_T^H = C$. The next proposition shows that, in the absence of arbitrage, replicable claims have a well defined value at time $t_0 = 0$. Its proof is a simple application of the definition of arbitrage in this simplified model.

Proposition 9.2. *If a contingent claim C can be replicated by a portfolio H, then the unique price at time $t_0 = 0$ that is consistent with absence of arbitrage is $c_0 = V_0^H$.*

In other words, replicable claims can be uniquely priced in terms of the underlying assets in the market. When all claims are replicable, the market is said to be *complete*.

Proposition 9.3. *The one-period binomial model is complete.*

Proof. For an arbitrary claim $C = \Phi(S)$, we want to find $H = (x, y)$ such that

$$\begin{aligned} xe^{rT} + yus &= C^u := \Phi(us) \\ xe^{rT} + yds &= C^d := \Phi(ds) \end{aligned}$$

This system has a unique solution given by

$$x = e^{-rT} \frac{uC^d - dC^u}{u - d} \tag{125}$$

$$y = \frac{C^u - C^d}{s(u - d)} \tag{126}$$

□

From the previous propositions, the arbitrage free price of a contingent claim $C = \Phi(S)$ in the one period binomial model is

$$c_0 = V_0^H = x + ys = e^{-rT} \frac{uC^d - dC^u}{u - d} + \frac{C^u - C^d}{u - d}.$$

A slight rearrangement of this expression leads to

$$c_0 = e^{-rT}[qC^u + (1-q)C^d] = e^{-rT}E^Q[C]. \tag{127}$$

In other words, the price of C is given by its discounted expectation under the equivalent martingale measure Q. Moreover, we can interpret (126) as saying that the number of shares held in the replicating portfolio is given by the ratio of the change in claim value $\Delta C = C^u - C^d$ and the change in stock value $\Delta S = s(u-d)$, which is commonly called a *delta hedging*. Observe further that the originally defined probabilities p for stock price changes in (122) play no role whatsoever in either pricing or hedging a contingent claims. All that is needed are the possible values of the stock, as specified by the parameters u and d, and the risk-neutral probabilities q, which in turn is uniquely determined by u, d and r according to (124).

9.3 The binomial tree

Consider now an equally spaced partition $0 = t_0 < \ldots < t_n < \ldots < t_N = T$ of the interval $[0, T]$ into N subintervals of length $\Delta t = T/N$. We extend the one-period binomial model to this multi-period setting by take the price dynamics for the cash account to be

$$B_0 = 1, \quad B_n = e^{r\Delta t}B_{n-1} \tag{128}$$

and that of the stock price to be

$$S_0 = s, \qquad S_{n+1} = \begin{cases} uS_n, & \text{with probability } p \\ dS_n, & \text{with probability } 1-p \end{cases}, \tag{129}$$

for $n = 1, \ldots, N-1$.

A *portfolio* in this model is a random vector $H_n = (x_n, y_n)$ corresponding to holding in the bank account and stock at time t_{n-1}. The portfolio is said to be *self-financing* if no money is either withdrawn or injected whenever there is a rebalancing (i.e change in weights) from one period from another, that is,

$$x_n B_{n+1} + y_n S_{n+1} = x_{n+1}B_{n+1} + y_{n+1}S_{n+1}, \tag{130}$$

where the left-hand side is the value of the portfolio at time t_{n+1} obtained from the holdings used in time t_n, whereas the right-hand side is the value after

rebalancing. This means that the change in value of a self-financing portfolio from t_n to t_{n+1} is given by

$$\begin{aligned} V_{n+1} - V_n &= \left(x_{n+1}B_{n+1} + y_{n+1}S_{n+1}\right) - \left(x_nB_n + y_nS_n\right) \\ &= \left(x_nB_{n+1} + y_nS_{n+1}\right) - \left(x_nB_n + y_nS_n\right) \\ &= x_n\left(B_{n+1} - B_n\right) + y_n\left(S_{n+1} - S_n\right), \end{aligned} \tag{131}$$

with (131) sometimes being used as the definition of self-financing instead of (130).

An *arbitrage* in this model is a self-financing portfolio satisfying

1. $V_0^H = 0$.
2. $V_T^H \geq 0$.
3. $E[V_T^H] > 0$

A probability measure Q is said to be an *equivalent martingale measure* if

$$0 < q := Q(S_{n+1} = uS_n) < 1$$

and

$$S_n = e^{-r\Delta t}E^Q[S_{n+1}|S_n].$$

Finally, in this multi-period model, a contingent claim $C = \Phi(S)$ is said to be *replicable* if there exists a self-financing portfolio satisfying $V_T^H = C$.

With a little bit more work (and somewhat tedious inductive arguments) one can prove the exact analogues of the three propositions from the previous section. The results are summarized as follows:

Proposition 9.4. *1. The multi-period binomial model is arbitrage free if and only if there exists an equivalent martingale measure Q given by*

$$q = \frac{e^{r\Delta t} - d}{u - d} \tag{132}$$

2. If a claim C is replicable, then its unique arbitrage-free value at intermediate times t_n is $c_n = V_n$.

3. The multi-period binomial model is complete, and the replicating portfolio

for a claim C is given by

$$x_n = e^{-rT}\frac{uV^d_{n+1} - dV^u_{n+1}}{u-d} \tag{133}$$

$$y_n = \frac{V^u_{n+1} - V^d_{n+1}}{S_n(u-d)} \tag{134}$$

where $V^u_{n+1} = V_{n+1}(uS_n)$ and $V^d_{n+1} = V_{n+1}(dS_n)$ are the values of the replicating portfolio conditional on the stock price in period $n+1$.

Combining the results in this proposition, we conclude that the value of the replicating portfolio for a claim C (and therefore its unique arbitrage-free price) can be calculated recursively as

$$V_n(S_n) = e^{-r\Delta t}[qV^u_{n+1} + (1-q)V^d_{n+1}|S_n] = e^{-r\Delta t}E^Q[V_{n+1}|S_n], \tag{135}$$

for $n = 0, \ldots, N-1$ and $V_T(S_T) = \Phi(S_T)$.

9.4 The continuum limit and the choice of parameters

By a continuum limit of the multi-period binomial model we mean the model obtained when $\Delta t \to 0$ (or equivalently when $N \to \infty$). According to the discussion in the previous section, pricing and hedging require that we specify the dynamics of S_t under the risk-neutral measure, so the only parameters we need to specify are u, d and q. Different limits will arise depending on the choice of parameters. One possibility is to require convergence to the Geometric Brownian motion

$$dS_t = rS_t dt + \sigma S_t dW^Q_t, \tag{136}$$

where W^Q denotes a Brownian motion in the risk-neutral measure Q. Our task is to make the multi-period binomial model consistent with a discrete-time approximation for (136), for example the *Euler scheme*

$$\Delta S_n := S_{n+1} - S_n = rS_n\Delta t + \sigma S_n \epsilon\sqrt{\Delta t}, \tag{137}$$

where $\epsilon \sim N(0,1)$.

For this, observe that (137) implies that

$$E^Q[S_{n+1} - S_n|S_n] = rS_n\Delta t,$$

that is,

$$E^Q[S_{n+1}|S_n] = S_n(1 + r\Delta t) \approx S_n e^{r\Delta t}. \tag{138}$$

On the other hand, in the multi-period binomial model we have

$$E^Q[S_{n+1}|S_n] = quS_n + (1-q)dS_n. \tag{139}$$

Equating (138) and (139) leads to

$$q = \frac{e^{r\Delta t} - d}{u - d}, \tag{140}$$

which we recognize as the martingale condition (132). Moreover, the approximation (137) implies that

$$\mathrm{Var}^Q[S_{n+1} - S_n|S_n] = \sigma^2 S_n^2 \Delta t,$$

that is,

$$\mathrm{Var}^Q[S_{n+1}|S_n] = \sigma^2 S_n^2 \Delta t. \tag{141}$$

On the other hand, the variance implied by the multi-period binomial model is

$$\mathrm{Var}^Q[S_{n+1}|S_n] = E^Q[S_{n+1}^2|S_n] - \big(E^Q[S_{n+1}|S_n]\big)^2 = (qu^2 + (1-q)d^2)S_n^2 - e^{2r\Delta t}S_n^2 \tag{142}$$

Equating (141) and (142) leads to the consistency condition

$$(u + d)e^{r\Delta t} - du - e^{2r\Delta t} = \sigma^2 \Delta t. \tag{143}$$

So far we guaranteed that the mean and variance in the multi-period binomial model agree with those of the Euler approximation for (136). It follows from the properties of Brownian motion (mostly the normal distribution of its increments) that these two conditions are enough to imply that the multi-period binomial model converges to (136) in distribution. Since we have three parameters to determine, conditions (140) and (143) need to be supplemented by another condition on u, d and q, leading to the different models proposed in the literature.

For example, in the *Cox-Rubenstein-Ross* (CRR) model, the extra condition is that $ud = 1$. This leads to the following expressions, which satisfy (140) and

(143) up to $\mathcal{O}(\Delta t)$:

$$\begin{aligned} u &= e^{\sigma\sqrt{\Delta t}} \\ d &= e^{-\sigma\sqrt{\Delta t}} \\ q &= \frac{e^{r\Delta t} - e^{-\sigma\sqrt{\Delta t}}}{e^{\sigma\sqrt{\Delta t}} - e^{-\sigma\sqrt{\Delta t}}} \end{aligned} \tag{144}$$

For another example, the *Jarrow-Rudd* (JR) model is based on the extra condition $q = 1/2$, leading to the following expressions satisfying (140) and (143) up to $\mathcal{O}(\Delta t)$:

$$\begin{aligned} u &= 1 + r\Delta t + \sigma\sqrt{\Delta t} \\ d &= 1 + r\Delta t - \sigma\sqrt{\Delta t} \\ q &= \frac{1}{2} \end{aligned} \tag{145}$$

9.5 American Options

The contingent claims treated in the previous sections could only be exercise at a fixed maturity time T and are generally called *European options*. By contrast, a contingent claim that gives its holder the right to exercise it at any time $0 \leq t \leq T$ prior to maturity is called an *American option*.

The canonical example is that of an American put option with payoff of the form

$$\Phi(S) = (K - S)^+. \tag{146}$$

If at a given time t we have that $S_t \leq K$, we might be tempted to exercise the option and obtain $(K - S_t)$. However, it could happen that holding the option for longer produces an even better payoff. Accordingly, the optimal strategy corresponds to the solution to the problem

$$V_t = \sup_{\tau \in [t,T]} E^Q[e^{-(\tau - t)r}(K - S_\tau)|S_t] \tag{147}$$

In continuous time, this problem does not admit a closed-form solution. On a binomial tree, however, it can be easily solved by replacing expression (135) with

$$V_n(S_n) = \max\{K - S_n, e^{-r\Delta t}[qV_{n+1}(uS_n) + (1 - q)V_{n+1}(dS_n)]\}, \tag{148}$$

for $n = 0, \ldots, N-1$ and $V_T(S_T) = (K - S_T)^+$. In other words, at each step of the tree, the value for the option is given as the maximum between its *exercise value* given by the payoff $K - S_n$ and its *continuation value* given by the discounted risk-neutral expectation $e^{-r\Delta t} E^Q[V_{n+1}(S_{n+1})|S_n]$.

For each time t_n, it is clear that the exercise value for the put option will be negligible at high stock prices S_n. As we move down a column on the tree, both the exercise value and the continuation value increase. The first node where the exercise value becomes larger than the continuation value corresponds to the *exercise threshold* for the time t_n, denoted by $S^*_{t_n}$. In the continuous time limit, we obtain a curve S^*_t called the *exercise boundary* for the put option. If the stock price at time t satisfies $S_t \leq S^*_t$, then the put option should be exercised, otherwise it should be held for longer.

9.6 Further Lattice Methods

There are several ways to extend the basic binomial model to incorporate more complicated asset price dynamics and more elaborate options on lattices. For example:

- trinomial trees
- lattice discretization of stock prices S following a jump-diffusion process
- path dependent options on a tree (e.g barrier options);
- trees with stochastic interest rates
- options depending on multiple underlying assets

As an example, we consider the first item above, namely an extension of the binomial tree model consisting of assuming that, instead of (129), the stock price in each period satisfies

$$S_{n+1} = \begin{cases} uS_n, & \text{with probability } q_u \\ S_n, & \text{with probability } 1 - q_u - q_d \\ dS_n, & \text{with probability } q_d \end{cases}, \tag{149}$$

for $n = 1, \ldots, N-1$. Observe that for this tree to be *recombining*, we need to impose that $ud = 1$, whereas this is not necessary in the binomial tree. Notice also that we are already expressing the probabilities for the different moves of the stock in terms of the risk-neutral measure Q, as this is what is relevant for

pricing. With this in mind, we have that the parameters u, q_u, q_d must be chosen in order to match the conditions (138) and (141) that we encounter before, in order to guarantee that, in the limit $\Delta t \to 0$, the trinomial tree model converges to the continuous-time model in (136). In other words, even with the additional condition $ud = 1$, there exist a family of trinomial trees consistent with the same continuous-time limit.

It is a straightforward exercise to verify that the following popular parametrization satisfy these conditions:

$$\begin{aligned} u &= e^{\sigma\sqrt{2\Delta t}} \\ q_u &= \left(\frac{e^{r\frac{\Delta t}{2}} - e^{-\sigma\sqrt{\frac{\Delta t}{2}}}}{e^{\sigma\sqrt{\frac{\Delta t}{2}}} - e^{-\sigma\sqrt{\frac{\Delta t}{2}}}} \right)^2 \\ q_d &= \left(\frac{e^{\sigma\sqrt{\frac{\Delta t}{2}}} - e^{r\frac{\Delta t}{2}}}{e^{\sigma\sqrt{\frac{\Delta t}{2}}} - e^{-\sigma\sqrt{\frac{\Delta t}{2}}}} \right)^2 \end{aligned} \tag{150}$$

9.7 Examples

Example 9.5. Consider the European call option with maturity $T = 1$ on a stock with $S_0 = 50$, $K = 50$, $r = 0.1$ and $\sigma = 0.4$. For a binomial tree with $\Delta t = 1/52$ (weekly intervals), the parameters for the CRR model are $u = 1.0570$, $d = 0.9460$ and $q = 0.5035$, resulting in $c_{CRR} = 10.12$ for this option (see implementation below). For comparison, the Black-Scholes price for this option is $c_{BS} = 10.16$ (see implementation below).

You can use the Matlab code below to verify the results in the previous example and try other examples in the binomial model. The code is adapted from *Numerical Methods in Finance and Economics: a MatLab-based introduction*, Paolo Brandimarte, Wiley Series in Probability and Statistics, 2nd edition, 2006.

Example 9.6. *Black-Scholes pricing function*

```
function [call,put] = blsprice(so,x,r,t,sig,q)
%BLSPRICE Black-Scholes put and call pricing.
%   Reference: Bodie, Kane, and Marcus, Investments, page 681.
%
%   See also BLSIMPV, BLSDELTA, BLSGAMMA, BLSLAMBDA, BLSTHETA, BLSRHO.

%       Author(s): C.F. Garvin, 2-23-95
```

```
%       Copyright 1995-2002 The MathWorks, Inc.
%       $Revision: 1.8 $   $Date: 2002/03/11 19:19:02 $

if nargin < 5
  error(sprintf('Missing one of SO, X, R, T, and SIG.'))
end
if any(so <= 0 | x <= 0 | sig < 0)
  error(sprintf('Enter SO, and X > 0. Enter SIG >= 0.'))
end
if nargin < 6
  q = zeros(size(so));
end

% prevent divide by zero warning : the functions do the right thing
Warnstate = warning;
warning('off');

d1 = (log(so./x)+(r-q+sig.^2/2).*t)./(sig.*sqrt(t));
d2 = d1 - (sig.*sqrt(t));
call = so.*exp(-q.*t).*normcdf(d1)-x.*(exp(-r.*t).*normcdf(d2));
put = x.*exp(-r.*t).*normcdf(-d2)-so.*exp(-q.*t).*normcdf(-d1);

% set warning state back
warning(Warnstate);
```

Using the parameters of Example 9.5 we find:

```
>> price_BS =  blsprice(50,50,0.1,1,0.4,0)

price_BS =

   10.1592
```

Example 9.7. *European call option price in a binomial model on a rectangular grid*

```
function [price, grid,S] = EuropeanCallGridCRR(S0,K,r,T,sigma,N)
dt = T/N;
u=exp(sigma * sqrt(dt));
```

```
d=1/u;
q=(exp(r*dt) - d)/(u-d);

% stock values

S = zeros(2*N+1,1);
S(N+1) = S0;
for i=1:N
    S(N+1+i) = d*S(N+i);
    S(N+1-i) = u*S(N+2-i);
end

grid = zeros(2*N+1,N+1);          % initalization for the grid

grid(1,:)=(S(1)-K)*ones(1,N+1);   % boundary condition at the top of the grid (highest value

grid(:,N+1)=max(0,S-K);           % boundary condition at the terminal time step


% working backwards on entire columns at once

for j=N:-1:1

    % option value one step later at an up tick)
    aux_up=grid(1:2*N-1,j+1);       % column j+1 shifted one row down

    % option value one step later at a down tick)
    aux_down=grid(3:2*N+1,j+1);     % column j+1 shifted one row up

    grid(2:2*N,j)=exp(-r*dt)*(q*aux_up+(1-q)*aux_down);

end
price = grid(N+1,1);
```

Using the parameters of Example 9.5 we find:

```
>> price_CRR = EuropeanCallGridCRR(50,50,0.1,1,0.4,52)
```

```
price_CRR =

   10.1220
```

Example 9.8. *American put option price in a binomial model on a rectangular grid*

```
function [price,grid,S,t] = AmPutGrid(S0,K,r,T,sigma,N)

% Precompute invariant quantities
deltaT = T/N;
t=[0:deltaT:T];
u=exp(sigma * sqrt(deltaT));
d=1/u;
q=(exp(r*deltaT) - d)/(u-d);
discount = exp(-r*deltaT);
q_u = discount*q;
q_d = discount*(1-q);

% set up S values
S = zeros(2*N+1,1);
S(N+1) = S0;
for i=1:N
    S(N+1+i) = d*S(N+i);
    S(N+1-i) = u*S(N+2-i);
end

grid = zeros(2*N+1,N+1);          % initalization for the grid

grid(2*N+1,:)=K*ones(1,N+1);   % boundary condition at the top of the grid (highest value fo

grid(:,N+1)=max(0,K-S);           % boundary condition at the terminal time step


% working backwards on entire columns at once

for j=N:-1:1
```

```
    % option value one step later at an up tick
    aux_up=grid(1:2*N-1,j+1);       % column j+1 shifted one row down

    % option value one step later at a down tick
    aux_down=grid(3:2*N+1,j+1);     % column j+1 shifted one row up

    hold=q_u*aux_up+q_d*aux_down;
    grid(2:2*N,j)=max(hold,K-S(2:2*N));

end
price = grid(N+1,1);
```

10 Monte Carlo Methods

10.1 Introduction

Suppose we want to determine the area of a region A within a unit square. If we choose N random points inside the square, then the area can be approximated by the ratio N'/N, where N' is the number of points that happen to lie inside A. This elementary procedure immediately raises the following questions: (i) what distribution should we use to generate the random points and (ii) how accurate can we make the approximation?

To generalize this idea, suppose we want to determine a quantity μ. Assume that we can sample n independent random variables $X_1, \ldots, X_n$ with $E[X_i] = \mu$ and $\mathrm{Var}[X_i] = \sigma^2$. Next define the sum

$$S_n = X_1 + \cdots + X_n, \tag{151}$$

which satisfy $E[S_n] = n\mu$ and $\mathrm{Var}[S_n] = n\sigma^2$. Then the random variable S_n/n has expected value equal to μ and variance equal to σ^2/n. As $n \to \infty$, it is intuitively clear that this random variable should converge to the non-random number μ, since its variance goes to zero. This is indeed the content of the following theorem:

Theorem 10.1 (Strong Law of Large Numbers)**.** *Suppose that $X_1, \ldots, X_n$ are independent random variables with $E[X_i] = \mu$ and $E[X_i^4] \leq k$ for some $k \geq 0$ and let $S_n = X_1 + \cdots X_n$. Then*

$$\frac{S_n}{n} \to \mu \quad \textit{(a.s.)}$$

This result provides an answer to the first question above: in principle any distribution (with finite forth moments) can be used for the random variables X_i, since in the limit $n \to \infty$ we can obtain the desired number μ with certainty as the expected value of S_n/n. It provides no answer, however, to the second question, which can now be rephrased as follows: given that we necessarily need to stop our calculations at some finite n, how accurate is the approximation for μ given by $E[S_n/n]$? For this we need the following fundamental theorem from probability:

Theorem 10.2 (Central Limit Theorem)**.** *Suppose that $X_1, X_2, \ldots$ is a sequence of i.i.d. random variables with mean μ and finite variance σ^2 and let*

$S_n = X_1 + \cdots X_n$. *Then*

$$\lim_{n\to\infty} P\left(\frac{S_n - n\mu}{\sigma\sqrt{n}} < x\right) = \Phi(x),$$

where $\Phi(\cdot)$ *denotes the cumulative distribution function for a standard normal random variable.*

Therefore, the asymptotic distribution for the random variable S_n/n is $N(\mu, \sigma/\sqrt{n})$, which can be used to measure the accuracy of the approximation of μ by $E[S_n/n]$.

10.2 Monte Carlo Integration

The prototypical application for the scheme proposed in the previous section is the problem of computing an integral of the form

$$I = \int_0^1 f(x)dx, \tag{152}$$

where $f : \mathbb{R} \to \mathbb{R}$ is a known function.

If X is uniformly distributed on the interval $[0, 1]$, then it follows that

$$E[f(X)] = \int_0^1 f(x)dx = I.$$

Therefore, if $X_1, \ldots, X_n$ are independent samples for the random variable X, we obtain that

$$I_n = \frac{1}{n}\sum_{i=1}^{n} f(X_i) \tag{153}$$

is an unbiased estimator for I with

$$I_n \to I \qquad a.s.$$

Example 10.3. Let $I = \int_0^1 e^{x^2} dx$. The following commands implement Monte Carlo approximations for this integral using uniform random variables generated by the command `rand` and compare them with the value obtained by the built-in numerical integration routine `quad`:

```
>> I=sum(exp(rand(100,1).^2))/100
```

```
I =
    1.4792

>> I=sum(exp(rand(1000,1).^2))/1000

I =
    1.4514

>> I=sum(exp(rand(10000,1).^2))/10000

I =
    1.4694

>> Iexact=quad('exp(x.^2)',0,1)

Iexact =
    1.4627
```

10.3 Random Number Generators

The procedure described in the previous sections relies on the ability to sample the random variable X. The most direct way to generate such *random numbers* is by using some physical process whose outcomes, at least in theory, satisfy some known probability distribution. Rudimentary examples of these processes are dice rolling, coin tossing and roulette spinning, whose results are deemed to be random according to the theory of unstable dynamical systems. More reliable processes involve microscopic phenomenon, such as thermal noise. The results of a large sequence of outcomes for one of these hardware generators can then be tested for randomness using well-known statistical criteria. After testing and filtering out unreliable outcomes, the random numbers can be recorded in a table for subsequent use. The first large scale reliable *table of random numbers* was produced by the RAND corporation in 1955 and contained one million uniformly distributed random numbers.

Instead of using true random numbers coming from a physical process, one can use sequences of numbers that are generated by well defined algorithms, and are therefore not random in essence. Despite their algorithmic origin, if these number pass the statistical tests for randomness, they can be used for

all practical purposes as if they were random. For this reason, they are called *pseudo-random numbers* and are vastly used for scientific computations.

An elementary algorithm for generating pseudo-random numbers with uniform distribution is the *middle-of-squares* method: start with a four digits number, square it and take the four digits in the middle as the new number in the sequence. For example,

$$\begin{aligned} z_1 = 9876 &\rightarrow z_1^2 = 97535376 \\ z_2 = 5353 &\rightarrow z_2^2 = 28654609 \\ z_3 = 6546 &\rightarrow z_3^2 = 42850116 \end{aligned}$$

As expected, this simple algorithm performs poorly in statistical tests. For example, it generates too many small numbers. Another problem is that it contains *traps*, that is periodic sequences such as $(6100, 2100, 4100, 8100, 6100, ...)$ which clearly should not be present in bona fide random sequences.

A more sophisticated algorithm is the *Linear Congruential Generator* (LCG), which starts with a seed z_0 and proceeds to calculate subsequent numbers according to the formula

$$z_i = (az_{i-1} + b)(\text{mod m}) \tag{154}$$

The numbers z_i/m are then deemed to be $U(0, 1)$. The obvious drawback of this algorithm is that it produces only rational numbers in the interval $[0, 1]$. It is also periodic with maximum period equal to m, and often much smaller. In practice, the paramters a, b and m are chosen so that the sequence performs well in statistical tests. For example, the popular *Numerical Recipes in C* advocates

$$a = 1664525, \quad b = 1013904223, \quad m = 2^{32}.$$

For Monte Carlo simulation, however, even an LCG with long period and well chosen parameters produces unacceptable sequences (for example with high serial correlation). Commercial software such as MATLAB use more robust (and more complicated) algorithms, such as the Mersenne Twister algorithm developed in 1997.

10.4 Sampling from a given distribution

The methods discussed in the previous section produce pseudo-random numbers with uniform distribution on the unit interval $[0, 1]$. From now on, these will be

taken as given and used to obtain samples for random variables with arbitrary distributions.

10.4.1 Inverse Transform Method

Let X be a random variable with cumulative distribution $F(x) = P(X \leq x)$. It is easy to show that $F : \mathbb{R} \to [0, 1]$ is monotonically increasing. If we assume further that F is strictly increasing, then $F^{-1} : [0, 1] \to \mathbb{R}$ is well defined. In this case, if Y_i are samples from a uniform random variable on $[0, 1]$, we have that $X_i = F^{-1}(Y_i)$ satisfies

$$P(X_i \leq x) = P(F^{-1}(Y_i) \leq x) = P(Y_i \leq F(x)) = F(x).$$

Therefore, this *inverse transform* method produces an arbitrary random variable X from a uniform random variable Y, provided the cumulative distribution of X is invertible.

This is clearly not the case for discrete random variables, since the cumulative distribution is piecewise constant and jumps at the points $x_1, \ldots, x_n$ where $P(X = x_i) = p_i$, $i = 1, \ldots, n$. In this case, if $Y \sim U(0, 1)$, then

$$\begin{aligned} P(0 \leq Y < p_1) &= p_1 \\ P(p_1 \leq Y < p_1 + p_2) &= p_2 \\ \vdots &\quad \vdots \\ P(1 - p_n \leq Y \leq 1) &= p_n. \end{aligned}$$

Therefore, we can set

$$X = \begin{cases} x_1 \text{ if } Y < p_1 \\ x_2 \text{ if } p_1 \leq Y < p_1 + p_2 \\ \vdots \\ x_n \text{ if } 1 - p_n \leq Y \leq 1 \end{cases}.$$

It is easy to see that this is a modification of the inverse transform method using the generalized inverse of F, which can also be applied to random variables that contain both continuous and discrete parts.

10.4.2 Acceptance-Rejection Method

An obvious limitation of the previous method is that it requires the explicit calculation of the inverse function F^{-1}. In practical situations, the cumulative distribution is either not known explicitly or not easy to invert. Suppose however that X has a known (albeit complicated) density function $p(x)$, that is, suppose that F is given by

$$F(x) = \int_{-\infty}^{x} p(\tilde{x})d\tilde{x}.$$

The *acceptance-rejection* method is designed to produce X in the cases where we can find another function $q(x) \geq p(x)$ such that $q(x)/\int q(x)dx$ is a density for which we already know how to sample. In particular, this always happens when $p(x)$ has a bounded support J, since we can simply take

$$q(x) = \max_{J} p(x),$$

so that $q(x)/\int_J q(x)dx$ corresponds to a uniform distribution on J. We can then sample Z_i from the distribution $q(x)/\int q(x)dx$ and compare the values of $q(Z_i)$ and $p(Z_i)$ to determine if Z_i is a good approximate sample for the random variable X. By construction, we have that $p(Z_i)/q(Z_i) \leq 1$, and the closer this ratio is to 1 the more accurate the approximation. The steps for the method are the following:

1. Sample $U_i \sim U(0,1)$.

2. Sample $Z_i \sim \frac{q(x)}{\int q(x)dx}$.

3. If $U_i \leq \frac{p(Z_i)}{q(Z_i)}$, accept $X_i = Z_i$. Otherwise, reject.

Combining the inverse transform and the acceptance-rejection methods we can generate random variables for a large number of distributions.

10.4.3 Box-Muller Method

Some methods are designed to sample random variables for specific distributions. For example, the *Box-Muller* method generates a pair (X, Y) of independent standard normal random variables, whose joint density function is

$$f(x,y) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}\frac{1}{\sqrt{2\pi}}e^{-y^2/2} = \frac{1}{2\pi}e^{-(x^2+y^2)/2},$$

using the coordinate transformation

$$\begin{aligned} R^2 &= X^2 + Y^2 \\ \theta &= \tan^{-1} \frac{Y}{X} \end{aligned}$$

Since the Jacobian of the inverse transformation is $J = 1/2$, we obtain that

$$f(x,y)dxdy = \frac{1}{2} f(R^2, \theta) dR^2 d\theta = \frac{1}{2\pi} \frac{e^{-R^2}}{2} dR^2 d\theta.$$

Therefore, if R^2 is exponentially distributed with parameter $\lambda = 2$ and θ is uniform on the interval $[0, 2\pi]$, we recognize this last expression as the joint density of the pair of independent random variables (R^2, θ). This suggests the following algorithm to generate the random variables (X, Y):

1. Generate independent random variables $U_1 \sim U(0,1)$ and $U_2 \sim U(0,1)$.
2. Set $R^2 = -2 \log U_1$ and $\theta = 2\pi U_2$.
3. Put $X = R \cos\theta$ and $Y = R \sin\theta$.

10.5 Monte Carlo Error Estimate

Recall that if $E[X] = \mu$ and $\mathrm{Var}[X] = \sigma^2$, the Monte Carlo estimator for μ,

$$\frac{S_n}{n} = \frac{1}{n} \sum_{i=1}^{n} X_i,$$

is (i) *unbiased*, that is $E[S_n/n] = \mu$, (ii) *strongly convergent*, that is $S_n/n \to \mu$ a.s., and (iii) *asymptotically normal*, that is $S_n/n \sim N(\mu, \sigma^2/n)$.

Suppose we are given a confidence level $(1 - \delta)$ and would like to find the number of replications necessary to guarantee that

$$\left| \frac{S_n}{n} - \mu \right| \leq \epsilon,$$

with probability $(1 - \delta)$. Define the critical value z_δ as the solution to

$$\Phi(z_\delta) = 1 - \frac{\delta}{2},$$

where $\Phi(\cdot)$ is the cumulative distribution for a standard normal random variable.

Then with probability $(1-\delta)$ we have that, asymptotically,

$$-z_\delta \frac{\sigma}{\sqrt{n}} \leq \frac{S_n}{n} - \mu \leq z_\delta \frac{\sigma}{\sqrt{n}}.$$

That is, if we set

$$\sqrt{n} = \frac{z_\delta \sigma}{\epsilon}, \tag{155}$$

then, asymptotically, the error in the Monte Carlo estimate S_n/n will be smaller than ϵ with probability $(1-\delta)$.

In practice, since we do not know the true variance σ, we can use the sample variance

$$\bar{\sigma}_k^2 = \frac{1}{k-1} \sum_{i=1}^{k} [X_i - (S_k/k)]^2, \tag{156}$$

for a fixed value of k (say $k = 100$) and then use this estimate to find the number of replications n according to

$$\sqrt{n} = \frac{z_\delta \bar{\sigma}_k}{\epsilon} \tag{157}$$

Numerical integration through quadrature methods, such as the trapezoidal or Simpson's rule, have an error of order $1/n^{2/d}$, where d is the dimension of the integral. As we can see, the error in Monte Carlo integration does not depend on the dimension of the problem, and becomes significantly smaller than that of quadrature methods whenever $d > 4$.

10.6 Variance Reduction

Rearranging (155) we find that the error for a Monte Carlo estimate is given by

$$\epsilon = \frac{z_\delta \sigma}{\sqrt{n}}. \tag{158}$$

Instead of increasing the number of replications, we can try to achieve a smaller error by reducing the variance of the samples themselves.

10.6.1 Antithetic Variates

A straightforward way to do this is by using *antithetic variates*. Suppose that we sample a sequence of identically distributed pairs of random variables $(X_i, \widetilde{X}_i)$ with the property that X_i and $\widetilde{X}_j$ are independent whenever $i \neq j$. Then assuming that $E[X_i] = E[\widetilde{X}_i] = \mu$ and $\mathrm{Var}[X_i] = \mathrm{Var}[\widetilde{X}_i] = \sigma^2$, we have that

the new random variables

$$\widehat{X}_i = \frac{X_i + \widetilde{X}_i}{2}, \quad i = 1, 2, \ldots, n$$

are independent and have $E[\widehat{X}_i] = \mu$. We can then compute a Monte Carlo sum directly from $\widehat{X}_i$ and obtain

$$\frac{S_n}{n} = \frac{1}{n}\sum_{i=1}^{n} \widehat{X}_i \to \mu \qquad (a.s.).$$

if we now calculate the variance for this Monte Carlo sum we find

$$\mathrm{Var}\left(\frac{S_n}{n}\right) = \frac{\mathrm{Var}[\widehat{X}_i]}{n} = \frac{\mathrm{Var}[X_i] + \mathrm{Var}[\widetilde{X}_i] + 2\mathrm{Cov}(X_i, \widetilde{X}_i)}{4n}$$

We can see that, whenever $\mathrm{Cov}(X_i, \widetilde{X}_i) < 0$, we have

$$\mathrm{Var}\left[\frac{S_n}{n}\right] < \frac{\sigma^2}{2n}.$$

This shows that the antithetic variates $(X_i, \widetilde{X}_i)$ produced a reduction in variance for the Monte Carlo sum when compared to $2n$ samples of independent random variables.

A simple way to generate samples with negative correlation is to use $U_i \sim U(0,1)$ as the basic random variables used to generate X_i and then use $(1-U_i)$ to generate $\widetilde{X}_i$. This is because both U_i and $(1-U_i)$ are uniformly distributed on $[0,1]$ and have

$$\mathrm{Cov}(U_i, 1-U_i) = -\frac{1}{12}.$$

Alternatively, we can use $Z_i \sim N(0,1)$ to generate X_i and $-Z_i$ to generate $\widetilde{X}_i$. In this case, both Z_i and $-Z_i$ are standard normal and

$$\mathrm{Cov}(Z_i, -Z_i) = -1.$$

In any case, we need to verify that the negative correlation propagates to X_i and $\widetilde{X}_i$. This will always be the case if the method used to generate X_i and $\widetilde{X}_i$ from the basic random variables (say the inverse transform method) involves a monotone transformation. On the other hand, non-monotone transformations, such as symmetric integrands in the problem of computing $\int_0^1 f(u)du$, might

turn negative correlations into positive ones, in which case antithetic sampling performs even worse than naive Monte Carlo.

10.6.2 Control Variates

Another way to achieve a reduction in variance is by using a *control variate*, that is, a secondary random variable Y, possibly correlated with X, with the property that we somehow know (either from other estimation procedure or from theoretical reasons) its expected value $E[Y]$. We then consider the *controlled* random variable

$$X_c = X + c(Y - E[Y]),$$

where c is a control parameter. It is clear that this random variable can be used in the Monte Carlo sums instead of the original X, since

$$E[X_c] = E[X]$$

for any $c \in \mathbb{R}$. The advantage of using it, however, relies on the fact that we can fine tune its variance, which is given by

$$\mathrm{Var}[X_c] = \mathrm{Var}[X] + c^2\mathrm{Var}[Y] + 2c\mathrm{Cov}(X, Y).$$

We can see that this is a quadratic expression in c, which is minimized at

$$c^* = -\frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}[Y]}.$$

Since we are not assuming that we know anything about Y other than its mean, the value for c^* needs to be estimated using k preliminary Monte Carlo replications (say $k = 50$). Substituting c^* this into the expression fro $\mathrm{Var}[X_c]$, we have that

$$\frac{\mathrm{Var}[X_{c^*}]}{\mathrm{Var}[X]} = 1 - \rho^2_{XY},$$

so that the effectiveness of the method increases with the correlation between X and Y.

10.6.3 Conditioning

Our final method makes use of the general fact that X and $E[X|\mathcal{F}]$ are random variables with the same mean but with variances satisfying

$$\text{Var}[X] = E[\text{Var}[X|\mathcal{F}] + \text{Var}[E[X|\mathcal{F}]],$$

which implies that

$$\text{Var}[X] \geq \text{Var}[E[X|\mathcal{F}]].$$

That is, taking conditional expectations preserves the mean but reduces the variance of the random variable X. A concrete example of this method is what is called *stratified sampling*. Suppose that Y is a secondary random variable whose support J is partitioned into a finite number of disjoint sets $A_1, \ldots, A_k$. Then, it follows from Bayes's rule that

$$E[X] = \sum_{i=1}^{k} E[X|Y \in A_i]P(Y \in A_i). \tag{159}$$

Suppose now that we simulated n_i values for X given that $Y \in A_i$ and label these values $X_{i1}, \ldots, X_{in_i}$. We then have a total of $n = \sum_{i=1}^{k} n_i$ replications of the random variable X, but separated according to the *strata* $A_1, \ldots, A_k$. Using (159), we can approximate $E[X]$ by

$$E[X] \approx \sum_{i=1}^{k} P(Y \in A_i) \left(\frac{1}{n_i} \sum_{j=1}^{n_i} X_{ij} \right).$$

In particular, if we choose the numbers n_i so that $n_i/n = P(Y \in A_i)$ - a criterion called *proportional allocation*, then

$$E[X] = \frac{1}{n} \sum_{i=1}^{k} \sum_{j=1}^{n_i} X_{ij}.$$

Since each of the terms in the sum above is a sample from a random variable with variance smaller than the variance of X (by conditioning), we see that a Monte Carlo sum based on the samples $X_{i1}, \ldots, X_{in_i}$, $i = 1, \ldots k$ is an estimator for $E[X]$ with a smaller variance than the one obtained from direct sampling of X. The method relies, however, on the ability to choose a secondary random variable Y and the sets A_i such that it is feasible to sample X conditioned on

$Y \in A_i$.

10.6.4 Computational Efficiency

In all the methods above, it is necessary to consider a trade-off between the reduction in variance and the associated computational cost required by the method. To formalize this concept, let T be a fixed computational time for the entire task of computing μ by Monte Carlo and let τ be the time per replication of the underlying random variable X with mean μ and variance σ^2. Then the total number of replications is $n = [T/\tau]$ and, according to the theory just presented,

$$\sqrt{\left[\frac{T}{\tau}\right]}\left(\frac{S_n}{n} - \mu\right) \sim N(0, \sigma^2), \text{ as } n \to \infty.$$

That is,

$$\sqrt{T}\left(\frac{S_n}{n} - \mu\right) \sim N(0, \sigma^2\tau), \text{ as } n \to \infty.$$

We see from this last expression that, given a fixed total computational time T, the overall performance for the method is measured by the product $\sigma^2\tau$.

10.7 Pricing European Options

According to the paradigm of risk-neutral valuation, the price at time t of an option with payoff $\Phi \equiv \Phi(S_T)$ is giving by the following discounted expectation

$$\pi_t = E^Q\left[e^{-\int_t^T r_s ds}\Phi\right], \tag{160}$$

where Q is an equivalent martingale measure and r_t denotes the (possibly stochastic) risk-free interest rate.

For example, assuming a constant interest rate r, the initial price of an European call option with payoff $\Phi(S_T) = (S_T - K)^+$ on a stock with price following the Geometric Brownian motion (136) is given by

$$\pi_0 = e^{-rT}E^Q\left[\left(S_0 e^{\left(r - \frac{\sigma^2}{2}\right)T + \sigma W_T} - K\right)^+\right].$$

To find this expected value using Monte Carlo, all we need are n samples of the

random variable

$$S_T^i = S_0 \exp\left[\left(r - \frac{\sigma^2}{2}\right)T + \sigma\sqrt{T}\varepsilon^i\right],$$

where $\varepsilon^i \sim N(0,1)$. Once this is done, the Monte Carlo estimate for the price is simply given by

$$\pi_0 = e^{-rT}\frac{1}{n}\sum_{i=1}^{n}(S_T^i - K)^+.$$

To obtain better confidence intervals for the same number of replications we can apply the method of antithetic variates by using both ε^i and $-\varepsilon^i$. In this case, the negative correlations are preserved since $(S_T - K)^+$ is a monotone function of W_T. Further variance reduction can be obtained if we use S_T as a control variate, since we know its expected value

$$E^Q[S_T] = S_0 e^{rT}.$$

Clearly, the Monte Carlo method is somewhat superfluous to calculate the price of such simple option, for which the Black-Scholes formula is available. The method becomes useful, however, when dealing with more complicated dynamics for S_T, where closed-form expressions are difficult to find.

For example, suppose that the stock price is given by a more general SDE of the form

$$dS_t = \mu(S_t, t) + \sigma(S_t, t)dW_t \tag{161}$$

where $\mu(\cdot,\cdot)$ and $\sigma(\cdot,\cdot)$ are functions satisfying the integrability conditions needed to guarantee that a solution S_t exists. In this case, one case use the *Euler-Maruyama* method to simulate stock price paths recursively as follows

$$S_{t_{j+1}} = S_{t_j} + \mu(S_{t_j}, t_j)\Delta t + \sigma(S_{t_j}, t_j)\sqrt{\Delta t}\epsilon_j, \qquad j = 0, \cdots, k-1 \tag{162}$$

where $S_{t_0} = S_0$ is the starting price, $t_k = T$ is the end time, $\Delta t = T/k$ is a fixed time step and ϵ_j are $N(0,1)$ random variables.

10.8 Pricing Exotic Options

Pretty much any derivative other than European call and put options is termed an exotic option. In general, they present path dependence, in the sense that the price is affected by some property of the path $(S_t)_{0\leq t\leq T}$ rather than just

the terminal value S_T. Nevertheless, the basic risk-neutral valuation formula (160) still applies, and Monte Carlo methods can be used to find the price of exotic options as long as we know how to incorporate the path dependence in the simulation procedure.

10.8.1 Barrier Options

These are options that either come to existence ("in") or cease to exist ("out") provided the stock price S_t crosses a specified barrier S_b at some time $0 \leq t \leq T$. The options are further classified according to the position of the barrier with respect to the initial value of the stock, that is, either $S_0 < S_b$ ("up") or $S_0 > S_b$ ("down").

For instance, an up-and-out call option has a payoff of the form

$$\Phi_1 = (S_T - K)^+ \mathbf{1}_{\{M < S_b\}},$$

where $M = \max_{0 \leq t \leq T} S_t$ and $S_0 < S_b$. Compared with a vanilla call option, this type of option protects the writer against an abnormally high increase in the stock price, and is therefore cheaper.

Another example is a down-and-in put option with payoff

$$\Phi_2 = (K - S_T)^+ \mathbf{1}_{\{m \leq S_b\}},$$

where $m = \min_{0 \leq t \leq T} S_t$ and $S_b < S_0$. In this case, the put option only comes to existence when the stock price is extremely low, and can be considered as a form of insurance against default of the underlying company.

For continuous time, we may assume that the barrier is checked against the current value of the stock continuously. This formulation is useful for obtaining explicit formulas for the values of barrier options using properties of the Brownian motion. In Monte Carlo valuation, however, it is necessary to assume that the barrier is checked at discrete intervals.

For example, consider again a Geometric Brownian motion (136) and constant interest rate r. In this case, the value for an up-and-out call option is given by the expression

$$\pi_0 = e^{-rT} E^Q[(S_T - K)^+ \mathbf{1}_{\{M < S_b\}}]. \tag{163}$$

Differently from the case of an European option, we now need to simulate the

entire path $(S_t)_{0\le t\le T}$ for the stock price and compare it with the barrier at specified times. For the Geometric Brownian motion we can use the property that, for given times $t_j = j\Delta t$, $j = 1, \dots k$, the price satisfies

$$S_{t_j} = S_{t_{j-1}} e^{\left(r-\frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\varepsilon^j}, \qquad S_{t_0} = S_0,$$

where $\varepsilon^j \sim N(0,1)$. Therefore, for each Monte Carlo simulation Φ_1^i for the integrand in (163) we need to sample k normal random variables ε^{ij} and then calculate

$$\Phi_1^i = \begin{cases} S_0 \Pi_{j=1}^k e^{\left(r-\frac{\sigma^2}{2}\right)\Delta t + \sigma\sqrt{\Delta t}\varepsilon^{ij}} & \text{if } S_{t_j} < S_b \text{ for all } j \\ 0 \text{ otherwise} \end{cases}$$

As before, the price is then given by the Monte Carlo sum

$$\pi_0 \approx e^{-rT} \frac{1}{n} \sum_{i=1}^{n} \Phi_1^i.$$

Observe that a large number of replications n results in a small confidence interval for the Monte Carlo estimate, whereas a large number of intermediate times k results in a more frequent comparison between the stock price and the barrier, leading to a more accurate value for the barrier option.

10.8.2 Asian Options

These options have a stronger path dependence in the sense that their payoff depends on the average value of the stock price during the life of the option. For example, an *average strike call* consists of

$$\Phi_3 = (S_T - A)^+,$$

whereas an *average rate call* has payoff

$$\Phi_4 = (A - K)^+.$$

In the formulas above, the method for calculating the average A is chosen among the following:

- *discrete arithmetic*: $A := \frac{1}{k}\sum_{j=1}^{k} S_{t_j}$

- *discrete geometric*: $A := \left(\Pi_{j=1}^{k} S_{t_j}\right)^{1/k}$
- *continuous arithmetic*: $A := \frac{1}{T}\int_0^T S_t dt$
- *continuous geometric*: $A := \exp\left[\frac{1}{T}\int_0^T \log S_t dt\right]$

For the case of an Asian option with discrete arithmetic averaging, an effective variance reduction can be obtained by using $Y = \sum_{j=1}^{k} S_{t_j}$ as a control variate, since its mean is given by

$$E[Y] = E\left[\sum_{j=1}^{k} S_{t_j}\right] = \sum_{j=1}^{k} E[S_{t_j}] = \sum_{j=1}^{k} S_0 e^{rj\Delta t} = S_0 \frac{1 - e^{r(k+1)\Delta t}}{1 - e^{r\Delta t}}$$

10.8.3 Lookback Options

For these options the path dependence comes from a payoff that depends explicitly on the maximum or minimum for the stock over the life of the option. For example, two versions of a lookback put with the maximum of the stock are

$$\Phi_5 = (M - S_T),$$

and

$$\Phi_6 = (K - M)^+,$$

depending on whether the maximum $M = \max_{0 \le t \le T} S_t$ is used to replace the strike price or the final value for the stock.

11 Finite-Difference Methods for Partial Differential Equations

11.1 Introduction

Suppose we want to find a solution $u(t,x)$ to the partial differential equation (PDE) problem

$$L\,u = 0 \qquad \mathcal{D} \in \mathbb{R}^2 \tag{164}$$

$$u = f \qquad \text{on } \partial\mathcal{D} \tag{165}$$

where L is a linear second order partial differential operator, $\mathcal{D}$ is a domain in $\mathbb{R}^2$ and f is a known function specified on the boundary $\partial\mathcal{D}$. We begin with a discretization of $\mathcal{D}$ by means of a regular lattice $\Lambda \in k\mathbb{Z} \times h\mathbb{Z}$, that is, we consider points of the form $t_j = jk$ and $x_i = ih$, or in other words, fixed increments $\Delta t = k$ and $\Delta x = h$.

For a fixed time t, consider now the Taylor expansions for $u(t, x+h)$ and $u(t, x-h)$ around the point (t,x):

$$u(t,x+h) = u(t,x) + h\frac{\partial u}{\partial x} + \frac{h^2}{2}\frac{\partial^2 u}{\partial x^2} + \frac{h^3}{6}\frac{\partial^3 u}{\partial x^3} + \mathcal{O}(h^4) \tag{166}$$

$$u(t,x-h) = u(t,x) - h\frac{\partial u}{\partial x} + \frac{h^2}{2}\frac{\partial^2 u}{\partial x^2} - \frac{h^3}{6}\frac{\partial^3 u}{\partial x^3} + \mathcal{O}(h^4) \tag{167}$$

If we now use the notation[7]

$$u_{i,j} = u(t_j, x_i) = u(jk, ih) \tag{168}$$

we can use (166) to obtain the *forward-difference* approximation for the first order derivative:

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i,j}}{h} + \mathcal{O}(h). \tag{169}$$

Alternatively, we can use (167) to obtain the *backward-difference* approximation

[7] Observe that the usual calculus notation $u(t,x)$ implies that t is represented in the horizontal axis and x in the vertical axis, so that $t = 0, 1, 2, \ldots$ correspond to vertical lines and $x = 0, 1, 2, \ldots$ correspond to horizontal lines. In matrix notation, this means that different values of t correspond to different columns, whereas different values of x correspond to different rows. Because rows are normally denoted by the first index i and columns by the second index j of a matrix element of the form a_{ij}, we arrive at the slightly confusing notation $u_{i,j} = u(t_j, x_i)$.

for the first order derivative:

$$\frac{\partial u}{\partial x} = \frac{u_{i,j} - u_{i-1,j}}{h} + \mathcal{O}(h). \tag{170}$$

Finally, subtracting (167) from (166) leads to the *central-difference* approximation for the first derivative:

$$\frac{\partial u}{\partial x} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + \mathcal{O}(h^2). \tag{171}$$

Similarly, we can add (166) and (167) and obtain a *central-difference* approximation to the second derivative:

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + \mathcal{O}(h^2). \tag{172}$$

Using Taylor expansions for $u(t+k, x)$ and $u(t-k, x)$ around the point (t, x) for a fixed x leads to the corresponding finite-difference approximations for the partial derivatives with respect to t.

11.2 Errors

Consider as an example the problem

$$\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial x^2} = 0, \quad \text{for } 0 \le t \le 1, \quad 0 \le x \le 1 \tag{173}$$

$$u(0, x) = f_1(x), \qquad u(t, 0) = g_1(t) \tag{174}$$

$$\frac{\partial u}{\partial t}(1, x) = f_2(x), \qquad \frac{\partial u}{\partial x}(t, 1) = g_2(t) \tag{175}$$

We see that the boundary conditions for $t = 0$ and $x = 0$ involve the function u itself, whereas the boundary conditions for $t = 1$ and $x = 1$ involve partial derivatives of u, and are examples of *Dirichlet* and *Neumann* boundary conditions, respectively.

For the numerical solution of this problem using finite-differences, we consider the increments $\Delta t = k$ and $\Delta x = h$ and the grid[8]

$$t_j = jk, \qquad j = 0, \dots M, \qquad M = 1/k \tag{176}$$

$$x_i = ih, \qquad i = 0, \dots N, \qquad N = 1/h \tag{177}$$

[8]We chose to start the indices i, j from zero for notational convenience, although this can be easily adjusted for languages that require indices to start from one such as Matlab.

Using the central-difference approximations for the second derivatives and dropping the $\mathcal{O}(h^2)$ terms we find the following approximation for the Laplace equation (173):

$$u_{i,j} = \frac{1}{2(h^2+k^2)}\left[k^2\big(u_{i+1,j}+u_{i-1,j}\big)+h^2\big(u_{i,j-1}+u_{i,j+1}\big)\right], \tag{178}$$

for $i = 1, \ldots, N-1$ and $j = 1, \ldots, M-1$. Notice that we did not include the grid points corresponding to $i = 0$ or N and $j = 0$ or M, as these are determined by the boundary conditions. The error that arises by replacing the PDE in (173) with its finite-difference approximation (178) is called *truncation* error.

Additionally, we need to consider the errors introduced by the presence of derivatives in the boundary conditions. For Dirichlet boundary conditions this is not an issue, as we can set [9]

$$u_{i,0} = f_1(ih), \qquad i = 0, \ldots, N \tag{179}$$

$$u_{0,j} = g_1(jk), \qquad j = 0, \ldots, M \tag{180}$$

For Neumann boundary conditions we need to work a little harder. Using backward-difference approximation for the partial derivatives in (175) and dropping the $\mathcal{O}(h)$ terms, we find

$$u_{i,M} = kf_2(ih) + u_{i,M-1}, \qquad i = 0, \ldots N \tag{181}$$

$$u_{N,j} = hg_2(jk) + u_{N-1,j}, \qquad j = 0, \ldots M \tag{182}$$

The combined error arising from the approximation of the PDE itself and boundary conditions is called the *discretization* error.

Notice that (178) consists of several systems of linear equations that need to be solved simultaneously for the unknowns $u_{i,j}$ with the help of (179)-(182). In the next sections we will see how this can be done efficiently using numerical methods involving matrix algebra in the context of the heat equation, but in all cases the solution of these systems require computations that cannot be done exactly, either because of the floating-point arithmetic used by computers or because of the use of iterative methods, or both. The additional errors arising from the actual solution of the finite-difference approximations using numerical methods is generically called *round-off* error.

[9] Recall from the previous section that the first index in $u_{i,j}$ refers to the variable x.

11.3 Consistency, convergence, and stability

We say that a numerical scheme for solving a PDE is *consistent* if the truncation error goes to zero as $h, k \to 0$, that is to say, if the finite-difference equation tends to the actual PDE, which is always the case provided we used the appropriate finite-difference approximations for the partial derivatives and follow the correct algebraic steps when deriving the scheme.

Moreover, we say that the numerical scheme is *convergent* if the exact solution to the finite-difference equations tends to the exact solution of the PDE problem as $h, k \to 0$. In other words, the scheme is convergent if the *discretization error* tends to zero as $h, k \to 0$.

It is tempted to define stability in terms of the remaining source of error, namely the round-off error, but the machine-dependence (and therefore essentially probabilistic nature) of such errors renders this kind of definition inappropriate. The appropriate definition, due to von Neumann, focuses on boundedness of propagated errors in the exact solution.

For this, let U_{i_0,j_0} be the exact solution to the finite-difference approximation at time t_{j_0} and consider its replacement by $U_{i_0,j_0} + \varepsilon_{i_0,j_0}$, where ε_{i_0,j_0} is an arbitrary error introduced at this time. Let $\widetilde{U}^0_{i,j}$ denote the exact solution to the finite-difference approximation at a subsequent time t_j assuming that no further errors are introduced. We then call $|\widetilde{U}^0_{i,j} - U_{i,j}|$ the *departure* from the exact solution resulting from the error ε_{i_0,j_0}. More generally, the *cumulative departure* is the departure resulting from errors introduced at all times between t_{j_0} and t_j when calculations have to be performed in the numerical scheme. Naturally, this cumulative departure increases when $k \to 0$ (and the number of time steps increases).

Assuming that the maximum absolute error across all times can be bounded by a constant, that is, $|\varepsilon(t, x)| < \delta$ for all (t, x), we say that the numerical scheme is *stable* if the cumulative departure goes to zero as $\delta \to 0$ and is $\mathcal{O}(k^{-1})$ as $k \to 0$. In practice, stability is asserted either by using a Fourier decomposition of the solution $U_{i,j}$ or by using the expression to compute $U_{i,j}$ in matrix form, as it will be illustrated in the examples that follow.

11.4 Methods for the Heat Equation

Consider the following PDE problem:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}, \quad \text{for } 0 \le t \le T, \quad -L \le x \le L \tag{183}$$

$$u(0, x) = f(x) \tag{184}$$

$$u(t, -L) = g_{-L}(t) \tag{185}$$

$$u(t, L) = g_L(t) \tag{186}$$

For the numerical solution of this problem using finite-differences, we consider again the increments $\Delta t = k$ and $\Delta x = h$ and the grid

$$t_j = jk, \qquad j = 0, \dots M, \qquad M = T/k \tag{187}$$

$$x_i = ih, \qquad i = -N, \dots N, \qquad N = L/h. \tag{188}$$

11.4.1 The Explicit Method

Using a forward-difference approximation for the first order derivative with respect to t and a central-difference approximation for the second order derivative with respect to x in equation (183), and dropping all $\mathcal{O}(k)$ and $\mathcal{O}(h^2)$ terms, leads to

$$\frac{u_{i,j+1} - u_{i,j}}{k} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{189}$$

for $i = -N+1, \dots, 0, \dots, N-1$ and $j = 1, \dots M-1$. Observe that we did not include the points corresponding to $j = 0$ and $i = -N$ or N, as these are associated with the boundary conditions. Introducing the parameter

$$\alpha = \frac{k}{h^2} = \frac{\Delta t}{(\Delta x)^2} \tag{190}$$

and rearranging terms, we obtain

$$u_{i,j+1} = \alpha\, u_{i+1,j} + (1 - 2\alpha)\, u_{i,j} + \alpha\, u_{i-1,j} \tag{191}$$

This is called the *explicit* method for the heat equation, because the unknown $u_{i,j+1}$ at time t_{j+1} can be obtained explicitly in terms of quantities previously calculated at time t_j, starting from the boundary condition at $t_0 = 0$. Moreover, we see that the solution $u_{i,j+1}$ at x_j depend only on previously calculated values for the solution at the three points x_{i+1}, x_i, and x_{i-1}. These two observations

mean that, if we represent the solution at time t_{j+1} by a column vector U_{j+1} (with difference row representing different values for x), then not only U_{j+1} can be explicitly calculated from U_j, but each entry in U_{j+1} depends on at most three entries in U_j.

The reason we say at most three is that we need to treat the boundary conditions separately, namely

$$u_{i,0} = f(ih), \qquad i = -N+1, \ldots, 0, \ldots, N-1 \tag{192}$$

$$u_{-N,j} = g_{-L}(j\,k), \qquad j = 0, \ldots, M \tag{193}$$

$$u_{N,j} = g_L(j\,k), \qquad j = 0, \ldots, M \tag{194}$$

If we now define the $(2N-1) \times 1$ column vectors

$$U_j = \begin{bmatrix} u_{N-1,j} \\ u_{N-2,j} \\ \vdots \\ u_{-N+2,j} \\ u_{-N+1,j} \end{bmatrix}, \qquad g_j = \begin{bmatrix} u_{N,j} \\ \\ \vdots \\ \\ u_{-N,j} \end{bmatrix} \tag{195}$$

we can rewrite (191) in matrix form as

$$U_{j+1} = AU_j + \alpha g_j \tag{196}$$

where A is the following $(2N-1) \times (2N-1)$ matrix:

$$A = \begin{bmatrix} 1-2\alpha & \alpha & 0 & 0 & \cdots & 0 \\ \alpha & 1-2\alpha & \alpha & 0 & \cdots & 0 \\ 0 & \alpha & 1-2\alpha & \alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \alpha & 1-2\alpha & \alpha \\ 0 & \cdots & 0 & 0 & \alpha & 1-2\alpha \end{bmatrix} \tag{197}$$

Observe that it follows from (196) that

$$\begin{aligned}
U_1 &= AU_0 + \alpha g_0 \\
U_2 &= AU_1 + \alpha g_1 = A^2U_0 + \alpha\left(Ag_0 + g_1\right) \\
U_3 &= AU_2 + \alpha g_2 = A^3U_0 + \alpha\left(A^2g_0 + Ag_1 + g_2\right) \\
&\vdots = \vdots \\
U_j &= A^jU_0 + \alpha\left(A^{j-1}g_0 + A^{j-2}g_1 + \cdots + g_{j-1}\right)
\end{aligned} \tag{198}$$

Now consider a perturbed initial vector $\widetilde{U}_0 = U_0 + \varepsilon_0$. It follows from (198) that, if no further errors are introduced up to time t_j, then

$$\varepsilon_j := \widetilde{U}_j - U_j = A^j(\widetilde{U}_0 - U_0) = A^j\varepsilon_0. \tag{199}$$

Decomposing the initial error as a linear combination of eigenvectors v_s of A, that is

$$\varepsilon_0 = \sum_{s=-N+1}^{N-1} c_s v_s \tag{200}$$

we find that

$$\varepsilon_j = A^j\varepsilon_0 = \sum_{s=-N+1}^{N-1} c_s\lambda_s^j v_s, \tag{201}$$

where λ_s is the eigenvalue associated with the eigenvector v_s. Therefore, provided that the spectral radius $r(A)$ of the matrix A (namely the largest absolute values of its eigenvalues) is bounded by one, the initial error does not increase exponentially with j. Recalling that the spectral radius is bounded by any norm of A, we can use the ∞-norm (defined as the largest row sum of absolute values) to obtain

$$r(A) \le \|A\|_\infty = 2\alpha + |1 - 2\alpha|. \tag{202}$$

That is, provided $\alpha \le 1/2$ we have that $r(A) \le 1$ and the explicit method is stable. It can also be shown that this is a sufficient condition. In other words, we conclude that the explicit method for the heat equation is *conditionally stable*, with the condition for stability being that

$$\Delta t \le \frac{1}{2}(\Delta x)^2. \tag{203}$$

Example 11.1. *Consider the heat equation on the space domain* $0 \le x \le 1$

with boundary conditions

$$u(0,x) = f(x) = \begin{cases} 2x, & 0 \le x \le \frac{1}{2} \\ 2(1-x), & \frac{1}{2} \le x \le 1 \end{cases} \tag{204}$$

$$u(t,0) = u(t,1) = 0 \tag{205}$$

The solution gradually converges to zero as $t \to \infty$, but exhibits instability if we use $\alpha > 1/2$.

11.4.2 The Implicit Method

Consider now using a backward-difference approximation for the first order derivative with respect to t and the same central-difference approximation for the second order derivative with respect to x in equation (183). Dropping all $\mathcal{O}(k)$ and $\mathcal{O}(h^2)$ terms leads to

$$\frac{u_{i,j} - u_{i,j-1}}{k} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{206}$$

for $i = -N+1, \ldots, 0, \ldots, N-1$ and $j = 1, \ldots M$. Replacing $j-1, j$ by $j, j+1$ and rearranging terms we obtain

$$-\alpha\, u_{i-1,j+1} + (1+2\alpha)\, u_{i,j+1} - \alpha\, u_{i+1,j+1} = u_{i,j} \tag{207}$$

This is called the *implicit* method for the heat equation, because the three unknowns $u_{i-1,j+1}$, $u_{i,j+1}$, and $u_{i+1,j+1}$ at time t_{j+1} are given implicitly as the solution of (207) (i.e they cannot be found separately in terms of quantities previously calculated at time t_j). In matrix notation, this can be rewritten as

$$BU_{j+1} = U_j + \alpha g_j, \tag{208}$$

where U_j and g_j are the same $(2N-1)\times 1$ column vectors defined in (195) and B is the following $(2N-1)\times(2N-1)$ matrix:

$$B = \begin{bmatrix} 1+2\alpha & -\alpha & 0 & 0 & \cdots & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & \cdots & 0 \\ 0 & -\alpha & 1+2\alpha & -\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -\alpha & 1+2\alpha & -\alpha \\ 0 & \cdots & 0 & 0 & -\alpha & 1+2\alpha \end{bmatrix}. \tag{209}$$

In other words, the vector U_{j+1} needs to be found as the solution to the system of linear equations shown in (208).

It can be shown that, provided $\alpha \geq 0$, the matrix B is invertible, so that (208) is equivalent to

$$U_{j+1} = B^{-1}U_j + \alpha B^{-1}g_j \tag{210}$$

and stability can in principle be determined from the spectral radius of the matrix B^{-1}. In practice, finding the inverse of a large matrix is a computationally expensive problem, and one should solve large linear systems through other methods, such as the LU factorization. Specifically, we proceed by finding a lower triangular matrix L and an upper triangular[10] matrix V such that $B = LV$. Substituting this in (208) leads to

$$(LV)U_{j+1} = U_j + \alpha g_j, \tag{211}$$

which can be solved by forward and backward substitution for the triangular linear systems

$$Lx_{j+1} = U_j + \alpha g_j, \tag{212}$$

$$VU_{j+1} = x_{j+1} \tag{213}$$

Because of the specific format of the matrix B, it is easy to see that the only non-zero entries of the triangular matrices L and V in the decomposition $B = LV$

[10] We use V to denote the upper triangular matrix in an LU factorization, instead of the more common notation U found in the numerical analysis literature, to avoid confusion with the column vectors U_j

are given by

$$\begin{aligned} V_{11} &= 1 + 2\alpha \\ V_{ii} &= 1 + 2\alpha - \frac{\alpha^2}{V_{i-1,i-1}}, \qquad i = 2, \ldots, 2N-1 \\ V_{i,i+1} &= -\alpha \\ L_{ii} &= 1, \qquad\qquad\qquad\qquad i = 1, \ldots, 2N-1 \\ L_{i+1,i} &= \frac{-\alpha}{V_{ii}}, \qquad\qquad\qquad i = 1, \ldots, 2N-2 \end{aligned}$$

To avoid having to deal with the inverse matrix B^{-1}, we also present an alternative way to study the stability of the implicit method based on Fourier analysis. Specifically, let us write the initial error at the point x in the form of a finite Fourier series, that is

$$\varepsilon(0,x) = \sum_{n=-N}^{N} c_n e^{\frac{in\pi}{2L}x}. \tag{214}$$

It follows from linearity that we can concentrate on each component of the expansion above separately and then superimposed their contributions to the total error, that is, we can concentrate on a single error term of the form

$$\varepsilon_n(0,x) = e^{\frac{in\pi}{2L}x}. \tag{215}$$

Assuming that the effect of the numerical scheme on this error term can be written as

$$\varepsilon_n(t,x) = e^{\lambda t} e^{\frac{in\pi}{2L}x}, \tag{216}$$

that is, the original error times a growth factor $e^{\lambda t}$, we can substitute[11]

$$\varepsilon_{m,j} = \varepsilon_n(t_j, x_m) = e^{\lambda jk} e^{\frac{in\pi m}{2N}}, \tag{217}$$

where we used $t_j = jk$ and $x_m = mh = m\frac{L}{N}$, into the scheme (207) and obtain

$$e^{\frac{in\pi m}{N}} e^{\lambda jk} = -\alpha e^{\frac{in\pi(m-1)}{N}} e^{\lambda(j+1)k} \tag{218}$$

$$+ (1+2\alpha) e^{\frac{in\pi m}{N}} e^{\lambda(j+1)k} - \alpha e^{\frac{in\pi(m+1)}{N}} e^{\lambda(j+1)k}. \tag{219}$$

[11] We use m to denote the grid points for the x variable in this formula, instead of i as in the remainder of the chapter, to avoid confusion with the imaginary number i appearing in the exponents of each term.

After cancellations, this gives

$$e^{-\lambda k} = -\alpha e^{\frac{-in\pi}{N}} + (1+2\alpha) - \alpha e^{\frac{in\pi}{N}} \tag{220}$$

that is, after rearranging the exponential term,

$$e^{\lambda k} = \frac{1}{1 + 4\alpha \sin^2\left(\frac{n\pi}{2N}\right)} \leq 1, \qquad \text{for all } \alpha \geq 0. \tag{221}$$

This shows that the growth factor is always less than one and implicit method is *unconditionally* stable.

11.5 The Crank-Nicolson Method

Taking the average of (191) and (207) and rearranging terms leads to

$$-\alpha\, u_{i-1,j+1} + 2(1+\alpha)\, u_{i,j+1} - \alpha\, u_{i+1,j+1} = \alpha\, u_{i-1,j} + 2(1-\alpha)\, u_{i,j} + \alpha\, u_{i+1,j}, \tag{222}$$

which can be rewritten in matrix form as

$$CU_{j+1} = DU_j + \alpha\,(g_{j+1} + g_j) \tag{223}$$

where U_j and g_j are the same $(2N-1)\times 1$ column vectors defined in (195) and C and D are the following $(2N-1)\times(2N-1)$ matrices:

$$C = \begin{bmatrix} 2(1+\alpha) & -\alpha & 0 & 0 & \cdots & 0 \\ -\alpha & 2(1+\alpha) & -\alpha & 0 & \cdots & 0 \\ 0 & -\alpha & 2(1+\alpha) & -\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -\alpha & 2(1+\alpha) & -\alpha \\ 0 & \cdots & 0 & 0 & -\alpha & 1+2\alpha \end{bmatrix} \tag{224}$$

$$D = \begin{bmatrix} 2(1-\alpha) & \alpha & 0 & 0 & \cdots & 0 \\ \alpha & 2(1-\alpha) & \alpha & 0 & \cdots & 0 \\ 0 & \alpha & 2(1-\alpha) & \alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \alpha & 2(1-\alpha) & \alpha \\ 0 & \cdots & 0 & 0 & \alpha & 1-2\alpha \end{bmatrix}. \tag{225}$$

Using either the matrix method based on the spectral radius of $C^{-1}D$ or the Fourier method explained in the previous section, it is straightforward to established that the Crank-Nicolson scheme for the heat equation is also *unconditionally* stable.

Remark 11.2. This scheme is equivalent to taking a central-difference approximation at the point $(i, j+1/2)$ for the time derivative, resulting in a truncation error of order $\mathcal{O}(h^2 + k^2)$, as opposed to $\mathcal{O}(h^2 + k)$ for the explicit and implicit methods, which use forward and backward-differences respectively. Based on this observation, one could be tempted to consider the scheme

$$\frac{u_{i,j+1} - u_{i,j-1}}{2k} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \tag{226}$$

which uses a central-difference approximation at the point (i, j) and is also of order $\mathcal{O}(h^2 + k^2)$. This is called the Richardson scheme, which when applied to the heat equation happens to be *unstable* for all values of $\alpha \geq 0$.

11.6 From Black-Scholes to the Heat Equation

Consider the Black-Scholes PDE problem

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf = 0 \tag{227}$$

$$f(T, S) = \Phi(S) \tag{228}$$

for the price $f(t, S)$ of a European derivative with payoff $\Phi(S)$ at maturity T over the domain $(t, S) \in [0, T] \times [0, \infty)$. Applying finite-differences methods directly to this problem gives rise to numerical scheme with non-constant coefficients, because of the terms rS and $\sigma^2 S^2$ above. Whereas it is possible to implement such schemes, the presence of non-constant coefficients can lead to delicate issues associated with boundary conditions and stability. In what follows, we present a canonical change of variables that reduces the Black-Scholes equation to the heat equation, for which we can use the methods introduced in the previous sections.

Start with the new variables

$$\tau = \frac{1}{2}\sigma^2(T - t) \tag{229}$$

$$x = \log \frac{S}{K}, \tag{230}$$

where K is an appropriate scaling factor for the stock price, most often the strike price of a put, call, or other exotic options. It then follows from the chain rule that

$$\frac{\partial f}{\partial t} = \frac{\partial f}{\partial \tau}\frac{\partial \tau}{\partial t} = -\frac{1}{2}\sigma^2 \frac{\partial f}{\partial \tau} \tag{231}$$

$$\frac{\partial f}{\partial S} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial S} = \frac{1}{S}\frac{\partial f}{\partial x} \tag{232}$$

$$\frac{\partial^2 f}{\partial S^2} = \frac{\partial}{\partial S}\left(\frac{1}{S}\frac{\partial f}{\partial x}\right) = -\frac{1}{S^2}\frac{\partial f}{\partial x} + \frac{1}{S^2}\frac{\partial^2 f}{\partial x^2} = \frac{1}{S^2}\left(\frac{\partial^2 f}{\partial x^2} - \frac{\partial f}{\partial x}\right) \tag{233}$$

Substituting these expression into (227) leads to

$$-\frac{1}{2}\sigma^2\frac{\partial f}{\partial \tau} + r\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2\left(\frac{\partial^2 f}{\partial x^2} - \frac{\partial f}{\partial x}\right) - rf = 0 \tag{234}$$

Introducing the dimensionless parameter

$$\lambda = \frac{2r}{\sigma^2} \tag{235}$$

leads to the slightly revised equation

$$\frac{\partial f}{\partial \tau} = \frac{\partial^2 f}{\partial x^2} + (\lambda - 1)\frac{\partial f}{\partial x} - \lambda f = 0. \tag{236}$$

Defining a new function $u(\tau, x)$ by

$$f(\tau, x) = Ke^{\frac{(1-\lambda)}{2}x - \frac{(1+\lambda)^2}{4}\tau}u(\tau, x) \tag{237}$$

leads to

$$\frac{\partial f}{\partial \tau} = -\frac{(1+\lambda)^2}{4}f + \frac{f}{u}\frac{\partial u}{\partial \tau} \tag{238}$$

$$\frac{\partial f}{\partial x} = \frac{(1-\lambda)}{2}f + \frac{f}{u}\frac{\partial u}{\partial x} \tag{239}$$

$$\frac{\partial^2 f}{\partial x^2} = \frac{(1-\lambda)}{2}\frac{\partial f}{\partial x} + \frac{(1-\lambda)}{2}\frac{f}{u}\frac{\partial u}{\partial x} + \frac{f}{u}\frac{\partial^2 u}{\partial x^2} \tag{240}$$

Substituting these expressions into (236), and carefully doing all the necessary cancellations, finally leads to

$$\frac{\partial u}{\partial \tau} = \frac{\partial^2 u}{\partial x^2}, \tag{241}$$

which we recognize as the heat equation for u. It follows from the change of

variables (229)-(230) that this equation needs to be solved over the domain $(\tau, x) \in [0, \sigma^2 T/2] \times [-L, L]$ where $L = \log S_{\max}/K$ for a sufficiently large stock price $S_{\max}$. Notice that this results in the smallest value for the stock price computed on this grid being $S_{\min} = Ke^{-L} \approx 0$. This leads to the following boundary conditions for the function u:

$$u(0, x) = \frac{1}{K} e^{\frac{(\lambda-1)}{2} x} \Phi(e^x) \tag{242}$$

$$u(\tau, L) = \frac{1}{K} e^{\frac{(\lambda-1)}{2} L + \frac{(1+\lambda)^2}{4} \tau} f(\tau, L) \tag{243}$$

$$u(\tau, -L) = \frac{1}{K} e^{\frac{-(\lambda-1)}{2} L + \frac{(1+\lambda)^2}{4} \tau} f(\tau, -L) \tag{244}$$

Notice that the expression above should take into account the behaviour of $f(t, S)$ for $S \to 0$ and $S \to \infty$, which in turn depend on the derivative being evaluated.

11.7 Examples of derivative pricing

11.7.1 Explicit method for European put option

For an European put option with strike price K, the boundary conditions are

$$f(T, S) = (K - S)^+ \tag{245}$$

$$\lim_{S \to \infty} f(t, S) = 0 \tag{246}$$

$$f(t, 0) = Ke^{-r(T-t)} \tag{247}$$

The first condition above is simply the payoff of the put option at maturity. The second condition follows from the fact that a put option becomes worthless for very high values of the stock price, as it is unlikely that it will be exercised at maturity. The third condition arises because, for a geometric Brownian motion (or any other process with multiplicative increments), once the stock price hits zero it remains there and consequently the put option becomes a contract that is guaranteed to pay an amount K at maturity, so that its value at time t is just the present value of this amount. Using the transformation (237), these

boundary conditions become

$$u(0,x) = e^{\frac{\lambda-1}{2}x}\left(1-e^{x}\right)^{+} = \left(e^{\frac{(\lambda-1)}{2}x} - e^{\frac{(\lambda+1)}{2}x}\right)^{+} \tag{248}$$

$$u(\tau, L) = 0 \tag{249}$$

$$u(\tau,-L) = e^{-\lambda\tau}e^{\frac{-(\lambda-1)}{2}L+\frac{(1+\lambda)^2}{4}\tau} = e^{\frac{-(\lambda-1)}{2}L+\frac{(1-\lambda)^2}{4}\tau}. \tag{250}$$

For the discretization we first choose a suitably small time interval Δt and then set

$$k = \Delta\tau = \frac{\sigma^2}{2}\Delta t \tag{251}$$

$$h = \Delta x = \left(\frac{k}{\alpha}\right)^{1/2} \tag{252}$$

for some $\alpha < 1/2$ so that the stability condition for the explicit method is satisfied. Next choose a suitably large value for $S_{\max}$ and set the number of grid points as

$$M = \left[\frac{\sigma^2 T}{2k}\right] \tag{253}$$

$$N = \left[\frac{\log\left(S_{\max}/K\right)}{h}\right], \tag{254}$$

where $[\cdot]$ denotes the integer part of a number. In other words, we consider grid points of the form

$$\tau_j = jk, \qquad j = 0,\ldots,M \tag{255}$$

$$x_i = ih, \qquad i = -N,\ldots,N \tag{256}$$

The solution u_{ij} to the heat equation on these grid points can then be found using the method described in Section 11.4.1, which in turn leads to the price of the European put option through (237).

11.7.2 Implicit method for European call option

The boundary conditions for a call option are

$$f(T,S) = (S-K)^+ \tag{257}$$

$$f(t,S) = S - Ke^{-r(T-t)} \qquad \text{as} \quad S \gg K \tag{258}$$

$$f(t,0) = 0 \tag{259}$$

The first condition is simply the payoff of the option at maturity. The second condition follows from the fact that for very large stock prices the call option becomes a contract that is guaranteed to pay $(S_T - K)$ at maturity, so that its value at time t is the present value of this amount. The last condition reflects the fact that a call option becomes worthless if the stock price ever reaches zero.

Using the same change of variables as in the previous section we obtain the following boundary conditions for the function u:

$$u(0,x) = \left(e^{\frac{(\lambda-1)}{2}x} - e^{\frac{(\lambda+1)}{2}x}\right)^+ \tag{260}$$

$$u(\tau,L) = e^{\frac{(\lambda+1)}{2}L + \frac{(1+\lambda)^2}{4}\tau} - e^{\frac{(\lambda-1)}{2}L + \frac{(1-\lambda)^2}{4}\tau} \tag{261}$$

$$u(\tau,-L) = 0 \tag{262}$$

We can now use the same discretization as in the previous section, but with no restriction on the constant $\alpha \geq 0$, and the implicit method described in Section 11.4.2 to find the solution $u_{i,j}$ for these boundary conditions. The price of the call option then follows from (237).

11.7.3 Barrier option by Crank-Nicolson

Consider a down-and-out call option, that is, a call option that becomes worthless if the stock price reaches a barrier $S_b < S_0$ at any time before maturity. We can use the same change of variables and discretization as in the previous section, except that the domain is further restricted to $x \in [x_b, L]$, where $L = \log(S_{\max}/K)$ as before and $x_b = \log(S_b/K)$.

This leads to the following modified boundary conditions for the function u:

$$u(0,x) = \left(e^{\frac{(\lambda-1)}{2}x} - e^{\frac{(\lambda+1)}{2}x}\right)^+ \tag{263}$$

$$u(\tau,L) = e^{\frac{(\lambda+1)}{2}L+\frac{(1+\lambda)^2}{4}\tau} - e^{\frac{(\lambda-1)}{2}L+\frac{(1-\lambda)^2}{4}\tau} \tag{264}$$

$$u(\tau,x_b) = 0 \tag{265}$$

We can then use the method described in Section 11.5 to find the solution $u_{i,j}$ for the heat equation with these boundary conditions and use (237) to recover the price of the down-and-out call option.

11.8 American Options

Consider again the American put option discussed in Section 9.5. Denoting its price by $f(t,S)$ as before, the possibility of early exercise implies that

$$f(t,S) \geq \max(K-S,0) \geq 0 \tag{266}$$

must be satisfied for all times and stock values. A less trivial consequence of the possibility of early exercise is that, instead of satisfying the Black-Scholes equation in (227), the price of an American option satisfy the inequality

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf \leq 0, \tag{267}$$

since it is not possible to hold an exact replicating portfolio earning the same interest rate as the risk-free account. This is because any long position on the option can be exercised against, so that the best one can do is to construct a portfolio with return at most as large as the risk-free rate.

Because the payoff for a put option is monotonically decreasing in S, it is clear that, at each time t, the decision to exercise it will depend on a single value $S^*(t)$ as follows:

- If $0 \leq S < S^*(t)$, then it is optimal to exercise the option and $f(t,S) = K - S$, in which case

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf = -rK < 0. \tag{268}$$

- If, on the other hand, $S > S^*(t)$, then early exercise is not optimal, $f(t,S) > K - S$, and f satisfies the Black-Scholes equation.

In order the *free boundary* $S^*(t)$, we impose the so-called "smooth pasting conditions"

$$f(t, S^*(t)) = K - S^*(t) \tag{269}$$

$$\frac{\partial f}{\partial S}(t, S^*(t)) = -1, \tag{270}$$

that is, continuity of the function f with respect to S, which follows from the absence of arbitrage, and continuity of the slope of f with respect to S, which is a little harder to justify but follows from standard arguments in optimal control.

It can be shown, using techniques from variational analysis, that this problem has a unique solution, which is nevertheless not given by a closed-form expression (in contrast with, for example, the price of the European put option, which is given by the Black-Scholes formula). For numerical solutions, it is convenient to replace it with an equivalent problem where there is no explicit use of the free boundary $S^*(t)$. This is achieved by the idea of *linear complementarity* as follows.

Observe first that (266) and (267) hold regardless on which side of the free boundary the value of the stock happens to be. Moreover, it is always the case that either (i) $f(t, S) = K - S$, in which case (267) holds as a strict inequality, or (ii) an equality holds in (267), in which case $f(t, S) > K - S$. That is to say, we have that the following conditions always hold:

$$f(t, S) - (K - S) \geq 0 \tag{271}$$

$$\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf \leq 0 \tag{272}$$

$$\left(\frac{\partial f}{\partial t} + rS\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 f}{\partial S^2} - rf\right)\big(f(t, S) - K + S\big) = 0 \tag{273}$$

Once this problem is solved, the free boundary $S^*(t)$ can be obtained by a continuity argument, namely as the largest stock value satisfying $f(t, S) = K - S$ for each fixed t.

11.8.1 Linear complementarity and the heat equation

Consider again the change of variables

$$\tau = \frac{1}{2}\sigma^2(T-t) \tag{274}$$

$$x = \log\left(\frac{S}{K}\right) \tag{275}$$

$$f(\tau, x) = Ke^{\frac{(1-\lambda)}{2}x - \frac{(1+\lambda)^2}{4}\tau}u(\tau, x). \tag{276}$$

We can then see that the inequality $f(t,S) > K - S$ can be rewritten as

$$u(\tau, x) > e^{\frac{(\lambda-1)}{2}x + \frac{(1+\lambda)^2}{4}\tau}\left(1 - e^x\right) = e^{\frac{(1+\lambda)^2}{4}\tau}\left(e^{\frac{(\lambda-1)}{2}x} - e^{\frac{(\lambda+1)}{2}x}\right), \tag{277}$$

whereas, as we have shown, the Black-Scholes equation for $f(t,S)$ becomes the heat equation for $u(\tau, x)$. It therefore follows that, if we define

$$q(\tau, x) = e^{\frac{(1+\lambda)^2}{4}\tau}\left(e^{\frac{(\lambda-1)}{2}x} - e^{\frac{(\lambda+1)}{2}x}\right), \tag{278}$$

we can rewrite (271)-(273) as

$$u(\tau, x) - q(\tau, x) \geq 0 \tag{279}$$

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} \geq 0, \tag{280}$$

$$\left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)\left(u(\tau, x) - q(\tau, x)\right) = 0 \tag{281}$$

and the requirement that u and $\frac{\partial u}{\partial x}$ be continuous.

Choosing the Crank-Nicolson scheme, the three equations in (279)-(281) can be approximated by

$$\left[-\alpha u_{i-1,j+1} + 2(1+\alpha)u_{i,j+1} - \alpha u_{i,j+1} - \left(\alpha u_{i-1,j} + 2(1-\alpha)u_{i,j} + \alpha u_{i+1,j}\right)\right] \times \left[u_{i,j+1} - q_{i,j+1}\right] = 0 \tag{282}$$

$$-\alpha u_{i-1,j+1} + 2(1+\alpha)u_{i,j+1} - \alpha u_{i,j+1} \geq \alpha u_{i-1,j} + 2(1-\alpha)u_{i,j} + \alpha u_{i+1,j} \tag{283}$$

$$u_{i,j+1} \geq q_{i,j+1} \tag{284}$$

or in matrix notation

$$(CU_{j+1} - b_j) * (U_{j+1} - Q_{j+1}) = 0 \tag{285}$$

$$CU_{j+1} \geq b_j \tag{286}$$

$$U_{j+1} \geq Q_{j+1} \tag{287}$$

where $b_j = DU_j + \alpha(g_{j+1} + g_j)$ and

$$C = \begin{bmatrix} 2(1+\alpha) & -\alpha & 0 & 0 & \cdots & 0 \\ -\alpha & 2(1+\alpha) & -\alpha & 0 & \cdots & 0 \\ 0 & -\alpha & 2(1+\alpha) & -\alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & -\alpha & 2(1+\alpha) & -\alpha \\ 0 & \cdots & 0 & 0 & -\alpha & 1+2\alpha \end{bmatrix} \tag{288}$$

$$D = \begin{bmatrix} 2(1-\alpha) & \alpha & 0 & 0 & \cdots & 0 \\ \alpha & 2(1-\alpha) & \alpha & 0 & \cdots & 0 \\ 0 & \alpha & 2(1-\alpha) & \alpha & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \alpha & 2(1-\alpha) & \alpha \\ 0 & \cdots & 0 & 0 & \alpha & 1-2\alpha \end{bmatrix} \tag{289}$$

$$U_j = \begin{bmatrix} u_{N-1,j} \\ u_{N-2,j} \\ \vdots \\ u_{-N+2,j} \\ u_{-N+1,j} \end{bmatrix}, \quad Q_j = \begin{bmatrix} q_{N-1,j} \\ q_{N-2,j} \\ \vdots \\ q_{-N+2,j} \\ q_{-N+1,j} \end{bmatrix}, \quad g_j = \begin{bmatrix} u_{N,j} \\ \\ \vdots \\ \\ u_{-N,j} \end{bmatrix}. \tag{290}$$

Note that in (286) and (287), an inequality $a \geq b$ between two vectors a and b means that $a_i \geq b_i$ for all i. With this in mind, one might be tempted to approach this problem by first solving the system $CU_{j+1} = b_j$ and then ensuring that $U_{j+1} \geq Q_{j+1}$ by setting

$$U_{i,j+1} = \max(U_{i,j+1}, Q_{i,j+1}).$$

However, because the components of the vector U_{j+1} depend implicitly on each other, if we do this we would no longer be sure that $CU_{j+1} \geq b_j$ and $(CU_{j+1} - b_j)(U_{j+1} - Q_{j+1}) = 0$. Having said that, notice that this procedure would work

for explicit methods, and is in fact used in practice. Because of the instability that is inherent to these methods, however, we adopt an alternative approach, namely solving the system $CU_{j+1} = b_j$ by what is called a *projective iterative method*. For this, we first need to take a small detour through iterative schemes for solving linear systems.

11.8.2 Iterative schemes for solving $Ax = b$

The idea behind iterative schemes is to split the matrix A into $A = M - N$ and convert the problem $Ax = b$ into the fixed point problem

$$x = F(x) = M^{-1}(Nx + b), \tag{291}$$

that is, perform iterations of the form

$$Mx^{(k+1)} = Nx^{(k)} + b \tag{292}$$

until the difference $\|x^{(k+1)} - x^{(k)}\|$ falls below a pre-specified tolerance.

The most popular of such methods is the *Jacobi method*, obtained by setting [12] $A = \Delta + L + V$, where Δ the diagonal, L is the lower-triangular and V is the upper-triangular parts of A. Taking $M = \Delta$ and $N = -(L + V)$ gives

$$x^{(k+1)} = \Delta^{-1}(b - (L + V)x^{(k)}) \tag{293}$$

or equivalently

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j \neq i} A_{ij} x_j^{(k)} \right). \tag{294}$$

A simple modification of this procedure leads to the *Gauss-Seidel method*, namely setting $M = \Delta + L$ and $N = -V$, that is

$$x^{(k+1)} = \Delta^{-1}(b - Lx^{(k+1} - Vx^{(k)}) \tag{295}$$

or equivalently

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} A_{ij} x_j^{(k)} \right). \tag{296}$$

[12] We use Δ and V instead of the more common symbols D and U for a diagonal and upper triangular matrices to avoid confusion with the other uses of these symbols in this section.

As a further modification, let us define

$$r^{(k)} = x^{(k+1)} - x^{(k)} \tag{297}$$

and interpret it as the *correction* that must be added to $x^{(k)}$ in order to obtain the next iteration $x^{(k+1)}$. The *successive over-relaxation method* (SOR) is based on trying to overcorrect $x^{(k)}$, that is, write

$$x^{(k+1)} = x^{(k)} + \omega r^{(k)} \tag{298}$$

for some parameter $1 < \omega < 2$. It can be shown that over-relaxation improves the convergence of the Gauss-Seidel method, whereas under-relaxation (namely $0 < \omega < 1$) makes the method slower, and $\omega > 2$ leads to divergence.

Concretely, we can implement SOR for the Gauss-Seidel method as

$$x^{(k+1)} = x^{(k)} + \omega \left(\Delta^{-1}(b - Lx^{(k+1} - Vx^{(k)} - \Delta x^{(k)} \right), \tag{299}$$

or equivalently

$$x_i^{(k+1)} = x_i^{(k)} + \frac{\omega}{A_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} - \sum_{j=i}^{n} A_{ij} x_j^{(k)} \right). \tag{300}$$

11.8.3 Projected SOR method for American Options

Returning to our linear complementarity problem, we can now use the following modified version of the SOR method:

$$U_{i,j+1}^{(k+1)} = \max \left[U_{i,j+1}^{(k)} + \frac{\omega}{C_{ii}} \left(b_i - \sum_{p=1}^{i-1} C_{ip} U_{p,j+1}^{(k+1)} - \sum_{p=i}^{2N-1} C_{ip} U_{p,j+1}^{(k)} \right), Q_{i,j+1} \right]. \tag{301}$$

We then iterate this until $\|U_{j+1}^{(k+1)} - U_{j+1}^{(k)}\|$ becomes smaller than some pre-specified tolerance and then declare that $U_{j+1} = U^{(k+1)}$. By construction, we have that either

$$U_{i,j+1} = Q_{i,j+1}$$

or the i-th component of $(CU_{j+1} - b_j)$ vanishes. Therefore both (285) and (287) are satisfied, whereas (286) follows from positive definiteness of the matrix C.

A Statistical Classifiers

As we have seen, the logistic regression discussed in Section 5.3 can be used for the problem of binary classification, whereby we interpret $f_{\mathbf{w},b}(\mathbf{x})$ as the probability that a data point with features $\mathbf{x}$ belongs to the class labelled by $y = 1$. In the language of statistics, this corresponds to $P(y = 1|\mathbf{x})$, that is to say, the conditional probability of observing the output $y = 1$ given that the input is $\mathbf{x}$. For a more general classification problem, we are interested in estimating $P(y = k|\mathbf{x})$, where $k = 1, \ldots, q$ correspond to one of q possible classes.

In this context, a *statistical classifier* is a rule for assigning a data point to a class based on estimates for $P(y = k|\mathbf{x})$. An example of such rule is the *Bayes classifier*, which consists of assigning a data point with input $\mathbf{x}_j$ to the class k for which the estimate for $P(y_j = k|\mathbf{x}_j)$ is largest. Observe that this exactly the rule that we encountered at the end of Section 5.3.4 for logistic regression with multiple classes using the *one-versus-all* algorithm. For binary classification using logistic regression, the Bayes classifier consists of using $p_0 = 0.5$ in (42), namely assigning the data point to the class $k = 1$ if $f_{\widehat{\mathbf{w}},\widehat{b}}(\mathbf{x}_j) \geq 0.5$.

Having estimated a statistical classifier using a training dataset, we define the *training error rate* as the proportion of misclassified training data points, that is

$$\hat{\mathcal{R}}_{\mathcal{D}} = \frac{1}{n} \sum_{j=1}^{n} \mathbf{1}_{\{y_j \neq \hat{y}_j\}}, \tag{302}$$

where $\hat{y}_j$ is the predicted class label for the j-th data point. This plays the role of the training cost function discussed in Section 2.1.4 in the context of regression problems. Similarly, we can use a validation dataset to define the *test error rate* as

$$J^{val} = \frac{1}{n_{test}} \sum_{j=1}^{n_{test}} \mathbf{1}_{\{y_j^{val} \neq \hat{y}_j^{val}\}}. \tag{303}$$

It can be shown that the Bayes classifier is the statistical classifier with the lowest possible test error rate, called the *Bayes error rate* and given by

$$1 - E\left(\max_k P(y = k|\mathbf{x})\right). \tag{304}$$

An alternative to using logistic regression or neural networks to estimate the

	Long	Sweet	Yellow	Total
Banana	400	350	450	500
Orange	0	150	300	300
Other	100	150	50	200
Total	500	650	800	1000

Table 2: Dataset for naive Bayes classifier

conditional probability $P(y = k|\mathbf{x})$ consists in applying *Bayes' theorem*, that is,

$$P(y = k|\mathbf{x}) = \frac{P(\mathbf{x}|y = k)P(y = k)}{P(\mathbf{x})} = \frac{P(\mathbf{x}|y = k)P(y = k)}{\sum_{l=1}^{q} P(\mathbf{x}|y = l)P(y = l)}. \tag{305}$$

In this formulation, instead of estimating the *posterior* probabilities $P(y = k|\mathbf{x})$ directly, we focus instead on estimating the *prior* probabilities $P(y = k)$ and the *likelihoods* $P(\mathbf{x}|y = k)$ using a training dataset. Observe that the term $P(\mathbf{x})$ in the denominator of (305) does not depend on k, and therefore does not need to be calculated explicitly in algorithms that attempt to maximize $P(y = k|\mathbf{x})$ over the different classes. Observe also that, for the case of continuous random variables $\mathbf{x}$, the likelihoods appearing in (305) are replaced by the conditional densities $f(\mathbf{x}|y = k)$.

A.1 Naive Bayes Classifier

The simplest classifier using (305) consists of assuming that the features in the input vector $\mathbf{x}$ are independent, that is

$$P(\mathbf{x}|y = k) = P(x_1|y = k)P(x_2|y = k) \cdots P(x_p|y = k) \tag{306}$$

$$P(\mathbf{x}) = P(x_1)P(x_2) \cdots P(x_p) \tag{307}$$

We illustrate such *naive Bayes classifier* in the context of *categorical* features first, where we can make as few assumptions as possible about the priors and the densities. Consider the popular example with the training dataset given be Table 2. In this example, the classes are {Banana, Orange, Other}, which we

can label as $k = 1, 2, 3$ and the features are

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \text{length} \\ \text{taste} \\ \text{colour} \end{bmatrix},$$

each with two possible values (for example, $x_1 = 1$ if the fruit is "Long" or $x_1 = 0$ if the fruit is "Not Long"). By inspecting the table, we can readily calculate the following empirical estimates

$$\begin{aligned}
P(y=1) &= \frac{500}{1000} = 0.5, \qquad P(y=2) = \frac{300}{1000} = 0.3, \qquad P(y=3) = \frac{200}{1000} = 0.2 \\
P(\text{Long}|y=1) &= \frac{400}{500} = 0.8, \qquad P(\text{Long}|y=2) = \frac{0}{300} = 0, \qquad P(\text{Long}|y=3) = \frac{100}{200} = 0.5 \\
P(\text{Sweet}|y=1) &= \frac{350}{500} = 0.7, \qquad P(\text{Sweet}|y=2) = \frac{150}{300} = 0.5, \qquad P(\text{Sweet}|y=3) = \frac{150}{200} = 0.75 \\
P(\text{Yellow}|y=1) &= \frac{450}{500} = 0.9, \qquad P(\text{Yellow}|y=2) = \frac{300}{300} = 1, \qquad P(\text{Yellow}|y=3) = \frac{50}{200} = 0.25 \\
P(\text{Long}) &= \frac{500}{1000} = 0.5, \qquad P(\text{Sweet}) = \frac{650}{1000} = 0.65, \qquad P(\text{Yellow}) = \frac{800}{1000} = 0.8
\end{aligned}$$

Suppose now that we are given a test fruit that is long, sweet, and yellow, that is, with a feature vector $\mathbf{x} = (1, 1, 1)$. Using the estimates above, the naive Bayes classifier produces

$$\begin{aligned}
P(y=1|(\text{Long}, \text{Sweet}, \text{Yellow})) &= \frac{0.8 \times 0.7 \times 0.9 \times 0.5}{0.5 \times 0.65 \times 0.8} = 0.9692 \\
P(y=2|(\text{Long}, \text{Sweet}, \text{Yellow})) &= \frac{0 \times 0.5 \times 1 \times 0.3}{0.5 \times 0.65 \times 0.8} = 0 \\
P(y=3|(\text{Long}, \text{Sweet}, \text{Yellow})) &= \frac{0.5 \times 0.75 \times 0.5 \times 0.2}{0.5 \times 0.65 \times 0.8} = 0.1442,
\end{aligned}$$

so that we assign a fruit with this features to the class $y = 1$, that is, a Banana.

It is easy to see how these calculations generalize to the case of where each of the features is a discrete random variable. For example, suppose that in the example above the feature x_2 could take one of five preassigned values, say $x_2 = 1, \ldots, 5$, corresponding to different levels of sweetness (for example, with $x_2 = 5$ being the sweetest). In this case, each entry in the second column of Table 2 would be replaced by a 5-tuple representing the number of fruits with a given value of sweetness in each class. For example, the values for bananas could be $(20, 40, 90, 150, 200)$, showing that of the 500 bananas in the dataset, 20 had sweetness equal to 1, 40 had sweetness equal to 2, and so on. Observe

that this is just a finer discretization of the feature x_2, which in the previous example had the value "Sweet" for 350 out of 500 bananas. The likelihood for each value of feature x_2 can then be calculated using the empirical 5-tuples for each class. For example, for bananas we obtain

$$\begin{aligned} P(x_2 = 1|y = 1) &= \frac{20}{500} = 0.04 \\ P(x_2 = 2|y = 1) &= \frac{40}{500} = 0.08 \\ &\vdots \\ P(x_2 = 5|y = 1) &= \frac{200}{500} = 0.4 \end{aligned}$$

More generally, we could make other assumptions about the distribution of the discrete random variable representing each feature and estimate it using the empirical values provided in the training dataset.

The same idea applies when the features are represented by continuous random variables. A common assumption for the naive Bayes classifier in this case consists in supposing that, for each class k, the feature vector $\mathbf{x}$ has a multivariate Gaussian distribution $N(\mu_k, \mathbf{\Sigma}_k)$. In symbols, we assume that the conditional densities are of the form

$$f(\mathbf{x}|y = k) = f_1(x_1|y = k) \cdots f_p(x_p|y = k) \tag{308}$$

$$= \frac{1}{\sqrt{2\pi}\sigma_{1k}} \exp\left(-\frac{1}{2}\frac{(x_1 - \mu_{1k})^2}{\sigma_{1k}^2}\right) \cdots \frac{1}{\sqrt{2\pi}\sigma_{pk}} \exp\left(-\frac{1}{2}\frac{(x_1 - \mu_{pk})^2}{\sigma_{pk}^2}\right) \tag{309}$$

$$= \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}_k|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_k)'\mathbf{\Sigma}_k^{-1}(x - \mu_k)\right) \tag{310}$$

where the p-dimensional vector of means $\mu_k = [\mu_{1k}, \ldots, \mu_{pk}]'$ and the $p \times p$ matrix of variances $\mathbf{\Sigma}_k = \text{diag}(\sigma_{1k}^2, \ldots, \sigma_{pk}^2)$ can be estimated using the observations in each class.

For example, Table 3 shows the empirical means and standard deviations for the famous Iris database compiled by R.A. Fisher in 1936 consisting of four features for three classes of the iris flower. Consider, for example, the 51-th

	Sepal Length	Sepal Width	Petal Length	Petal Width	Prior
Setosa	(5.006, 0.3529)	(3.428,0.3791)	(1.462, 0.1737)	(0.246, 0.1054)	0.3333
Versicolor	(5.936, 0.5162)	(2.770,0.3138)	(4.260, 0.4699)	(1.326, 0.1976)	0.3333
Virginica	(6.588, 0.6359)	(2.974, 0.3225)	(5.552, 0.5519)	(2.026, 0.2747)	0.3333

Table 3: Mean and standard deviation for each feature in the Iris database.

entry in this database, for which the feature vector is

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 7 \\ 3.2 \\ 4.7 \\ 1.4 \end{bmatrix}.$$

Using the values in Table 3 and the assumption of independence in the naive Bayes classifier, we find

$$P(y=1|\mathbf{x}) \propto f_1(7|y=1) f_2(3.2|y=1) f_3(4.7|y=1) f_4(1.4|y=1) P(y=1) = 3.2674 \times 10^{-108}$$

$$P(y=2|\mathbf{x}) \propto f_1(7|y=2) f_2(3.2|y=2) f_3(4.7|y=2) f_4(1.4|y=2) P(y=2) = 0.0473$$

$$P(y=3|\mathbf{x}) \propto f_1(7|y=3) f_2(3.2|y=3) f_3(4.7|y=3) f_4(1.4|y=3) P(y=3) = 0.0117,$$

so that we assign this example to the class "Versicolor", which happens to be correct for this data point.

A.2 Linear and Quadratic Discriminant Analysis

For the case of a single feature x, that is $p = 1$, assuming a Gaussian distribution with mean μ_k and variance σ_k in each class leads to

$$P(y=k|x) = \frac{f(x|y=k)P(y=k)}{f(x)} = \frac{\frac{1}{\sqrt{2\pi}\sigma_k} \exp\left(-\frac{1}{2\sigma_k^2}(x-\mu_k)^2\right) P(y=k)}{f(x)} \tag{311}$$

Taking logarithm on both sides we find

$$\log P(y=k|x) = -\frac{x^2}{2\sigma_k^2} + x\frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} - \frac{1}{2}\log(2\pi\sigma_k^2) + \log P(y=k) - \log f(x) \tag{312}$$

If we assume further that $\sigma_k = \sigma$ for all classes, then we see that assigning an observation x to the class for which (311) is the largest is equivalent to assigning it to the class for which the *discriminant* function

$$\delta_k(x) = x\frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{2\sigma^2} + \log P(y = k) \tag{313}$$

is the largest. This corresponds to *linear discriminant analysis* (LDA) for the case of $p = 1$, where the term "linear" refers to the fact that the discriminant is a linear function of x. In the case of two classes, it is enough to consider the difference

$$\Delta_1(x) := \delta_1(x) - \delta_2(x) = x\frac{\mu_1 - \mu_2}{\sigma^2} - \frac{(\mu_1^2 - \mu_2^2)}{2\sigma^2} + \log\frac{P(y = 1)}{P(y = 2)}, \tag{314}$$

and decide that an observation with feature x belongs to the class 1 when $\Delta_1(x) \geq 0$ and to class 2 when $\Delta_1(x) < 0$. In the case of two classes with equal priors $P(y = 1) = P(y = 2)$ this reduces even further to a decision boundary of the form

$$x = \frac{\mu_1^2 - \mu_2^2}{2(\mu_1 - \mu_2)} = \frac{\mu_1 + \mu_2}{2}. \tag{315}$$

For three classes, it is enough to consider the differences

$$\begin{aligned}
\Delta_1(x) &:= \delta_1(x) - \delta_2(x) = x\frac{\mu_1 - \mu_2}{\sigma^2} - \frac{(\mu_1^2 - \mu_2^2)}{2\sigma^2} + \log\frac{P(y = 1)}{P(y = 2)} \\
\Delta_2(x) &:= \delta_2(x) - \delta_3(x) = x\frac{\mu_2 - \mu_3}{\sigma^2} - \frac{(\mu_2^2 - \mu_3^2)}{2\sigma^2} + \log\frac{P(y = 2)}{P(y = 3)}
\end{aligned}$$

and decide that an observation with feature x belongs to the class 1 when $\Delta_1(x) \geq 0$ and $\Delta_2(x) \geq 0$, to class 2 when $\Delta_1(x) < 0$ and $\Delta_2(x) \geq 0$, and to class 3 when $\Delta_1(x) < 0$ and $\Delta_2(x) < 0$.

For $p > 1$, we assume that, for each class, the feature vector $\mathbf{x}$ is distributed according to the multivariate Gaussian density

$$f(\mathbf{x}|y = k) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu_k)'\mathbf{\Sigma}^{-1}(\mathbf{x} - \mu_k)\right),$$

where μ_k is a $p \times 1$ vector means and $\mathbf{\Sigma}$ is a $p \times p$ covariance matrix, which we assume to be the same for all classes. It is then easy to see that the class with the largest probability $P(y = k|\mathbf{x})$ coincides with the class with the largest

discriminant function

$$\delta_k(\mathbf{x}) = \mathbf{x}'\boldsymbol{\Sigma}^{-1}\mu_k - \frac{\mu_k'\boldsymbol{\Sigma}^{-1}\mu_k}{2} + \log P(y=k), \tag{316}$$

which is the multidimensional analogue of (313). Because the covariance matrix is assumed to be common to all classes, this is still a linear function of the feature vector $\mathbf{x}$. Observe, however, that we do not assume that $\boldsymbol{\Sigma}$ is diagonal, which means that, unlike the naive Bayes classifier, linear discriminant analysis allows for non-zero correlation between the features.

If we drop the requirement that $\boldsymbol{\Sigma}_k = \boldsymbol{\Sigma}$ for all classes, we find that the discriminant function becomes

$$\delta_k(\mathbf{x}) = -\frac{(\mathbf{x}-\mu_k)'\boldsymbol{\Sigma}_k^{-1}(\mathbf{x}-\mu_k)}{2} + \log P(y=k), \tag{317}$$

which is then a quadratic function of the features $\mathbf{x}$, leading to a *quadratic discriminant analysis* (QDA).

A.3 Error Analysis

The error rate defined in (302) can be easily calculated with the help of the aptly called *confusion matrix*, namely a matrix whose columns are labelled by the true class of an observation and whose rows are labelled by the predicted class. For example, using a naive Bayes classifier for the Iris database results in the confusion matrix shown at the top of Table 4, from which we can see that whereas all Setosa flowers were classified correctly, 3 Versicolor flowers were misclassified as Virginica and 3 Virginica flowers were misclassified as Versicolor. It is easy to see that the training error rate corresponds to the sum of the off-diagonal terms in the diagonal matrix divided by the sum of all entries., which for the matrix at the top of Table 4 is equal to 4%. Using LDA for the Iris database leads to the confusion matrix at the bottom of Table 4, from which we obtain a training error rate of 2% instead.

Instead of the training error rate, which takes into account *all* misclassified data points, it is sometimes advantageous to focus on other metrics of performance for a classifier, especially if there are large discrepancies in the a priori probabilities for the different classes. This is particularly relevant in binary classification when one of the classes, typically referred as the *positive* class, corresponds to rare events, such as default in a credit card, a message that is spam, or the occurrence of a disease. In this case, it is common to label the

		True class		
		Setosa	Versicolor	Virginica
Predicted Class Naive Bayes	Setosa	50	0	0
	Versicolor	0	47	3
	Virginica	0	3	47

Predicted Class LDA	Setosa	50	0	0
	Versicolor	0	48	1
	Virginica	0	2	49

Table 4: Confusion matrix for naive Bayes Classifier and Linear Discriminant Analysis applied to Iris database.

		True class	
		Negative	Positive
Predicted Class	Negative	TN	FN
	Positive	FP	TP

Table 5: Confusion matrix for binary classification

entries of the confusion matrix as in Table 5, from which we can extract the following basic quantities

- TN (true negative): number of negative observations that are correctly classified
- FP (false positive): number of negative observations that are misclassified as positive
- FN (false negative): number of positive observations that are misclassified as negative
- TP (true positive): number of positive observations that are correctly classified

as well as

- N = TN + FP: total number of negative observations
- P = FN + TP: total number of positive observations
- N^*= TN+FN: total number of negative predictions
- P^*= FP+TP: total number of positive predictions

There are a number of performance measure that can then be calculate, including:

- Precision = TP/P*: proportion of positive predictions that are correct
- Recall = TP/P: proportion of positive observations that are correctly predicted
- Specificity = TN/N: proportion of negative observations that are correctly predicted
- Type I error = FP/N: proportion of negative observations that are incorrectly predicted
- Type II error = FN/P: proportion of positive observations that are incorrectly predicted
- $\hat{\mathcal{R}}_{\mathcal{D}}$ = (FN+FP)/(N+P): total error rate (as before)

It is clear from the definitions above that desirable properties for a good classifier are high Precision, Recall, and Specificity, but low Type I, Type II and training errors. We can also see that

$$\text{Recall} = \text{1-Type II error}$$
$$\text{Specificity} = \text{1-Type I error}$$

so reducing Type II error increases Recall, while reducing Type I error increases Specificity. But can these two errors be reduced simultaneously? Unfortunately not!

As we can see from the definition, a Type I error corresponds to a "false alarm", whereas a Type II error corresponds to a "miss". In general, using a more conservative classifier (for example, increasing the threshold p_0 in logistic regression) leads to higher Precision and Specificity (and therefore lower Type I error), but lower Recall (and therefore higher Type II error).

A.4 Examples

The two examples below are adapted from ISLR and should be compared with Example 2.10.

Example A.1. *Linear discriminant analysis for stock market data*

```
> library(MASS)
> library(ISLR)
> View(Smarket) # daily returns for S&P 500
> attach(Smarket)
> train =( Year <2005)
> Smarket.2005= Smarket [! train ,]
> Direction.2005=Direction[!train]
> lda.fit=lda(Direction~Lag1+Lag2,data=Smarket ,subset=train)
> lda.fit
> plot(lda.fit)
> lda.pred=predict(lda.fit, Smarket.2005)
> names(lda.pred)
> predictions<-as.data.frame(lda.pred)
> lda.class=lda.pred$class
> table(lda.class ,Direction.2005)
> mean(lda.class==Direction.2005)
> sum(lda.pred$posterior[,1]>=.5)
> sum(lda.pred$posterior[,1]<.5)
```

Example A.2. *Quadratic discriminant analysis for stock market data*

```
> qda.fit=qda(Direction~Lag1+Lag2,data=Smarket ,subset=train)
> qda.class=predict(qda.fit,Smarket.2005)$class
> table(qda.class ,Direction.2005)
> mean(qda.class==Direction.2005)
```

The next two examples are adapted from DAS.

Example A.3. *Linear discriminant analysis for NCAA data - two groups*

```
> ncaa = read.table("ncaa.txt",header=TRUE)
> x = as.matrix(ncaa[4:14])
> y1 = 1:32
> y1 = y1*0+1
> y2 = y1*0
> y = c(y1,y2)
> library (MASS)
> lda.fit = lda(y~x)
> plot(lda.fit)
```

```
> lda.pred=predict(lda.fit)
> predictions<-as.data.frame(lda.pred)
> table(lda.pred$class,y)
> mean(lda.pred$class==y)
```

Example A.4. *Linear discriminant analysis for NCAA data - 3 groups*

```
> y1 = rep(2,20)
> y2 = rep(1,20)
> y3 = rep(0,24)
> y = c(y1,y2,y3)
> lda.fit = lda(y~x)
> lda.pred=predict(lda.fit)
> predictions<-as.data.frame(lda.pred)
> table(lda.pred$class,y)
> mean(lda.pred$class==y)
```

B Support Vector Machines

B.1 Maximal Margin Classifier

Consider a binary classification problem with feature vector $\mathbf{x} \in \mathbb{R}^p$ and output $y \in \{-1, 1\}$, that is, the two classes are labeled by -1 and 1, respectively. By analogy with logistic regression, we seek to construct a hyperplane with coefficients $\mathbf{w}, b$ such that

$$\mathbf{w}\mathbf{x}_j + b = w_1 x_{1j} + \cdots w_p x_{pj} + b \geq M \qquad \text{if } y_j = 1$$

and

$$\mathbf{w}\mathbf{x}_j + b = w_1 x_{1j} + \cdots w_p x_{pj} + b \leq -M \qquad \text{if } y_j = -1$$

or equivalently

$$y_j(\mathbf{w}\mathbf{x}_j + b) \geq M, \qquad \forall j = 1, \ldots, n. \tag{318}$$

If such hyperplane exits, we say that it *separates* the observations in two classes with a margin M. The problem is that, if a separating hyperplane exits, then there must exist infinitely many other separating hyperplanes. A *maximal margin hyperplane* consists of the hyperplane obtained by choosing $\mathbf{w}, b$ such that M in equation (318) is as large as possible, subject to the additional constraint that

$$\sum_{i=1}^{p} w_i^2 = 1. \tag{319}$$

Observe that this last constraint simply imposes that the perpendicular distance from the j-th observation to the hyperplane is given by $y_j(\mathbf{w}\mathbf{x}_j + b)$.

Having obtained the parameters $\widehat{\mathbf{w}}, \widehat{b}$ as the solution of this optimization problem, we can classify a test observation $\mathbf{x}_0$ using the *maximal margin classifier* as follows:

$$\widehat{y}_0 = \begin{cases} 1 & \text{if} \quad \widehat{\mathbf{w}}\mathbf{x}_0 + \widehat{b} \geq 0 \\ -1 & \text{if} \quad \widehat{\mathbf{w}}\mathbf{x}_0 + \widehat{b} < 0 \end{cases} \tag{320}$$

B.2 Support Vector Classifier

One key problem with the procedure just described is that there might not be a separating hyperplane for a given set of observations. Even when the maximal separating hyperplane exists, it suffers from the drawback of being too sensitive to individual observations. For example, the addition of a single observation

can drastically change the maximal separating hyperplane, a typical example of large variance and overfitting.

To deal with both issues, we consider a generalized optimization problem consisting of choosing $\mathbf{w}, b$ and *slack variables* $\epsilon_1, \ldots, \epsilon_n \geq 0$ such that M satisfying

$$y_j(\mathbf{w}\mathbf{x}_j + b) \geq M(1 - \epsilon_j), \qquad \forall j = 1, \ldots, n, \tag{321}$$

is as large as possible, subject to (319) and

$$\sum_{j=1}^{n} \epsilon_j \leq \frac{1}{C}, \tag{322}$$

where C is a tuning parameter. A large C means that the margin constraint is tight (that is, can be violated by only a few points), which corresponds to small bias and potentially high variance. Conversely, a small C means that more points can violate the margin constraint (and even be on the wrong side of the hyperplane), leading to smaller variance but potentially high bias. We therefore see that $1/C$ plays a role analogous to the regularization constant λ.

It can be shown that the solution to this optimization problem is affected only by data points for which either $y_j(\mathbf{w}\mathbf{x}_j + b) = M$ (that is, are on the margin) or for which $\epsilon_j > 0$ (that is, violate the margin). For this reason, such data points are called *support vectors*. Specifically, if we denote the collection of indices for support vectors by $\mathcal{S}$, it can be shown that the support vector classifier can be represented as

$$\widehat{f}(\mathbf{x}) := \widehat{\mathbf{w}}\mathbf{x} + \widehat{b} = \sum_{j \in \mathcal{S}} \alpha_j \langle \mathbf{x}, \mathbf{x}_j \rangle + \widehat{b}, \tag{323}$$

where

$$\langle \mathbf{x}, \mathbf{x}_j \rangle = \sum_{i=1}^{p} x_i x_{ij} \tag{324}$$

is the inner product between two vectors in $\mathbb{R}^p$. This can then be use to classify a test observation $\mathbf{x}_0$ according to

$$\widehat{y}_0 = \begin{cases} 1 & \text{if} \quad \widehat{f}(\mathbf{x}_0) \geq 0 \\ -1 & \text{if} \quad \widehat{f}(\mathbf{x}_0) < 0 \end{cases} \tag{325}$$

B.3 Support Vector Machines

A support vector machine is a generalization of the support vector classifier when the inner product in (323) is replaced by a kernel $K(\mathbf{u}, \mathbf{v})$ measuring the *similarity* between the vectors $\mathbf{u}$ and $\mathbf{v}$. Using a *linear* kernel of the form

$$K(\mathbf{u}, \mathbf{v}) = \langle \mathbf{u}, \mathbf{v} \rangle \tag{326}$$

leads to the support vector classifier of the previous section. Two other popular choices of kernel are the *polynomial kernel* of degree d given by

$$K(\mathbf{u}, \mathbf{v}) = (1 + \langle \mathbf{u}, \mathbf{v} \rangle)^d \tag{327}$$

and the *radial* or *Gaussian* kernel of the form

$$K(\mathbf{u}, \mathbf{v}) = \exp\left(-\gamma \sum_{i=1}^{p} (u_i - v_i)^2\right). \tag{328}$$

These nonlinear kernels can lead to very complex decision boundaries for the classification problem.

B.4 Relationship with Logistic Regression

It turns out that the optimization problem characterizing a support vector classifier is equivalent to

$$\min_{b, \mathbf{w}} \left\{ C \sum_{j=1}^{n} \max(0, 1 - y_j f(\mathbf{x}_j)) + \sum_{i=1}^{p} w_i^2 \right\}, \tag{329}$$

where

$$f(\mathbf{x}) = \mathbf{w}\mathbf{x} + b. \tag{330}$$

This is very similar to the regularized logistic regression with $C = 1/\lambda$. The key difference is that the loss function above is exactly zero whenever $y_j f(\mathbf{x}_j) \geq 1$, whereas for logistic regression the loss function in (35) decays rapidly to zero when $y_j = 1$ and $\mathbf{w}\mathbf{x} + b$ is large, but otherwise the two functions are very similar.

Moreover, we can see that replacing the inner product $\mathbf{w}\mathbf{x}$ with a general kernel has a similar effect to adding nonlinear features to the logistic regression.

C Some Notes on Neural Networks in Practice

In general, an L-deep neural network has $L+1$ layers, consisting of an input layer denoted by $\ell = 0$ and having $p_0 = p$ nodes, an output layer denoted by L and having $p_L = q$ nodes, and $L-1$ *hidden* layers denoted by $\ell = 1, \ldots, L-1$ and having p_ℓ nodes. The weights connecting layer $\ell - 1$ to layer ℓ are given by the $p_\ell \times p_{\ell-1}$ matrix $\mathbf{W}^{[\ell]}$ and the bias in layer ℓ is given by the $p_\ell \times 1$ column vector $\mathbf{b}^{[\ell]}$. In addition, each layer has its own activation function $g^{[\ell]}$. In most applications, the activation function used in the output layer is the sigmoid function, because this is more directly related to classification problems and their associated cost functions. As it turns out, using a different activation function in the hidden layer, however, can vastly improve the performance of the neural network.

As before, we arrange the n data points as the $p \times n$ input matrix $\mathbf{X}$ and the $q \times n$ output matrix $\mathbf{Y}$.

C.1 Binary classification with one hidden layer

Consider first the example with $L = 2$ (one input layer, one hidden layer, one output layer) and $p_L = q = 1$ (single output node), which can be used for binary classification when a simple linear regression is unable to produce a sufficiently complicated decision boundary.

In this case, forward propagation of a single input $\mathbf{x}_j = \mathbf{a}_j^{[0]}$ consist of the following steps:

$$\mathbf{z}_j^{[1]} = \mathbf{W}^{[1]}\mathbf{a}_j^{[0]} + \mathbf{b}^{[1]} \tag{331}$$

$$\mathbf{a}_j^{[1]} = g^{[1]}(\mathbf{z}_j^{[1]}) \tag{332}$$

$$z_j^{[2]} = \mathbf{w}^{[2]}\mathbf{a}_j^{[1]} + b^{[2]} \tag{333}$$

$$a_j^{[2]} = g^{[2]}(z_j^{[2]}) \tag{334}$$

where $\mathbf{W}^{[1]}$ is a $p_1 \times p$ matrix, $\mathbf{b}^{[1]}$ is a $p_1 \times 1$ column vector, $\mathbf{w}^{[2]}$ is a $1 \times p_1$ row vector and $b^{[2]}$ is a scalar.

As before, in order to apply gradient descent, we need to compute the derivatives of the cost function

$$J_j(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{w}^{[2]}, b^{[2]}) = -\big(y_j \log a_j^{[2]} + (1 - y_j)\log(1 - a_j^{[2]})\big). \tag{335}$$

We do this again by backward propagation, which in this case starts with the following familiar steps:

$$\frac{\partial J_j}{\partial a_j^{[2]}} = -\frac{y_j}{a_j^{[2]}} + \frac{1-y_j}{1-a_j^{[2]}} \tag{336}$$

$$\frac{\partial J_j}{\partial z_j^{[2]}} = \frac{\partial J_j}{\partial a_j^{[2]}} \frac{\partial a_j^{[2]}}{\partial z_j^{[2]}} = \frac{\partial J_j}{\partial a_j^{[2]}} a_j^{[2]}(1-a_j^{[2]}) = a_j^{[2]} - y_j \tag{337}$$

$$\frac{\partial J_j}{\partial \mathbf{w}^{[2]}} = \frac{\partial J_j}{\partial z_j^{[2]}} \frac{\partial z_j^{[2]}}{\partial \mathbf{w}^{[2]}} = \frac{\partial J_j}{\partial z_j^{[2]}} \mathbf{a}_j^{[1]\prime} \tag{338}$$

$$\frac{\partial J_j}{\partial b^{[2]}} = \frac{\partial J_j}{\partial z_j^{[2]}} \frac{\partial z_j^{[2]}}{\partial b^{[2]}} = (a_j^{[2]} - y_j) \tag{339}$$

where we used the fact that $g^{[2]}$ is the sigmoid function. We recognize these as the analogues of (49)-(52) with the hidden layer activation vector $\mathbf{a}_j^{[1]}$ playing the role of the input vector. The novelty arises in the next series of applications of the chain rule, namely

$$\frac{\partial J_j}{\partial \mathbf{a}_j^{[1]}} = \frac{\partial J_j}{\partial z_j^{[2]}} \frac{\partial z_j^{[2]}}{\partial \mathbf{a}_j^{[1]}} = \frac{\partial J_j}{\partial z_j^{[2]}} \mathbf{w}^{[2]\prime} \tag{340}$$

$$\frac{\partial J_j}{\partial \mathbf{z}_j^{[1]}} = \frac{\partial J_j}{\partial \mathbf{a}_j^{[1]}} \frac{\partial \mathbf{a}_j^{[1]}}{\partial \mathbf{z}_j^{[1]}} = \left(\frac{\partial J_j}{\partial z_j^{[2]}} \mathbf{w}^{[2]\prime} \right) * \frac{dg}{dz}\left(\mathbf{z}_j^{[1]}\right) \tag{341}$$

$$\frac{\partial J_j}{\partial \mathbf{W}^{[1]}} = \frac{\partial J_j}{\partial \mathbf{z}_j^{[1]}} \frac{\partial \mathbf{z}_j^{[1]}}{\partial \mathbf{W}^{[1]}} = \frac{\partial J_j}{\partial \mathbf{z}_j^{[1]}} \mathbf{a}_j^{[0]\prime} \tag{342}$$

$$\frac{\partial J_j}{\partial \mathbf{b}^{[1]}} = \frac{\partial J_j}{\partial \mathbf{z}_j^{[1]}} \frac{\partial \mathbf{z}_j^{[1]}}{\partial \mathbf{b}^{[1]}} = \frac{\partial J_j}{\partial \mathbf{z}_j^{[1]}}, \tag{343}$$

where $\mathbf{a} * \mathbf{b}$ denotes the componentwise product of two vectors.

Forward propagation of all data points simultaneously then takes the form

$$\begin{cases} \mathbf{Z}^{[1]} &= \mathbf{W}^{[1]}\mathbf{A}^{[0]} + \mathbf{b}^{[1]}\mathbf{1}_n \\ \mathbf{A}^{[1]} &= g^{[1]}(\mathbf{Z}^{[1]}) \end{cases} \tag{344}$$

$$\begin{cases} \mathbf{Z}^{[2]} &= \mathbf{w}^{[2]}\mathbf{A}^{[1]} + b^{[2]}\mathbf{1}_n \\ \mathbf{A}^{[2]} &= g^{[2]}(\mathbf{Z}^{[2]}). \end{cases} \tag{345}$$

Using the total cost function

$$J = \frac{1}{n}\sum_{j=1}^{n} J_j, \tag{346}$$

backward propagation takes the form

$$\begin{cases} \frac{\partial J}{\partial \mathbf{Z}^{[2]}} & = \frac{1}{n}(\mathbf{A}^{[2]} - \mathbf{Y}) \\ \frac{\partial J}{\partial \mathbf{w}^{[2]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{w}^{[2]}} = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\mathbf{A}^{[1]\prime} \\ \frac{\partial J}{\partial b^{[2]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial b^{[2]}} = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\mathbf{1}_n' \\ \frac{\partial J}{\partial \mathbf{A}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{A}^{[1]}} = \mathbf{w}^{[2]\prime}\frac{\partial J}{\partial \mathbf{Z}^{[2]}} \end{cases} \tag{347}$$

$$\begin{cases} \frac{\partial J}{\partial \mathbf{Z}^{[1]}} & = \frac{\partial J}{\partial \mathbf{A}^{[1]}}\frac{\partial \mathbf{A}^{[1]}}{\partial \mathbf{Z}^{[1]}} = \left(\mathbf{w}^{[2]\prime}\frac{\partial J}{\partial \mathbf{Z}^{[2]}}\right) * \frac{dg^{[1]}}{dz}\left(\mathbf{Z}^{[1]}\right) \\ \frac{\partial J}{\partial \mathbf{W}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\frac{\partial \mathbf{Z}^{[1]}}{\partial \mathbf{W}^{[1]}} = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\mathbf{A}^{[0]\prime} \\ \frac{\partial J}{\partial \mathbf{b}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\frac{\partial \mathbf{Z}^{[1]}}{\partial \mathbf{b}^{[1]}} = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\mathbf{1}_n'. \end{cases} \tag{348}$$

C.2 Multiple classes with one hidden layer

Consider now the example with $L = 2$ (one input layer, one hidden layer, one output layer) and $p_L = q > 1$ (multiple output nodes), which can be used for classification problems with more than two classes as a neural network alternative to logistic regression for multiple classes.

It is straightforward to see that forward propagation of all data points in this case takes the form

$$\begin{cases} \mathbf{Z}^{[1]} & = \mathbf{W}^{[1]}\mathbf{A}^{[0]} + \mathbf{b}^{[1]}\mathbf{1}_n \\ \mathbf{A}^{[1]} & = g^{[1]}(\mathbf{Z}^{[1]}) \end{cases} \tag{349}$$

$$\begin{cases} \mathbf{Z}^{[2]} & = \mathbf{W}^{[2]}\mathbf{A}^{[1]} + \mathbf{b}^{[2]}\mathbf{1}_n \\ \mathbf{A}^{[2]} & = g^{[2]}(\mathbf{Z}^{[2]}) \end{cases} \tag{350}$$

where $\mathbf{W}^{[1]}$ is a $p_1 \times p$ matrix, $\mathbf{b}^{[1]}$ is a $p_1 \times 1$ column vector, $\mathbf{W}^{[2]}$ is a $q \times p_1$ row vector and $\mathbf{b}^{[2]}$ is a $q \times 1$ column vector.

A similar calculation as before for the cost function

$$J(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]}) = -\frac{1}{n}\sum_{j=1}^{n}\sum_{k=1}^{q}\left(y_{kj}\log A_{kj}^{[2]} + (1-y_{kj})\log(1-A_{kj}^{[2]})\right) \tag{351}$$

leads to the following steps for backward propagation:

$$\begin{cases} \frac{\partial J}{\partial \mathbf{Z}^{[2]}} & = \frac{1}{n}(\mathbf{A}^{[2]} - \mathbf{Y}) \\ \frac{\partial J_j}{\partial \mathbf{W}^{[2]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{W}^{[2]}} = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\mathbf{A}^{[1]\prime} \\ \frac{\partial J_j}{\partial \mathbf{b}^{[2]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{b}^{[2]}} = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\mathbf{1}_n' \\ \frac{\partial J}{\partial \mathbf{A}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[2]}}\frac{\partial \mathbf{Z}^{[2]}}{\partial \mathbf{A}^{[1]}} = \mathbf{W}^{[2]\prime}\frac{\partial J}{\partial \mathbf{Z}^{[2]}} \end{cases} \tag{352}$$

$$\begin{cases} \frac{\partial J}{\partial \mathbf{Z}^{[1]}} & = \frac{\partial J}{\partial \mathbf{A}^{[1]}}\frac{\partial \mathbf{A}^{[1]}}{\partial \mathbf{Z}^{[1]}} = \left(\mathbf{W}^{[2]\prime}\frac{\partial J}{\partial \mathbf{Z}^{[2]}}\right) * \frac{dg^{[1]}}{dz}\left(\mathbf{Z}^{[1]}\right) \\ \frac{\partial J}{\partial \mathbf{W}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\frac{\partial \mathbf{Z}^{[1]}}{\partial \mathbf{W}^{[1]}} = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\mathbf{A}^{[0]\prime} \\ \frac{\partial J}{\partial \mathbf{b}^{[1]}} & = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\frac{\partial \mathbf{Z}^{[1]}}{\partial \mathbf{b}^{[1]}} = \frac{\partial J}{\partial \mathbf{Z}^{[1]}}\mathbf{1}_n'. \end{cases} \tag{353}$$

References

[1] Beatrice Acciaio, Anastasis Kratsios, and Gudmund Pammer. Metric hypertransformers are universal adapted maps. *arXiv preprint arXiv:2201.13094*, 2022.

[2] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

[3] Jean-Pierre Aubin and Hélène Frankowska. *Set-valued analysis*, volume 2 of *Systems & Control: Foundations & Applications*. Birkhäuser Boston, Inc., Boston, MA, 1990.

[4] Amir Beck. *Introduction to nonlinear optimization: Theory, algorithms, and applications with MATLAB*. SIAM, 2014.

[5] Hans Buehler, Lukas Gonon, Josef Teichmann, and Ben Wood. Deep hedging. *Quantitative Finance*, 19(8):1271–1291, 2019.

[6] Patrick Cheridito, Arnulf Jentzen, and Florian Rossmannek. Efficient approximation of high-dimensional functions with neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[7] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[8] Ryszard Engelking. *General topology*, volume 6 of *Sigma Series in Pure Mathematics*. Heldermann Verlag, Berlin, second edition, 1989. Translated from the Polish by the author.

[9] Charles Fefferman. Whitney's extension problem for C^m. *Ann. of Math. (2)*, 164(1):313–359, 2006.

[10] Charles Fefferman. Whitney's extension problems and interpolation of data. *Bull. Amer. Math. Soc. (N.S.)*, 46(2):207–220, 2009.

[11] James Gareth, Witten Daniela, Hastie Trevor, and Tibshirani Robert. *An introduction to statistical learning: with applications in R*. Spinger, 2013.

[12] Lukas Gonon, Lyudmila Grigoryeva, and Juan-Pablo Ortega. Approximation bounds for random neural networks and reservoir systems. *Annals of Applied Probability (Forthcoming)*, 2022.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[14] JackT (https://math.stackexchange.com/users/454450/jackt). (solved) lipschitz continuity of huber function. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/4424276 (version: 2022-04-10).

[15] Sham M Kakade, Karthik Sridharan, and Ambuj Tewari. On the complexity of linear prediction: Risk bounds, margin bounds, and regularization. *Advances in neural information processing systems*, 21, 2008.

[16] Olav Kallenberg and Olav Kallenberg. *Foundations of modern probability*, volume 2. Springer, 1997.

[17] Patrick Kidger and Terry Lyons. Universal Approximation with Deep Narrow Networks. In Jacob Abernethy and Shivani Agarwal, editors, *Proceedings of Thirty Third Conference on Learning Theory*, volume 125 of *Proceedings of Machine Learning Research*, pages 2306–2327. PMLR, 09–12 Jul 2020.

[18] Anastasis Kratsios and Ievgen Bilokopytov. Non-euclidean universal approximation. *Advances in Neural Information Processing Systems*, 33:10635–10646, 2020.

[19] Anastasis Kratsios and Cody Hyndman. NEU: a meta-algorithm for universal UAP-invariant feature representation. *J. Mach. Learn. Res.*, 22:Paper No. 92, 51, 2021.

[20] Anastasis Kratsios and Leonie Papon. Universal approximation theorems for differentiable geometric deep learning. *Journal of Machine Learning Research*, 23(196):1–73, 2022.

[21] Anastasis Kratsios and Léonie Papon. Universal approximation theorems for differentiable geometric deep learning. *Journal of Machine Learning Research*, 23(196):1–73, 2022.

[22] Anastasis Kratsios and Behnoosh Zamanlooy. Do reLU networks have an edge when approximating compactly-supported functions? *Transactions on Machine Learning Research*, 2022.

[23] Anastasis Kratsios and Behnoosh Zamanlooy. Learning sub-patterns in piecewise continuous functions. *Neurocomputing*, 480:192–211, 2022.

[24] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[25] Song Mei and Andrea Montanari. The generalization error of random features regression: precise asymptotics and the double descent curve. *Comm. Pure Appl. Math.*, 75(4):667–766, 2022.

[26] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[27] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.

[28] Zuowei Shen, Haizhao Yang, and Shijun Zhang. Optimal approximation rate of relu networks in terms of width and depth. *Journal de Mathématiques Pures et Appliquées*, 157:101–135, 2022.

[29] Terence Tao and Van Vu. Random matrices: the distribution of the smallest singular values. *Geom. Funct. Anal.*, 20(1):260–297, 2010.

[30] Josef Teichmann and Wahid Khosrawi. Deep hedging. `https://gist.github.com/jteichma/4d9c0079dbf4e9c3cdff3fd1befabd23`, 2021.

[31] Edward Wagstaff, Fabian B Fuchs, Martin Engelcke, Michael A Osborne, and Ingmar Posner. Universal approximation of functions on sets. *Journal of Machine Learning Research*, 23(151):1–56, 2022.

[32] whuber (https://stats.stackexchange.com/users/919/whuber). How to derive the ridge regression solution? Cross Validated. URL:https://stats.stackexchange.com/q/164546 (version: 2022-04-12).

[33] Dmitry Yarotsky. Error bounds for approximations with deep relu networks. *Neural Networks*, 94:103–114, 2017.

[34] Dmitry Yarotsky and Anton Zhevnerchuk. The phase diagram of approximation rates for deep neural networks. *Advances in neural information processing systems*, 33:13005–13015, 2020.

[35] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.