Statistical Learning for Computer Science Students

March 2020

Contents

Introduction **2**

Lecture 1: Mathematical Background **3**

1 Recap: Probability and Statistics **3**
1.1 Probability Space . . . 3
1.2 Random Variables, discrete and continuous . . . 4
1.3 Mean and Variance . . . 5
1.4 A little Statistics: Mean and variance estimation . . . 7

2 High dimensional distributions **8**
2.1 Basic definitions . . . 8
2.2 Normal Distribution . . . 8
2.3 Covariance Matrix . . . 10
2.4 Estimating the Covariance Matrix . . . 11
2.5 Linear Transformations of the Data Set . . . 12

3 Probability Inequalities **14**
3.1 Motivation and Background . . . 14
3.2 Recap . . . 15
3.3 Coin Prediction . . . 16
3.3.1 Hoeffding's Inequality . . . 19
3.3.2 Coin Prediction Revisited . . . 19

Lecture 2: The Linear Model **20**

4 The Linear Model, Noiseless Case **20**
4.1 Problem: Customer Lifetime Value prediction . . . 20
4.2 The training data matrix . . . 20
4.3 Setup . . . 20
4.4 The Linear Hypothesis Class . . . 21
4.4.1 The Realizable Case . . . 21
4.4.2 The Non-Realizable Case . . . 21

4.4.3 Geometric interpretation . . . 21
4.5 Designing the Learning Algorithm . . . 22
4.5.1 The Loss Function . . . 22
4.5.2 Empirical Risk Minimization . . . 22
4.5.3 Least Squares . . . 23
4.6 The Normal Equations . . . 24
4.6.1 Solving the Normal Equations . . . 24
4.7 Singular Value Decomposition . . . 25
4.7.1 Making the SVD solution numerically stable . . . 26
4.8 How many samples do we need to learn a linear function? . . . 26
4.9 Summary - Noiseless Case . . . 26

5 The Linear Model - Noisy Case **27**
5.1 Data Generation Model With Noise . . . 27
5.2 The Maximum Likelihood Principle . . . 28
5.3 Noise, Bias and Variance . . . 29
5.3.1 Polynomial fitting . . . 29
5.3.2 Polynomial fitting: no noise . . . 29
5.3.3 Bias . . . 32
5.4 Polynomial fitting: with noise . . . 33
5.5 Variance . . . 35
5.6 Variance in linear model: Geometrical Interpretation . . . 36
5.7 Bias-Variance Tradeoff . . . 38

Lecture 3: Classification **39**

6 Introduction **39**
6.1 Textbook references . . . 40
6.2 Some preliminaries . . . 41
6.3 Examples for classification problems . . . 42
6.4 Heart disease data . . . 44
6.5 Plotting and imagining a feature space $\mathbb{R}^d$ with binary labeled data . . . 44
6.6 What loss function should we use? . . . 45
6.7 Type-I and Type-II errors . . . 45
6.8 False Positive, False Negative and all that jazz . . . 47
6.9 Decision boundary . . . 48
6.10 Our goal in this lecture . . . 48

7 The Half-Space classifier **49**
7.1 Assumption - training sample is linearly separable . . . 50
7.2 Learning using ERM . . . 51
7.3 Computationally implementing ERM for halfspace classifiers . . . 51
7.4 Commercial break: Convex optimization . . . 52
7.5 Solving ERM for half-space by linear programming . . . 53
7.6 What about a training sample that's not linearly separable? . . . 53
7.7 Summary . . . 53

8 Support Vector Machines **53**
8.1 A new learning principle: Maximum Margin . . . 54
8.2 Hard SVM . . . 55
8.3 Soft SVM . . . 56
8.4 A family of learners . . . 57
8.5 When is SVM useful? . . . 57
8.6 Summary - Soft SVM . . . 57

9 Logistic Regression **58**
9.1 A probabilistic model for noisy labels . . . 58
9.2 The hypothesis class . . . 58
9.3 The learning principle: maximum likelihood . . . 59
9.4 Computational implementation . . . 60
9.5 Interpretability . . . 61
9.5.1 Which features were important . . . 61
9.5.2 Why was this label predicted? . . . 61
9.5.3 Example: Interpretability of linear regression . . . 61
9.5.4 Interpretability of logistic regression . . . 61
9.6 How to make predictions on a new sample: working with estimated class probabilities and choosing the cutoff . . . 61
9.7 Summary . . . 64

10 Nearest Neighbors **65**
10.1 No hypothesis class . . . 65
10.2 Prediction with k-nearest neighbors . . . 65
10.3 Choosing k . . . 66
10.4 Computational implementation of k-nearest neighbors . . . 67
10.5 Other sample spaces . . . 69
10.6 Summary . . . 69

11 Classification Trees **69**
11.1 Tree-induced, axis-parallel partitions of $\mathbb{R}^d$ . . . 70
11.2 The Regression Tree hypothesis class . . . 70
11.3 Any hypothesis in $\mathcal{H}_{CT}$ corresponds to a Decision Tree, and vice-versa . . . 71
11.4 How *not* to grow a classification tree . . . 72
11.5 How to grow a classification tree . . . 74
11.6 Growing classification Trees a-la CART . . . 74
11.7 Why to prune a classification tree . . . 75
11.8 How to make predictions on a new sample . . . 76
11.9 Interpretability . . . 76
11.10Summary- Classification Trees . . . 76

Lecture 4: PAC Theory of Statistical Learning, Part I **77**

12 Introduction **77**

13 A Theoretical framework for learning **78**
13.1 A Data-generation Model . . . 78
13.2 Classifiers . . . 78
13.3 The framework so far . . . 79
13.4 Our goal in this lecture . . . 79
13.5 Learning as a Game - first attempt . . . 80

14 Probably correct & Approximately correct learners **81**
14.1 The game - for Probably Approximately correct learners . . . 84

15 No Free Lunch and Hypothesis Classes **85**
15.1 No Free Lunch! . . . 85
15.2 We need hypothesis classes . . . 86
15.3 Updating the game one last time . . . 87
15.4 Example: Threshold functions . . . 88
15.4.1 Threshold functions - conclusion . . . 90

16 PAC learning **90**
16.1 Finite hypothesis classes are PAC learnable . . . 91
16.1.1 Empirical Risk Minimization . . . 91
16.1.2 Learning Finite Classes . . . 92

17 VC Dimension **94**
17.1 Motivation . . . 94
17.2 Formal Definition . . . 95
17.3 Exercises to help you understand the definition of VC-dimension . . . 95
17.3.1 Axis aligned rectangles . . . 95
17.3.2 Finite classes . . . 96
17.3.3 Half-spaces through the origin . . . 96

Lecture 5: PAC Theory of Statistical Learning, Part II **97**

18 Recap of PAC Theory so far **97**

19 Finite Hypothesis Classes are PAC learnable **101**
19.0.1 Empirical Risk Minimization . . . 101
19.0.2 Learning Finite Classes . . . 102

20 The Fundamental Theorem of Statistical Learning **104**

21 Agnostic PAC: Extending our theoretical framework **105**
21.1 Moving from a probability distribution over $\mathcal{X}$ to a joint probability distribution over $\mathcal{X} \times \mathcal{Y}$ . . . 106
21.2 Removing the realizability assumption . . . 107
21.3 Introducing a general loss function . . . 108

22 Agnostic-PAC learnability **109**
22.1 Probably Approximately correct learner - in the new framework. 109
22.2 Agnostic-PAC learnability 109
22.3 PAC learnability is equivalent to Agnostic-PAC learnability 110

23 Back to the Fundamental Theorem of Statistical Learning **110**
23.1 Empirical Risk Minimization strikes again 110
23.2 ERM makes sense due to WLLN 111
23.3 The Fundamental Theorem - now with Agnostic-PAC 111

24 A Taste of the Proof **112**
24.1 The uniform convergence property 112

25 Proving that if $VCdim(\mathcal{H}) < \infty$ then $\mathcal{H}$ has the uniform convergence property **113**
25.1 Achieving uniformity in both $\mathcal{H}$ and $\mathcal{D}$ 114
25.2 The case of finite $\mathcal{H}$ 114
25.3 The general case - infinite $\mathcal{H}$ 115
25.3.1 First part of the proof: if $|\mathcal{H}_C|$ **grows polynomially in** $|C|$ **then** $\mathcal{H}$ **has the uniform convergence property** 116
25.3.2 If $VCdim(\mathcal{H}) < \infty$, $|\mathcal{H}_C|$ only grows polynomially in $|C|$ 117
25.3.3 Summary 117

Lecture 6: Ensemble Methods - Bagging and Boosting **118**

26 Introduction **118**
26.1 Bias/Variance 119
26.2 Ensemble / Committee methods 120
26.3 The uncorrelated case. 120
26.4 The correlated case. 121
26.5 Summary 122

27 Committee methods in machine learning **122**

28 The Bootstrap **123**
28.1 Why does the Bootstrap work? 123

29 Bagging **125**
29.1 This is shockingly effective 126
29.2 Bagging reduces variance 127
29.3 Decorrelation 128
29.4 Random Forest: Bagging of Decision Trees + De-correlation 128
29.5 Some discussion points about Bagging 129
29.6 Random Forest classifier summary 130

30 Boosting **131**
30.1 Classification problem with a weighted sample 133
30.2 Adaboost 134
30.3 PAC view of boosting 136

30.4 Bias and variance in boosting . . . 137
30.5 It's often better to Boost very simple learners . . . 137

31 Bagging vs Boosting - Comparison **138**

32 Summary **139**

Lecture 7: Regression, Regularization, Model Selection and Model Evaluation **140**

33 Introduction **140**

34 Regularization **140**
34.1 The setup: Choosing $h \in \mathcal{H}$ by minimizing fidelity . . . 140
34.2 Adding a regularization term . . . 141
34.3 Let's focus on Euclidean sample space, regression problems and ERM fidelity for the square loss . . . 142

35 CART Regression Trees **143**
35.1 Regression Trees . . . 143
35.2 Growing a CART regression tree . . . 144
35.3 Pruning a CART regression tree - using regularization . . . 145
35.4 The complete Random Forest algorithms for regression and for classification . . . 146

36 Modern Regression methods on $\mathbb{R}^d$ **146**
36.1 Linear Regression with high-dimensional data . . . 146
36.2 Best subset selection . . . 147
36.3 ℓ_0 regularization: Best subset selection . . . 149
36.4 Ridge . . . 149
36.5 Lasso . . . 151

37 The ℓ_1 norm induces sparsity **153**
37.1 Unit balls . . . 153
37.2 Orthogonal design . . . 155

38 ℓ_1-regularized logistic regression **156**

39 Practical considerations **157**
39.1 Choosing lambda . . . 157
39.2 Intercept . . . 157
39.3 Software implementation . . . 157

40 Introduction **157**
40.1 Model Selection . . . 157
40.2 Model Evaluation . . . 158
40.3 What is the generalization error, exactly? . . . 158

41 Bias-Variance **159**
41.1 Bias-Variance decomposition for square error loss . . . 159
41.2 The bias-variance tradeoff . . . 161

42 Can't naively use training sample for model selection / evaluation **163**

43 Model selection and evaluation with unlimited data **164**

44 k-fold Cross Validation **165**
44.1 Cross validation for model selection . . . 165
44.2 Cross validation for model evaluation . . . 166
44.3 Considerations in choosing number of folds k . . . 167

45 Bootstrap **167**

46 Two common mistakes in model evaluation and how to avoid them **168**
46.1 Over-estimating generalization error . . . 168
46.2 Under-estimating generalization error . . . 169
46.3 What can we do to avoid under-estimating generalization error? . . . 169

Lecture 8: Unsupervised Learning **171**

47 Unsupervised learning: Introduction **171**
47.1 This lecture . . . 174

48 Dimension reduction **174**
48.1 Linear dimension reduction . . . 175
48.2 Principal Components Analysis (PCA) . . . 176
48.3 PCA as Variance Maximization . . . 177
48.4 PCA - formal definition. . . . 178
48.5 Applying PCA dimension reduction to an arbitrary vector in $\mathbb{R}^d$ . . . 178
48.6 The subtle difference between the projected points and their coordinates. . . . 179
48.7 Interpreting and using the principal vectors as "typical data points" . . . 182
48.8 Practical considerations. . . . 183
48.8.1 Fast computation of PCA . . . 183
48.8.2 Choosing k . . . 184

49 Clustering **185**
49.1 k-means . . . 187

Lecture 9: ML Actually **190**

Lecture 10: Convex Optimization and SGD **191**

Lecture 11: Online Learning and RL **192**

Introduction

This is intro

Lecture 1: Mathematical Background

1 Recap: Probability and Statistics

1.1 Probability Space

Our intuitive notion of probability is related to uncertainty we have about events in the world, for example - will it rain tomorrow? How much will I get in the final exam? Although we cannot predict with certainty the answers to these questions, we usually know what is the set of all possible outcomes. This gives us the first ingredient in any *probability space.* [1]

Definition 1 *A* sample space Ω *is a set that contains all possible outcomes.* $\omega \in \Omega$ *denotes a single outcome.*

Examples:

- Coin toss: $\Omega = \{H, T\}$
- Toss a coin until heads comes up: $\Omega = \left\{T^{n-1}H : n \in \mathbb{N}_0\right\}$
- Rolling two dice: $\Omega = \{1, \ldots, 6\}^2$
- Waiting time at the post office: $\Omega = [0, \infty)$

Suppose someone threw a die (singular of dice), and told us that the result is an even number. This result is not an element in Ω, but it tells us that the outcome is an element in the subset $\{2, 4, 6\}$.

Definition 2 *An* event A *is any subset of possible outcomes,* $A \subseteq \Omega$.

Quick reminder:

- A^c (or sometimes $\overline{A}$) denotes the *complement* of A, $A^c = \Omega \backslash A$. In other words, A^c is the event that A did not occur.
- We say that A and B are *disjoint events* if $A \cap B = \varnothing$.
- Ω and $\varnothing$ are also events. If A and B are two events, then $A \cup B$ is also an event, and so is $A \cap B$.

Intuitively, a probability function assigns to an event a number which quantifies our knowledge or belief about the possibility of observing any outcome in that event.

Definition 3 *A* probability space *is a tuple* [2] $(\Omega, \mathcal{D})$ *where* Ω *is a* sample space *and* $\mathcal{D} : 2^{\Omega} \to \mathbb{R}$ *is a probability function such that*

1. $\mathcal{D}(\Omega) = 1$
2. *for all* $\omega \in \Omega, \mathcal{D}(\omega) \in [0, 1]$
3. *for all* $A, B \subseteq \Omega$ *such that* $A \cap B = \varnothing$, *we have* $\mathcal{D}(A \cup B) = \mathcal{D}(A) + \mathcal{D}(B)$.

[1] This section is based on lecture notes by Michal Moshkovitz and Alon Gonen

[2] For our needs this simple and intuitive definition is sufficient, though, richer definitions exist.

Example 1 *Suppose we throw two fair dice, then* $\Omega = \{1,\ldots,6\}^2$ *and* $\mathcal{D}((i,j)) = \frac{1}{36}$

Exercise 1 *For all* $A, B \subseteq \Omega$*:* $\mathcal{D}(A \cup B) = \mathcal{D}(A) + \mathcal{D}(B) - \mathcal{D}(A \cap B)$

Solution:
$A \cup B = (A \backslash B) \cup (B \backslash A) \cup (A \cap B)$, $A = (A \backslash B) \cup (A \cap B)$ and $B = (B \backslash A) \cup (A \cap B)$.

$$\mathcal{D}(A \cup B) = \mathcal{D}(A \backslash B) + \mathcal{D}(B \backslash A) + \mathcal{D}(A \cap B) = \mathcal{D}(A) - \mathcal{D}(A \cap B) + \mathcal{D}(B) - \cancel{\mathcal{D}(A \cap B)} + \cancel{\mathcal{D}(A \cap B)}$$

Definition 4 $A, B \subseteq \Omega$ *are called* independent *if the occurrence of one does not affect the probability of occurrence of the other. Consequently:*

$$\mathcal{D}(A \cap B) = \mathcal{D}(A) \cdot \mathcal{D}(B).$$

Exercise 2 *If* A *and* B *are independent then* A *and* B^c *are also independent.*

Solution: $\mathcal{D}(A) = \mathcal{D}(A \cap B) + \mathcal{D}(A \cap B^c) \Rightarrow \mathcal{D}(A \cap B^c) = \mathcal{D}(A)(1 - \mathcal{D}(B)) = \mathcal{D}(A) \cdot \mathcal{D}(B^c)$.

Lemma 1 ***(The union bound)*** *Let* $(\Omega, \mathcal{D})$ *be a probability space. The probability function is* sub-additive*, i.e., for any sequence* (A_k) *of events,*

$$\mathcal{D}(\cup_{k=1}^{\infty} A_k) \leqslant \sum_{k=1}^{\infty} \mathcal{D}(A_k)$$

Proof Let $B_1 = A_1$. For each $k \in \{2,3,\ldots\}$, let $B_k = A_k \backslash \cup_{i=1}^{k-1} A_i$. Note that the $B_1 ... B_k$ are disjoint, and $\cup_{i=1}^{k} A_i = \cup_{i=1}^{k} B_i$. Also, since $B_k \subseteq A_k$, $\mathcal{D}(B_k) \leqslant \mathcal{D}(A_k)$ for every $k \in \mathbb{N}$. It follows by the countable additivity of $\mathcal{D}$ that

$$\mathcal{D}(\cup_{k=1}^{\infty} A_k) = \mathcal{D}(\cup_{k=1}^{\infty} B_k) = \sum_{k=1}^{\infty} \mathcal{D}(B_k) \leqslant \sum_{k=1}^{\infty} \mathcal{D}(A_k) \ .$$

■

1.2 Random Variables, discrete and continuous

So far we have asked questions like: does an event happen or not and what is the probability of the event. But what about different questions- such as "how much." For example, if we get a dollar if a coin flip turns head, we can ask how many dollars would we get after 3 rounds?

In order to answer this question, we should understand the sample space, which is $\Omega = \{HHH, HHT, HTH, THH, HTT, THT, TTH, TTT\}$. Then we would need to define a reward function that will quantify ones's profit for each outcome of the experiment ($\omega \in \Omega$): $M(HHH) = 3, M(HHT) = 2$. This quantifying function M is called a *random variable* (RV).

Definition 5 *Given a probability space* $(\Omega, \mathcal{D})$*, a real-valued* random variable (RV) *is a function* $X : \Omega \to \mathbb{R}$.

Remark 1 *Any function* $g(X)$ *of a random variable is also a random variable (for instance* X^2, $\operatorname{sgn}(X)$*, etc.).*

For a discrete Ω we define the ***probability mass function (PMF)*** of X (a *discrete RV*) as

$$\mathcal{D}\left(\{X=x\}\right)=\sum_{\omega:X(\omega)=x}\mathcal{D}\left(\omega\right)$$

Instead of writing $\mathcal{D}\left(\{X=x\}\right)$ we usually write $\mathcal{D}\left(x\right)$.

Example 2 *A fair coin is tossed twice. The probability space is given by* $\Omega=\{T,H\}^2$ *and* $\mathcal{D}\left(\omega\right)=\frac{1}{4}$ *for any* $\omega\in\Omega$. *Let* X *denote the number of heads obtained in each outcome* (ω). *The image of* X *is* $\mathcal{X}=\{0,1,2\}$ *and*

$$\mathcal{D}\left(x\right)=\begin{cases}\frac{1}{4} & x=0,2\\ \frac{1}{2} & x=1\\ 0 & else\end{cases}$$

For a continuous Ω - we say that X is a *continuous RV* if there exists $f(x)\geqslant 0$ so that we can write, for every $D\subset\mathbb{R}$

$$\mathcal{D}\left(X\in D\right)=\int_D f\left(x\right)dx$$

In particular, this means that for any $a,b\in\mathbb{R}$, where $a\leqslant b$:

$$\mathcal{D}\left(X\in[a,b]\right)=\int_a^b f\left(x\right)dx$$

In this course we assume that $f(x)$ is a regular function and therefore the probability mass function for a single point is 0: $\mathcal{D}\left(X=a\right)=\int_a^a f\left(x\right)dx=0$ for any $a\in\mathbb{R}$.

The function f is called the ***probability density function*** **(PDF)** of X. Note that -

- $f\left(x\right)\geqslant 0$
- $\int_{-\infty}^{\infty}f\left(x\right)dx=1$
- $f\left(x\right)$ is a probability *density*, not a probability. For example, it could be that $f\left(x\right)>1$.

1.3 Mean and Variance

Definition 6 *The* ***expected value*** *of a random variable* X *is*

$$\mathbb{E}\left[X\right]=\sum_{x\in\mathcal{X}}x\mathcal{D}\left(x\right)\qquad or\qquad\mathbb{E}\left[X\right]=\int_{-\infty}^{\infty}xf\left(x\right)dx$$

Properties:

- $\mathbb{E}\left[g\left(X\right)\right]=\sum g\left(x\right)\mathcal{D}\left(x\right)$,
- $\mathbb{E}\left[c\right]=c$
- Linear: $\mathbb{E}\left[aX+Y\right]=a\mathbb{E}\left[X\right]+\mathbb{E}\left[Y\right]$
- If X and Y are independent then $\mathbb{E}\left[XY\right]=\mathbb{E}\left[X\right]\mathbb{E}\left[Y\right]$ (explanation: $\mathbb{E}\left[XY\right]=\sum\mathcal{D}(X=x,Y=y)x\cdot y=\sum_{x,y}\mathcal{D}(x)\mathcal{D}(y)xy=\mathbb{E}\left[X\right]\mathbb{E}\left[Y\right]$)

Definition 7 *Let X be a RV with $\mathbb{E}[X] = \mu$, then the **variance** of X is*

$$V(X) = \mathbb{E}\left[(X-\mu)^2\right]$$

*The **standard deviation** is defined as $\sigma = \sqrt{V(X)}$.*

Properties of the variance:

- $V(X) \geqslant 0$, $V(X) = 0 \iff X$ is a constant.
- $V(X) = \mathbb{E}\left[X^2\right] - (\mathbb{E}[X])^2$
- $V(aX+b) = a^2 V(X)$
- $V(X+Y) = \mathbb{E}\left[(X+Y-\mathbb{E}(X)-\mathbb{E}(Y))^2\right] = V(X)+V(Y)+2\overbrace{\mathbb{E}\left[(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])\right]}^{\mathrm{Cov}(X,Y)}$
- If $X_1, \ldots, X_n$ are independent then $V(\sum X_i) = \sum V(X_i)$

Definition 8 *The **covariance** of X and Y is*

$$\mathrm{Cov}(X,Y) = \mathbb{E}\left[(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])\right] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$$

Properties:

- $\mathrm{Cov}(X,Y) = \mathrm{Cov}(Y,X)$
- $\mathrm{Cov}(aX+b, cY+d) = a \cdot c \cdot \mathrm{Cov}(X,Y)$
- $\mathrm{Cov}(X,X) = V(X)$

Exercise 3 *Let Z be a Bernoulli random variable. Calculate $Var[Z]$.*

Solution: $Var[Z] = \mathbb{E}[Z^2] - (\mathbb{E}[Z])^2 = p - p^2 = p(1-p)$.

Example 3 *Find the mean and variance of $X \sim U([a,b])$, where $U[a,b]$ is the uniform distribution on $[a,b]$.*

$$\begin{aligned}
\mathbb{E}[X] &= \frac{1}{b-a}\int_a^b x dx = \frac{1}{2(b-a)}\left(b^2-a^2\right) = \frac{b+a}{2} \\
\mathbb{E}\left[X^2\right] &= \frac{1}{b-a}\int_a^b x^2 dx = \frac{1}{3(b-a)}\left(b^3-a^3\right) = \frac{b^2+ab+a^2}{3} \\
V(X) &= \frac{b^2+ab+a^2}{3} - \frac{(b+a)^2}{4} = \frac{(b-a)^2}{12}
\end{aligned}$$

1.4 A little Statistics: Mean and variance estimation

This is a course in *Statistical* Machine Learning. So let's start with a little bit of statistics. Some of you may have never seen statistics before. When we study *Probability* we assume a known distribution and calculate probability of events of interest. When we we study *Statistics* we turn the question on its head: using samples from an unknown probability distribution, we try to *estimate* or *test* properties of of the unknown distribution.

Take the mean and variance of a probability distribution for example. The mean gives us a taste of around which point the data is distributed. The variance on the other hand gives you a taste of the concentration of the "spread" of the distribution around its mean. Low variance usually means that the distribution is highly concentrated around the mean, while high variance means that the distribution is less concentrated there.
In a probability course, given that we know the distribution, we learned how to calculate its mean and variance. Now we would like to ask a different question. We are given *samples* that were drawn from an unkown distribution. Can we *estimate* the distribution's mean and variance using only those samples?

Formally, let X be a random variable, let p_X be its (unknown) distribution, and let $X_1, \ldots, X_n \overset{i.i.d}{\sim} p_X$. Here, i.i.d denotes *independent and identically distributed* random variables. We are given *data* samples: $x_1, \ldots, x_n$, where x_i is a *number* - the sample obtained by sampling X_i. In statistics, $\mathbb{E}[X]$ is called the *population* mean and $V(X)$ is called the population variance. So, what is a "reasonable" **estimator** for $\mathbb{E}[X]$ and $V(X)$ based only on those n samples?

Throughout the course we are going to use the following estimators (commonly denoted with the "hat" symbol $\hat{\circ}$):

- Sample mean (an estimator for the population mean)

$$\hat{\mu}_X = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- Sample variance (an estimator for population variance)

$$\hat{\sigma}_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu}_X)^2$$

Note the $\frac{1}{n-1}$ factor. Why not use $\hat{\Sigma}_X^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu}_X)^2$ as the variance estimator? *Answer*: calculate the expected value of these estimators and you will find (perhaps surprisingly) that $\mathbb{E}[\hat{\sigma}_X^2] = V(X)$ while $\mathbb{E}[\hat{\Sigma}_X^2] = \frac{n-1}{n}V(X)$. So, in expectation, $\hat{\sigma}_X^2$ gets the population variance exactly. When an estimator for some parameter of the unknown distribution gets the desired parameter exactly *in expectation*, we call that estimator *unbiased* for that parameter.

We invite you to read some additional material about the sample variance and the population variance **here** and **here** if you want to understand the reason for the difference in the formula.

2 High dimensional distributions

Back to probability. Now, instead of random variables that take real (scalar) value, we will consider a more general case in which they take vector values in $\mathbb{R}^d$, where $d \in \mathbb{N}$.

2.1 Basic definitions

Definition 9 (Random vector) *A finite collection of random variables, denoted $X_1, \ldots, X_n$ (defined on a common probability space $(\Omega, \mathcal{D})$) is a random (column) vector: $\boldsymbol{X} = (X_1, \ldots, X_n)^T$*

Definition 10 (Joint distribution) *Given random variables $X_1, \ldots, X_n$, that are defined on a probability space, the joint probability distribution for $X_1, \ldots, X_n$ is a probability distribution that gives the probability that each of $X_1, \ldots, X_n$ falls in any particular range (for continuous RVs) or discrete set (for discrete RVs) of values specified for that variable.*

Definition 11 (Joint PDF of a random vector) *Two random variables X and Y are jointly continuous if there exists a nonnegative function $f_{XY} : \mathbb{R}^2 \to \mathbb{R}$, such that, for any set $A \in \mathbb{R}^2$, we have*

$$\mathcal{D}\left((X, Y) \in A\right) = \int_A f_{XY}(x, y)dxdy$$

The function $f_{XY}(x, y)$ is called the joint probability density function (JPDF) *of X and Y. This can of course be generalized to any dimension.*

Remark 2 (PDF for a random vector) *Given $\boldsymbol{X} = (X_1, \ldots, X_n)^T$, each of the scalar random variables $X_1, \ldots, X_n$ can be characterized by its PDF. However, unless the scalar RV's are mutually independent, the PDF of each coordinate of a random vector does not completely describe the probabilistic behaviour of the whole vector. For instance, X_1 and X_2 could have the same PDF, but still the behavior of $\boldsymbol{X} = (X_1, X_2)$ and $\tilde{\boldsymbol{X}} = (X_1, X_1)$ can be drastically different. For example, consider $X_1 \sim U([-a, a])$ and $X_2 = -X_1$.*

2.2 Normal Distribution

As an example of random vectors, we now consider specific family of distributions- the Multivariate Normal (or Multivariate Gaussian) Distributions. The normal distribution is very useful for many reasons. One reason is the Central Limit Theorem (CLT). You have likely seen the one-dimensional ("univariate") version of the CLT in your probability course. In one of its forms, under some conditions, the CLT states that sample mean of random i.i.d variables (when properly scaled) converges in distribution to the normal distribution. There is a multivariate version of the CLT, which is one reason why you want to be familiar with the multivariate normal distribution. Another reason we like the (multivariate) normal distribution is that many results and methods (such as propagation of uncertainty and least squares parameter fitting) can be derived analytically in explicit form when the relevant variables are (multivariate) normally distributed.

Definition 12 (Univariate Normal Distribution (namely, in dimension $d = 1$)) *A random variable x has a normal distribution with expectation μ and variance σ^2 if it has a PDF of the form:*

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

In this case we write: $x \sim \mathcal{N}(\mu, \sigma^2)$

Definition 13 (Multivariate Normal Distribution) *random vector x has a multivariate normal distribution with expectation μ and covariance matrix (see definition below) Σ if it has a joint PDF of the form:*

$$f(x) = \frac{1}{\sqrt{(2\pi)^n |\Sigma|}} \exp\left\{ -\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2} \right\}$$

In this case we write: $x \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

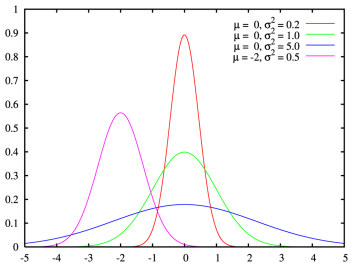


Figure 1: $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

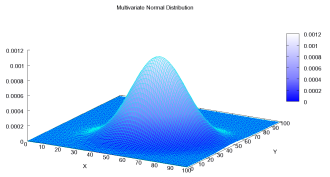


Figure 2: 2D case of $f(x) = \frac{1}{\sqrt{(2\pi)^2|\Sigma|}} \exp\left\{ -\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2} \right\}$

Observe that definition 13 is generalization of 12, i.e. when $d = 1$ both definitions are same.

In some cases we do not care about some of the variables in the distribution. for example, suppose we have joint probability function of the variables "height" and "weight" of some community, but we're only interested in one of them. This lead as to the following definition:

Definition 14 (Marginal Distribution) *Marginal distribution of a subset of a collection of random variables is the probability distribution of the variables contained in the subset. It gives the probabilities of various values of the variables in the subset without reference to the values of the other variables. It can be written as:*

$$p_X(x) = \int_y p_{X,Y}(x,y) \mathrm{d}y$$

Exercise 4 (Marginal of the 1st coord. of a 2D normal RV with diagonal covariance) *Let* $x \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ *where* $\mu = (\mu_1, \mu_2), \boldsymbol{\Sigma} = \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}$.

Find the PDF of the marginal distribution of x_1.

Solution:

Observe that we have:

$$
\begin{aligned}
&\frac{1}{\sqrt{(2\pi)^n \left|\Sigma\right|}} \exp\left\{-\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2}\right\} \\
&= \frac{1}{\sqrt{(2\pi)^2 \left|\begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix}\right|}} \exp\left\{-\frac{(x-\mu)^T \begin{pmatrix} \sigma_1^{-2} & 0 \\ 0 & \sigma_2^{-2} \end{pmatrix} (x-\mu)}{2}\right\} \\
&= \frac{1}{\sqrt{(2\pi)^2 \sigma_1^2 \sigma_2^2}} \exp\left\{-\frac{(x-\mu)^T \begin{pmatrix} \sigma_1^{-2} & 0 \\ 0 & \sigma_2^{-2} \end{pmatrix} (x-\mu)}{2}\right\} \\
&= \frac{1}{\sqrt{(2\pi)^2 \sigma_1^2 \sigma_2^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2 - \frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right\} \\
&= \frac{1}{\sqrt{(2\pi)\sigma_1^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right\} \frac{1}{\sqrt{(2\pi)\sigma_2^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right\}
\end{aligned}
$$

Now, since: $\int_{-\infty}^{\infty} \frac{1}{\sqrt{(2\pi)\sigma_2^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right\} \mathrm{d}x_2 = 1$, we get:

$$
\begin{aligned}
&\int_{-\infty}^{\infty} \frac{1}{\sqrt{(2\pi)^n \left|\Sigma\right|}} \exp\left\{-\frac{(x-\mu)^T \Sigma^{-1} (x-\mu)}{2}\right\} \, dx_2 \\
&= \frac{1}{\sqrt{(2\pi)\sigma_1^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right\} \int_{-\infty}^{\infty} \frac{1}{\sqrt{(2\pi)\sigma_2^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_2-\mu_2}{\sigma_2}\right)^2\right\} \, dx_2 \\
&= \frac{1}{\sqrt{(2\pi)\sigma_1^2}} \exp\left\{-\frac{1}{2}\left(\frac{x_1-\mu_1}{\sigma_1}\right)^2\right\}
\end{aligned}
$$

and by definition 12 we get: $x_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$

2.3 Covariance Matrix

Definition 15 (Covariance Matrix) *Recall the population variance $V[X]$ of a scalar (univariate) random variable X. What is the proper generalization for the case of a random vector? Consider a d-dimensional random vector $X = (X_1, X_2, ..., X_d)^T$. We define the* Covariance Matrix Σ *as the $d \times d$ matrix whose (i, j) entry is the covariance $\Sigma_{ij} = \mathrm{Cov}(X_i, X_j)$*

- *In other words:*

$$
\Sigma = \begin{pmatrix} \mathbb{E}\left[(X_1 - \mathbb{E}\left[X_1\right])\left(X_1 - \mathbb{E}\left[X_1\right]\right)\right] & \dots & \mathbb{E}\left[(X_1 - \mathbb{E}\left[X_1\right])\left(X_d - \mathbb{E}\left[X_d\right]\right)\right] \\ \vdots & \ddots & \vdots \\ \mathbb{E}\left[(X_d - \mathbb{E}\left[X_d\right])\left(X_1 - \mathbb{E}\left[X_1\right]\right)\right] & \dots & \mathbb{E}\left[(X_d - \mathbb{E}\left[X_d\right])\left(X_d - \mathbb{E}\left[X_d\right]\right)\right] \end{pmatrix}
$$

- *Recall that we can use matrix notations for random vectors. In matrix notation,*

$$
\Sigma = \mathbb{E}[(X - \mathbb{E}\left[X\right])(X - \mathbb{E}\left[X\right])^T]
$$

- *The diagonal elements of Σ are $V[X_i]$. Σ is symmetric positive semidefinite.*

2.4 Estimating the Covariance Matrix

Back to statistics, let us consider estimation of the covariance matrix. In this context, Σ is called the *population covariance matrix*.

Before we get started, let's take a quick look at the difference between covariance and variance. Variance measures the variation of a single random variable (like the height of a person in a population), whereas covariance is a measure of how much two random variables vary together.

Let's start with the case $d = 2$. Consider a two-dimensional random vector $X = (X_1, X_2)$, where (say) X_1 is the height of a person and X_2 is the weight.

Taking a sample of m people from the population, we denote the data as follows. Height samples are $x_{1,1}, .., x_{1,m}$ and the corresponding weight samples are $x_{2,1}, .., x_{2,m}$. Note that we can write the data as a matrix $X \in \mathbb{R}^{d \times m}$ where m is the *sample size* and d is the dimension (number of random variables). In our case, $d = 2$.

Using the definition of sample variance seen above, the sample variances of the heights and weights are given by:

$$\hat{\sigma}^2_{X_i} = \frac{1}{m-1} \sum_{j=1}^{m} (x_{i,j} - \hat{\mu}_i)^2, i = 1, 2$$

where $\hat{\mu}_i$ is the sample mean of the random variable X_i.

Let us define the sample covariance $\hat{\sigma}(X_1, X_2)$ of two random variables X_1 and X_2 similarly by:

$$\hat{\sigma}(X_1, X_2) = \frac{1}{m-1} \sum_{j=1}^{m} (x_{1,j} - \hat{\mu}_1)(x_{2,j} - \hat{\mu}_2)$$

By definition, the sample variance equals the sample covariance of the RV with itself: $\hat{\sigma}^2_{X_i} = \hat{\sigma}(X_i, X_i)$

We can now define the sample covariance matrix $\hat{\Sigma}$. (In statistics, the estimator of a parameter c is often denoted by $\hat{c}$. This is known as the "hat notation". Here, the notation hints that the sample covariance matrix $\hat{\Sigma}$ will be used as an estimator for the population covariance matrix Σ).

Consider a d-dimensional random vector $X = (X_1, \ldots, X_d)$. Suppose that we have m i.i.d samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$ of X. Here, the sample $\mathbf{x}_i$ is a vector in $\mathbb{R}^d$, and the boldface notation reminds us that $\mathbf{x}_i$ is a vector, not a scalar.

The sample covariance matrix of $\mathbf{x}_1, \ldots, \mathbf{x}_d$ is defined to be the square d-by-d matrix whose elements are given by $\hat{\Sigma}_{i,j} = \hat{\sigma}(X_i, X_j)$. The sample covariance matrix is symmetric since $\hat{\sigma}(X_j, X_i) = \hat{\sigma}(X_i, X_j)$. The diagonal entries of the sample covariance matrix are the variances and the other entries are the covariances.

In matrix notation, the sample covariance matrix can be equivalently defined as

$$\hat{C} = \frac{1}{m-1} \sum_{i=1}^{m} (\mathbf{x}_i - \bar{X})(\mathbf{x}_i - \bar{X})^T$$

where $\bar{X} \in \mathbb{R}^{d \times 1}$ is a column vector of the sample means (in the $d = 2$ case, $\bar{X}^T = (\hat{\mu}_1, \hat{\mu}_2)$). Depending on the context, $\bar{X}$ may also denote the means matrix: $\bar{X} \in \mathbb{R}^{d \times m}$ with elements $\bar{X}_{i,j} = \hat{\mu}_i$. Following from the above equation, the covariance matrix can be computed for a data set with zero mean with $\hat{C} = \frac{XX^T}{m-1}$ by using the symmetric matrix XX^T.

Exercise 5 *let* $X^T = \begin{pmatrix} 150 & 45 \\ 170 & 74 \\ 184 & 79 \end{pmatrix}$ *be samples of height and weight of 3 different people where* $X_{1,i}$ *is the height and* $X_{2,i}$ *is the weight of the i'th person. Calculate the covariance matrix of the sample.*

Solution: First of all we will make the data centered: We have: $\hat{\mu}_1 = 168, \hat{\mu}_2 = 66$, so

$$X^T_{centered} = X^T - \bar{X}^T = \begin{pmatrix} 150 & 45 \\ 170 & 74 \\ 184 & 79 \end{pmatrix} - \begin{pmatrix} 168 & 66 \\ 168 & 66 \\ 168 & 66 \end{pmatrix} = \begin{pmatrix} -18 & -21 \\ 2 & 8 \\ 16 & 13 \end{pmatrix}$$

and:

$$\hat{C} = \frac{X_{centered} X^T_{centered}}{3-1} = \begin{pmatrix} 292 & 301 \\ 301 & 337 \end{pmatrix}$$

2.5 Linear Transformations of the Data Set

Although we will now focus on the two-dimensional case, the results in this section can be easily generalized to higher dimensional data. The covariance matrix for two dimensions, $d = 2$, is

$$C = \begin{pmatrix} \sigma(X_1, X_1) & \sigma(X_1, X_2) \\ \sigma(X_2, X_1) & \sigma(X_2, X_2) \end{pmatrix}$$

We want to show how linear transformations affect the data set and as a result, the covariance matrix. First we will take random points with zero mean values $\bar{X}_1 = \bar{X}_2 = 0$ and equal variances $\sigma^2_{X_1} = \sigma^2_{X_2} = \sigma^2$

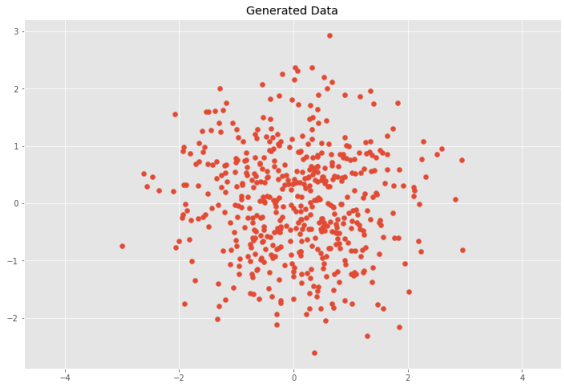


Figure 3: Uncorrelated Variables

This would mean that X_1 and X_2 are uncorrelated [3] and the covariance matrix C is:

$$C = \begin{pmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{pmatrix}$$

Next, we will look at how transformations affect our data and the covariance matrix C. We will transform our data with the following scaling matrix.

[3] Uncorrelated does not imply independent. See, e.g., `https://en.wikipedia.org/wiki/Normally_distributed_and_uncorrelated_does_not_imply_independent`

$$S = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix}$$

where the transformation simply scales the X_1 and X_2 components by multiplying them by s_1 and s_2 respectively. Now the covariance matrix of our transformed data set will be:

$$\frac{SX(SX)^T}{n-1} = S\frac{XX^T}{n-1}S^T = SCS^T = \begin{pmatrix} (s_1\sigma)^2 & 0 \\ 0 & (s_2\sigma)^2 \end{pmatrix}$$

.

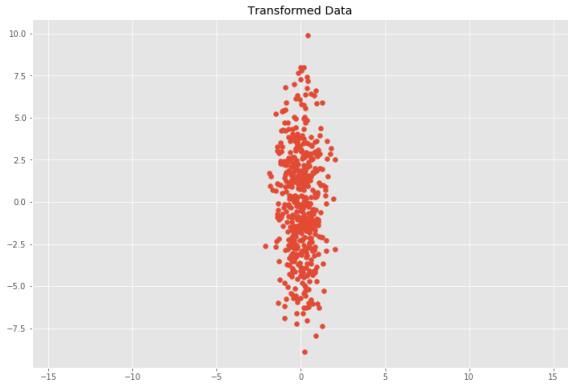


Figure 4: Uncorrelated Scaled Variables

Now we will apply a linear transformation in the form of a transformation matrix T to the data set which will be composed of a two dimensional rotation matrix R and the previous scaling matrix S as follows:

$$T = RS$$

where the rotation matrix R is given by:

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

where θ is the rotation angle. The transformed data is then given by RSX and the covariance matrix is

$$\frac{RSX(RSX)^T}{n-1} = RS\frac{XX^T}{n-1}S^TR^T = RSCS^TR^T = R\begin{pmatrix} (s_1\sigma)^2 & 0 \\ 0 & (s_2\sigma)^2 \end{pmatrix}R^T$$

$$= \sigma^2 \times \begin{pmatrix} s_1^2\cos^2\theta + s_2^2\sin^2\theta & \sin\theta\cos\theta(s_1^2 - s_2^2) \\ \sin\theta\cos\theta(s_1^2 - s_2^2) & s_1^2\sin^2\theta + s_2^2\cos^2\theta \end{pmatrix}$$

Note that without applying an *asymmetric* scaling, rotation alone would not be enough to create correlations since for $s_1 = s_2$, the off-diagonal elements vanish.

As we can see now, the x, y coordinate of each variable are not uncorrelated anymore. For example, if we know that the x value of the sample is high, it is more likely that the y value are low, and vice versa. Note, that R is orthogonal, thus we got the eigenvalue decomposition (EVD) of the covariance matrix. Later in the course we are going to use some of the properties of the covariance matrix and its eigenvalue decomposition in the context of dimensionality reduction.

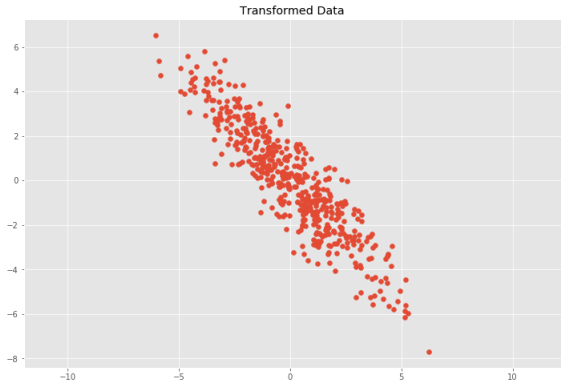


Figure 5: Correlated Scaled Variables

3 Probability Inequalities

3.1 Motivation and Background

[4] Probability inequalities are very useful in analyzing random processes. In particular, they are used to analyze machine learning algorithms. As we will see in class, in many learning tasks, we wish to estimate the expectation of some random variable (e.g. the generalization error of a hypothesis) using the average of independently and identically distributed (i.i.d) random variables (e.g. the empirical loss of the same hypothesis). The law of large numbers states that the empirical average converges to the expected value. Probability inequalities provides us a means by which we can tell how good our estimates are when the sample is finite.

To be more concrete, consider the following task. Suppose that there is a bag containing red and blue balls. We would like to estimate the fraction p of red balls in the bag. However, we are only allowed to sample randomly from the bag with replacement. The straightforward strategy is to draw n samples and predict

$$\hat{p} = \frac{\#\text{ of red balls}}{n} .$$

It is clear that $\mathbb{E}[\hat{p}] = p$. However, we might err due to the fact that we get only limited information. Thus, we are interested in an estimation with a certain additive error, ϵ. Another obstacle is that since our process is random, there is a probability that our sample doesn't represent the distribution "well". Thus, we will further refine our objective. We will be interested to guarantee with high probability that our estimator has an additive error at most ϵ.

More generally, concentration inequalities deal with the following fundamental question: Given an accuracy parameter $\epsilon > 0$ and a confidence parameter $\delta \in (0, 1)$, how many samples are needed to guarantee that with probability of at least $1 - \delta$, our estimate is within an additive error of at most ϵ. The corresponding notion in supervised learning is called *probably approximately correct (PAC) learnability*. That is, we aim at constructing a learner whose estimations are probably (with probability at least $1 - \delta$) approximately (within additive error of at most ϵ) correct.

The exact formulation of the PAC framework will be given in the following lectures. Here, we will consider the well-known challenge of estimating the bias of a given coin. As we shall see, this moderate challenge reveals many of the ingredients of the PAC framework.

[4] This section based on lecture notes by Michal Moshkovitz and Alon Gonen

3.2 Recap

We next recall Markov's and Chebyshev's inequalities and then apply both inequalities to derive concentration inequalities for averages.

Theorem 1 ***Markov's inequality:*** *For a nonnegative random variable X (i.e. $Im(X) \subseteq \mathbb{R}_{\geqslant 0}$) with finite mean and a positive scalar t,*

$$\mathbb{P}[X \geqslant t] \leqslant \frac{\mathbb{E}[X]}{t} \ .$$

Proof Let $f(x)$ be the density function of x. Since X is non-negative, we obtain

$$\begin{aligned}
\mathbb{E}[X] &= \int_{x \in \mathbb{R}} f(x) x \, dx \\
&= \int_{x=0}^{t} f(x) x \, dx + \int_{x=t}^{\infty} f(x) x \, dx \\
&\geqslant \int_{x=t}^{\infty} f(x) x \, dx \\
&\geqslant \int_{x=t}^{\infty} f(x) t \, dx \\
&= t \cdot \int_{x=t}^{\infty} f(x) \, dx \\
&= t \cdot \mathbb{P}[X \geqslant t] \ .
\end{aligned}$$

The desired inequality is obtained by dividing both sides by t. ■

Corollary 1 *Let X be a nonnegative random variable with finite mean. Denote its distribution by $\mathcal{D}$. Let $X_1, \ldots, X_m$ be m i.i.d. copies of X, i.e., $X_1, \ldots, X_m$ are i.i.d. random variables which are distributed according to $\mathcal{D}$. Denote their average by $\overline{X} = \frac{1}{m} \sum_{i=1}^{m} X_i$. Finally, let $t > 0$. Then*

$$\mathbb{P}[\overline{X} \geqslant t] \leqslant \frac{\mathbb{E}[X]}{t} \ .$$

Proof By the linearity of the expectation,

$$\begin{aligned}
\mathbb{E}\left[\bar{X}\right] &= \mathbb{E}\left[\frac{1}{m} \sum_{i=1}^{m} X_i\right] \\
&= \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}\left[X_i\right] \\
&= \frac{1}{m} \sum_{i=1}^{m} \mathbb{E}\left[X\right] \\
&= \mathbb{E}\left[X\right] \ .
\end{aligned}$$

Applying Markov's inequality, we obtain the desired bound. ■

Applying Markov inequality to the random variable $(X - \mathbb{E}[X])^2$, we obtain Chebyshev's inequality.

Theorem 2 ***Chebyshev's inequality:*** *For a random variable X with finite mean and variance, and for every $t > 0$,*

$$\mathbb{P}[|X - \mathbb{E}[X]| \geqslant t] \leqslant \frac{Var[X]}{t^2}.$$

Proof Simply observe that $\mathbb{P}[|X - \mathbb{E}[X]| \geqslant t] = \mathbb{P}[(X - \mathbb{E}[X])^2 \geqslant t^2]$ and apply Markov's inequality. ■

Unlike the expectation, the variance of the average of i.i.d. random variables is not equal to the variance of the original random variable. Concretely, for a random variable X and m i.i.d. copies of X, denoted $X_1, \ldots, X_m$, we have

$$\begin{aligned}
\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m} X_i\right] &= \frac{1}{m^2}\mathrm{Var}\left[\sum_{i=1}^{m} X_i\right] \\
&= \frac{1}{m^2}\sum_{i=1}^{m}\mathrm{Var}[X_i] \\
&= \frac{1}{m^2}\sum_{i=1}^{m}\mathrm{Var}[X] \\
&= \frac{1}{m}\mathrm{Var}[X]\ . \qquad (1)
\end{aligned}$$

In particular, the variance of the average tends to zero when m tends to infinity. This leads to a much better concentration inequality.

Corollary 2 *Let X be a nonnegative random variable with finite variance and let $X_1, \ldots, X_m$ be m i.i.d. copies of X. Denote their average by $\overline{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$. Finally, let $t > 0$. Then*

$$\mathbb{P}[|\overline{X} - \mathbb{E}[\overline{X}]| \geqslant t] = \mathbb{P}[|\overline{X} - \mathbb{E}[X]| \geqslant t] \leqslant \frac{\mathrm{Var}[X]}{mt^2}\ .$$

For a positive integer k, the k-th *moment* of a random variable X is defined by $\mathbb{E}[X^k]$. As we just saw, Markov's inequality exploits only information about the first moment, while Chebyshev's inequality exploits both the first and the second moments. Comparing the bounds in Corollary 1 and Corollary 2, we see that using both the first and the second moments, we obtain better concentration inequalities than using only the first. We shall see soon that more generally, the more moments we use, the better the bounds we get.

3.3 Coin Prediction

Let us formulate the problem of estimating the bias of a coin (in short, coin prediction)[5]. Formally, a coin is a Bernoulli random variable Z with a bias $p - 1/2$ ('bias' is defined here as the *deviation* from $p = 1/2$, where p is the probability to obtain "heads"). The random variable Z is simply a function from Ω to $\{0, 1\}$. Let $Z_1, \ldots, Z_m$ be a sequence of independent and identically distributed (i.i.d.) random variables according to the distribution $\mathcal{D}_Z$ (that is, each of the Z_i has the same distribution as Z, and all of the Z_i are mutually independent). Denote the random sequence (a.k.a. sample) $(Z_1, \ldots, Z_m)$ by S, where $|S| = m$. The distribution which is associated with S is denoted by $\mathcal{D}_Z^m$ (or simply by $\mathcal{D}^m$).

[5] We take this opportunity to recall some basic notions from probability theory

The input of a *learning algorithm* A for the task of coin prediction consists of a random sequence S which is drawn according to $\mathcal{D}^m$. Based on this input, the algorithm has to predict the value of p. We denote this prediction, (that is, the output of the algorithm) by $A(S)$ or simply by $\hat{p}$. Clearly, the drawn sequence can not fully describe the distribution. Hence, we introduce an *accuracy* parameter $\epsilon > 0$, and allow $\hat{p}$ to deviate from p by at most ϵ. Furthermore, there is always a chance that the drawn sequence would be highly non-representative, so it is impossible to obtain a guarantee that holds with absolute certainty. Hence, we introduce a *confidence* parameter $\delta > 0$, and allow that the event $B := \{s : |\hat{p} - p| > \epsilon\}$ occur with probability at most δ. More formally:

Definition 16 *A mistake-bound learning algorithm A for the task of (batch) coin prediction is a function $\hat{p}(S)$ where $S \in \bigcup_{m=1}^{\infty}\{0,1\}^m$ and $\hat{p}(S) \in [0,1]$ which satisfies the following condition:*

For any $0 \leqslant \epsilon, \delta \leqslant 1$, there exists a non-negative integer $m_A(\epsilon,\delta)$ such that, if a sequence S of m numbers, where $m \geqslant m_A$, is generated according to $\mathcal{D}_p^m$, (that is, $\mathcal{D}^m$ with parameter p, which is added temporarily here as a subscript but omitted later on) then the probability that the algorithm's output $\hat{p}(S)$ will not be in $[p-\epsilon, p+\epsilon]$ is bounded by:

$$\mathcal{D}_p^m\left[|\hat{p} - p| > \epsilon\right] < \delta,$$

for any *$0 \leqslant p \leqslant 1$**,** and if a sequence S of m numbers, where $m < m_A$, is generated,* ***there exist a*** *p, with $0 \leqslant p \leqslant 1$ such that:*

$$\mathcal{D}_p^m\left[|\hat{p} - p| > \epsilon\right] > \delta$$

The function $m_A(\epsilon,\delta)$: $[0,1] \times [0,1] \to \mathbb{N}$ is called the sample complexity *of the learning algorithm A.*

The first condition means that, no matter what p is, it is enough to draw m_A samples (i.e., to flip the coin m_A times) in order to know p, with a certainty of $1-\delta$ and an accuracy of $\pm\epsilon$. The second conditions means that, at least for some values of p, drawing $m_A(\epsilon,\delta) - 1$ samples would *not* be enough for that.

Note about the confidence requirement: In the above definition the confidence requirement must hold *independently of the procedure by which the coins are provided*, that is, independently of the way p is chosen. It does not matter if, for example, the coin is randomly drawn from a pile of coins where most coins are approximately fair (most of their p's are close to 1/2) or where most are faked in a specific way, (their p's are concentrated around, say, $\frac{1}{4}$), or where all p's are equally probable.

Another way to think of the sample complexity is the following. Consider all possible sequences, S, of m 0's and 1's. For each value of p, each of these sequences has a different probability to occur. Each S can be labeled as 'good' ('typical') or 'bad'('atypical'), 'good' means that $m \cdot \hat{p}(S)$ is between $m(p-\epsilon)$ and $m(p+\epsilon)$ and 'bad' otherwise (Note that, even for the same coin, if S is good with respect to algorithm $A_1(S)$ it may still be bad with respect to another algorithm $A_2(S)$, if $\hat{p}_1(S) \neq \hat{p}_2(S)$.) Given a coin, let us denote the smallest m for which the total sum of the probabilities of all the good sequences is at least $1-\delta$, by $m_A^{(p)}(\epsilon,\delta)$. We can think of $m_A^{(p)}$ as the *coin-specific* sample complexity. The algorithm sample complexity $m_A(\epsilon,\delta)$ is the maximal number among the coin-specific sample complexities, that is: $m_A(\epsilon,\delta) = \max_p(m_A^{(p)}(\epsilon,\delta))$.

Given an i.i.d. random sample $S = (z_1,\ldots,z_m)$, a reasonable estimate of p is the empirical proportion of heads (ones), namely $\hat{p} = \frac{1}{m}\sum_{i=1}^{m} z_m$. The first basic property of this average is

that its expected value equals p. This follows by the linearity of expectation as follows:

$$\mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} Z_i\right] = \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[Z_i\right] = \frac{1}{m}\cdot m \cdot p = p \ .$$

We say in this case that $\hat{p}$ forms an *unbiased estimate* of p. What can we say about the quality of $\hat{p}$ as an estimate of p? In other words, how tightly $\hat{p}$ is concentrated around its expectation? To answer this question, we can use concentration inequalities.

First attempt: Markov's inequality:

For a direct application of Markov's inequality we take $|\hat{p} - p|$ as our RV (note that $\hat{p} - p$ is not a positive RV and therefore cannot be used in Markov's inequality). We then need to calculate $\frac{1}{\epsilon}\mathbb{E}\left[|\hat{p} - p|\right]$ and extract its dependence on the sample size m. Since we are going to get much better bounds below, here we only state the result (and leave the calculation as a challange exercise):

$$\mathcal{D}^m\left[|\hat{p} - p| \geqslant \epsilon\right] \leqslant \frac{1}{\sqrt{4m\epsilon^2}}$$

This implies that the sample complexity of coin prediction is bounded above by $m(\epsilon, \delta) \leqslant \left\lceil \frac{1}{4\epsilon^2} \cdot \frac{1}{\delta^2} \right\rceil$. (You can verify by substituting $m = \frac{1}{4\epsilon^2} \cdot \frac{1}{\delta^2}$ in the bound)

Exercise 6 (Markov's inequality) *Prove that:* $\mathcal{D}^m\left[|\hat{p} - p| \geqslant \epsilon\right] \leqslant \frac{1}{\sqrt{4m\epsilon^2}}$.

- Hint: Show that for any RV h with $\mathbb{E}\left(h\right) = 0$:

  $$\mathbb{E}\left(|h|\right) \leqslant \sqrt{Var(h)}$$

  and then apply this result to the RHS of Markov's inequality.

- Use the above to obtain a bound over the sample complexity and compare it to the one obtained from the Chebyshev's inequality.

Second attempt: Chebyshev's inequality: The variance of a Bernoulli random variable is $p(1-p) \leqslant 1/4$. Applying Corollary 2, we obtain that

$$\mathcal{D}^m[|\hat{p} - p| \geqslant \epsilon] = \mathcal{D}^m[|\hat{p} - \mathbb{E}[\hat{p}]| \geqslant \epsilon] \leqslant \frac{Var[Z]}{m\epsilon^2} \leqslant \frac{1}{4m\epsilon^2}$$

The bound obtained using Chebyshev's inequality tends to zero as $\frac{1}{m}$ while the one obtained from Markov's inequality tends to zero as $\frac{1}{\sqrt{m}}$. This fact enables us to obtain a better bound for the sample complexity:

Corollary 3 *The sample complexity of coin prediction is bounded above by* $m(\epsilon, \delta) \leqslant \left\lceil \frac{1}{4\epsilon^2} \cdot \frac{1}{\delta} \right\rceil$. *(You can verify by substituting* $m = \frac{1}{4\epsilon^2} \cdot \frac{1}{\delta}$ *in the bound)*

A natural question which arises is whether the obtained bound is optimal (tight). There is a good reason to suspect that this bound is not optimal; When we applied Markov's or Chebyshev's inequality, we did not exploit the fact that our RV's are bounded (between 0 and 1). Indeed, in the next section we prove a much better bound [6]. For this end, we will introduce Hoeffding's inequality for the average of independent and bounded random variables. This inequality will give us exponential (instead of polynomial!) convergence in terms of the sample size m.

[6] Which is tight, although we will not prove that

3.3.1 Hoeffding's Inequality

Theorem 3 (Hoeffding's inequality) *Let $X_1, \dots X_m$ be independent and bounded random variables with $a_i \leqslant X_i \leqslant b_i$. Let $\bar{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$. Then,*

$$\mathbb{P}[|\bar{X} - \mathbb{E}[\bar{X}]| \geqslant \epsilon] \leqslant 2\exp\left(\frac{-2m^2\epsilon^2}{\sum_{i=1}^{m}(b_i - a_i)^2}\right)$$

Corollary 4 *Let $X_1, \dots X_m$ be a sequence of i.i.d. (independent **and** identically distributed) and bounded with $a \leqslant X_i \leqslant b$ (same a, b for all X_i's). Let $\bar{X} = \frac{1}{m}\sum_{i=1}^{m} X_i$ and let $\mu = \mathbb{E}[X_i]$. Then,*

$$\mathbb{P}[|\bar{X} - \mu| \geqslant \epsilon] \leqslant 2\exp\left(\frac{-2m\epsilon^2}{(b-a)^2}\right)$$

3.3.2 Coin Prediction Revisited

Lets apply Hoeffding's inequality for the case of coin prediction which was discussed above. For a sample of size m, we obtain

$$\mathcal{D}^m[|\hat{p} - p| \geqslant \epsilon] \leqslant 2\exp(-2m\epsilon^2) \ .$$

Indeed, we achieve an exponential improvement (in terms of δ).

Corollary 5 *The sample complexity of coin prediction is bounded above by $m(\epsilon, \delta) \leqslant \left\lceil \frac{1}{2\epsilon^2} \cdot \log(\frac{2}{\delta}) \right\rceil$. (You can verify by substituting $m = \frac{1}{2\epsilon^2} \cdot \log(\frac{2}{\delta})$ in the bound)*

Lecture 2: The Linear Model

4 The Linear Model, Noiseless Case

4.1 Problem: Customer Lifetime Value prediction

We are working for an online store and would like to predict the "Customer Lifetime Value", that is, the total future net profit that an online customer will provide. In order to do so, we collect d features on each customer (age, income, amount spent up till now, etc). So our sample domain $\mathcal{X} = \mathbb{R}^d$, is a customer feature space, and the lifetime values form our label set $\mathcal{Y} = \mathbb{R}$. This is called a **Regression** problem. Each customer is represented by a vector of d numbers, $\mathbf{x} \in \mathbb{R}^d$ and each example, also sometimes called sample, or data point, is represented by $(\mathbf{x}, y)$, where y is the Customer Lifetime Value of that customer.

4.2 The training data matrix

We are trying to guess ("predict") future y's from many past examples of $(\mathbf{x}_i, y_i)$, $i = 1..m$ which we will call (the **training set**). So the training data we have is m **samples**, $\mathbf{x}_1, \ldots \mathbf{x}_m$ (each containing d **features**) and their m **labels**, $y_1, \ldots, y_m$. Let's organize the training data in a matrix form by defining a *column vector* $\mathbf{y} = (y_1, \ldots, y_m)^\top$ and by stacking the samples $\mathbf{x}_1, \ldots, \mathbf{x}_m$ to get the d-by-m matrix (Note: soon we will add a line of 1's at the top of our matrix and obtain a $d+1$-by-m matrix, but we will keep using the same notation for that enlarged matrix).

$$X = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_m \\ | & | & & | \end{bmatrix}$$

4.3 Setup

We assume that there is a deterministic (unknown to us) function[7] $: \mathcal{X} \to \mathcal{Y}$ that is *underlying* the relation between each $\mathbf{x}_i$ and y_i. *If* we have a perfect noiseless data, without bugs in our database, without errors in the measurement of the features, etc, then $y_i = f(\mathbf{x}_i)$ for every $i = 1..m$. Even if we have a noisy data, f is still underlying the relation between each $\mathbf{x}_i$ and y_i but it is obscured by the noise, so $y_i = f(\mathbf{x}_i) + z_i$ where z_i is a random variable which represents the noise. We will start with the noiseless case ($\mathbf{z} = 0$)) and then turn on the noise.

Our goal is to learn f from many examples of $(\mathbf{x}_i, f(\mathbf{x}_i))$, $i = 1..m$ (the training set), so we can make good guesses about future y's. Using an algorithm we then guess ("predict") f. We call this guess the **prediction rule** and we will denote it by $\hat{f}$ or h_s, where S denotes the above set of m examples (our input data) with which we feed our algorithm.

Even before we receive any additional data, we can measure the quality of our prediction, $\hat{f}$, over the training set, by defining a **Loss function**:

$$\sum_{i=1}^{m} L(f(\mathbf{x}_i)\,,\, \hat{f}(\mathbf{x}_i)), \quad i = 1..m.$$

where $L(f(\mathbf{x}_i)\,,\, \hat{f}(\mathbf{x}_i))$ is some function that we will have to choose later in order to quantify the "cost" (or penalty) for predicting $\hat{f}(\mathbf{x}_i)$ instead of $f(\mathbf{x}_i)$.

[7] "Deterministic", that is, if we insert the same $\mathbf{x}$ into its argument, we always get the same $f(\mathbf{x})$

4.4 The Linear Hypothesis Class

We must[8] choose a set of functions, namely our **Hypothesis Class**, $\mathcal{H}$, to pick our guess from. We will soon assume that $f \in \mathcal{H}$, however, even if not, we may still be able to find a "good enough" approximation for f among the functions in $\mathcal{H}$.

Suppose that, examining our data, we discover that customer lifetime value y seems to be more or less *linear* in the features $\mathbf{x}$, so we choose the **Linear Hypothesis Class**:

$$\mathcal{H}_{reg} = \left\{ h: \quad h(x_1, \ldots, x_d) = w_0 + \sum_{i=1}^{d} x_i w_i, \quad w_0, w_1, \ldots, w_d \in \mathbb{R} \right\}$$

Each function h in the class is characterized by the **weights** $w_1, \ldots, w_d$ it gives to each of the d features [9] and an **intercept** w_0. Learning the function f means learning the weights and the intercept from the training data.

To simplify notation, for any sample $(x_1, \ldots, x_d) \in \mathbb{R}^d$ we add a zero-th coordinate and define $\mathbf{x} = (1, x_1, \ldots, x_d)^\top$. We also write $\mathbf{w} = (w_0, w_1, \ldots w_d)^\top$. In this notation, any function h in the hypothesis class is of the form $h(\mathbf{x}) = \langle \mathbf{w}, \mathbf{x} \rangle = \mathbf{x}^\top \mathbf{w}$

4.4.1 The Realizable Case

We are looking for some $\mathbf{w} \in \mathbb{R}^{d+1}$ for which $y_i = \mathbf{x}_i^\top \mathbf{w}$, where $i = 1, \ldots, m$. Or, in matrix form:

$$X^\top \mathbf{w} = \mathbf{y}$$

which is a set of m equations that we want to solve for $\mathbf{w}$ (which is a set of $d+1$ variables).

If there is a (at least one) solution, it means that f is in $\mathcal{H}_{reg}$. This is called **The Realizable Case** or **The Realizability Assumption**

4.4.2 The Non-Realizable Case

The assumption that the labels y are *exactly* linear in the samples $\mathbf{x}$ is not realistic. The data may be almost linear but never exactly linear, as shown in the typical figure below.

In such cases, f is *not* in $\mathcal{H}_{reg}$. This is called **The Non-Realizable Case** in which we must settle for finding $\hat{f} \in \mathcal{H}_{reg}$ which is "good enough" for our purposes.

4.4.3 Geometric interpretation

Let's denote by $\varphi_i \in \mathbb{R}^m$ the i-th row of X, so

$$X = \begin{bmatrix} — & \varphi_0 & — \\ — & \varphi_1 & — \\ & \vdots & \\ — & \varphi_d & — \end{bmatrix}$$

[8] Recall that a choice of hypothesis class, $\mathcal{H}$, is always required due to "No Free Lunch".

[9] Note the difference between x_i and $\mathbf{x}_i$. We use normal fonts for the d customer features, x_i, while **bold fonts** for the feature vector, $\mathbf{x}$. Thus, for example, $\mathbf{x} = (x_1, x_2, .., x_d)^\top$, while $\mathbf{x_i} = (x_{1,i}, x_{2,i}, .., x_{d,i})^\top$

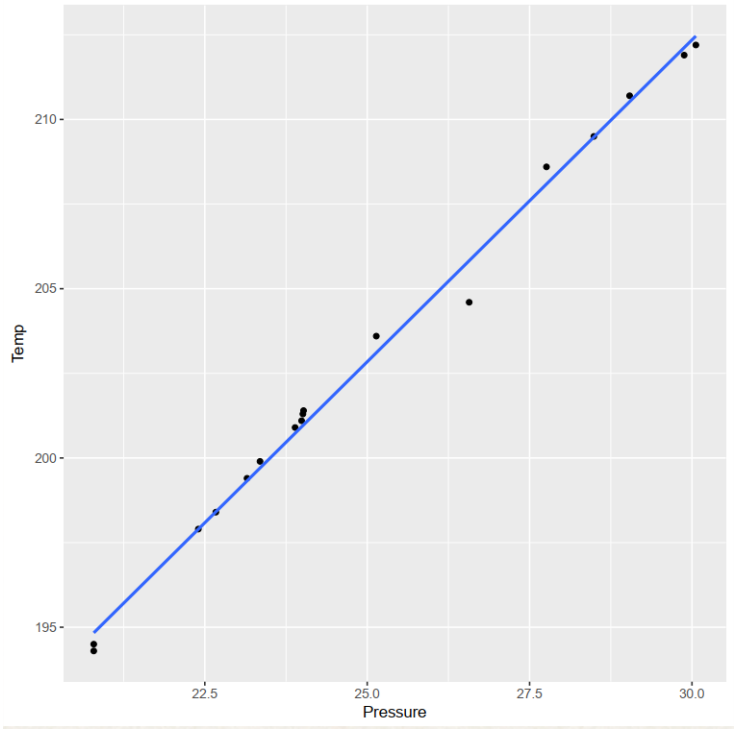


Figure 6: In realistic cases, y is never exactly linear in the samples $\mathbf{x}$

In the realizable case, the system of equations $X^\top \mathbf{w} = \mathbf{y}$ has either a unique solution (when $dim(Ker(X^\top)) = 0$) or an infinite number of solutions ($dim(Ker(X^\top)) \neq 0$). In both case $\mathbf{y} \in Im(X^\top)$. If $dim(Ker(X^\top)) = 0$ (unique solution) then the $\varphi_0, \varphi_1, \ldots, \varphi_d$ are linearly independent.

In the non-realizable case $X^\top \mathbf{w} = \mathbf{y}$ has no solution so $\mathbf{y} \notin Im(X^\top)$.

4.5 Designing the Learning Algorithm

4.5.1 The Loss Function

To design the algorithm with which we will learn (predict) f we need to choose a loss function with respect to which we will pick the "most probable" / "most likely" / "best fitting" $\hat{f}(x)$ in the linear hypothesis class $\mathcal{H}_{reg}$. We could for example choose the **Absolute Value Loss**

$$L(y, \hat{f}(x)) = |y - \hat{f}(x)|$$

or the **Squared Loss**

$$L(y, \hat{f}(x)) = (y - \hat{f}(x))^2$$

There are good reasons to use each one. In this lecture *we use the squared loss* since the related math is much simpler.

4.5.2 Empirical Risk Minimization

Since on new data we will evaluate performance by $(y - \hat{f}(x))^2$, it makes sense to choose $\hat{f}$ that minimizes that same loss L *on the training data we already have*. For a given prediction rule

$\hat{f} \in \mathcal{H}$, the quantity

$$\sum_{i=1}^{m} L(y_i, \hat{f}(x_i))$$

where $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$ is our training data, is called the **empirical risk**. In our case the empirical risk of the linear function $\hat{f}(\mathbf{x}_i) = \mathbf{x}_i^\top \mathbf{w}$ is, based on the above choice of a square-loss function, given by

$$\sum_{i=1}^{m} (y_i - \mathbf{x}_i^\top \mathbf{w})^2 = \left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2 = (\mathbf{y} - X^\top \mathbf{w})^\top (\mathbf{y} - X^\top \mathbf{w})$$

4.5.3 Least Squares

Minimizing the empirical risk in our case means minimizing the sum of squares of the deviations of the labels from a linear function. In other words we choose the linear function in $\mathcal{H}_{reg}$ that is closest to the labels in terms of the squared error distance. The deviation $y_i - \mathbf{x_i}^\top \mathbf{w}$ is called the i-th **residual** and the total empirical risk in our case is called **Residual Sum of Squares** (or **RSS**) and is denoted by

$$RSS(\mathbf{w}) = \left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$$

So to learn the linear function by **Empirical Risk Minimization**, we want to find $\operatorname{argmin} RSS(\mathbf{w}) = \operatorname{argmin} \left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$. The empirical risk function $\left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$ is a *quadratic form* in $\mathbf{w}$, i.e., it is a polynomial in the w_i's with terms all of degree two. It is therefore a smooth function of $\mathbf{w}$ with a minimum (or minima). If X has full rank, then $\left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$ has a unique minimum (as in the figure below), which we need to find in order to find $\hat{f}$.

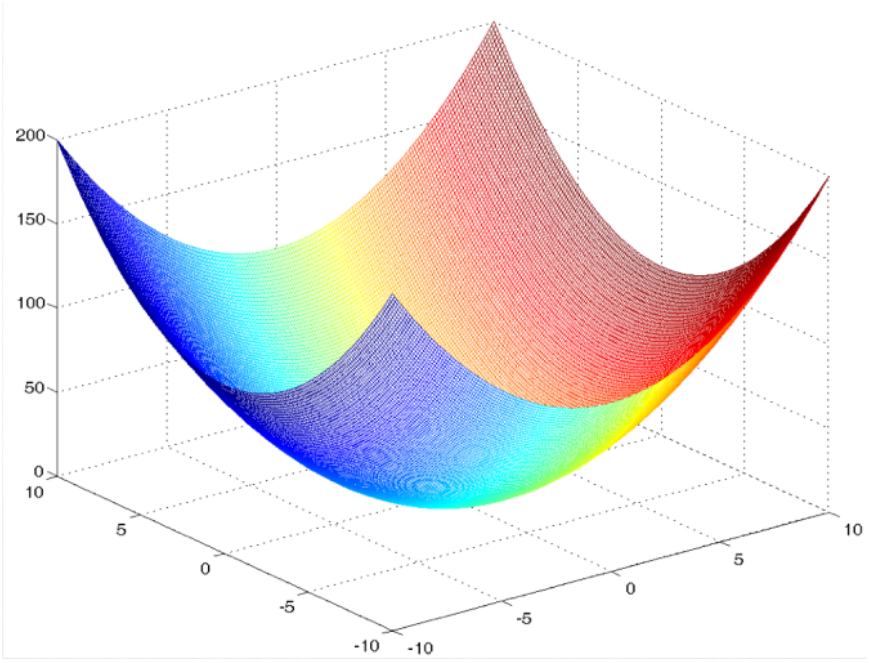


Figure 7: $RSS(\mathbf{w}) = \left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$ in the case of X with a full rank - a unique minimum

A *necessary* condition for $\mathbf{w}$ to be a minimizer of the function $\left\|\mathbf{y} - X^\top \mathbf{w}\right\|^2$ is that all its

partial derivative vanish at $\mathbf{w}$. Recalling the definition of the inner product, this condition can be written as:

$$\frac{\partial}{\partial w_j} RSS(\mathbf{w}) = -2 \sum_{i=1}^{m} (\mathbf{x}_i)_j \cdot (y_i - \mathbf{x}_i^\top \mathbf{w}) = 0$$

for all $j = 0 \ldots, d$, where $(x_i)_j$ is the j-th entry of $\mathbf{x}_i$, that is, it is the $x_{j,i}$ element of the matrix X. We can write all these $d + 1$ equations in matrix form as a linear system:

$$\nabla RSS(\mathbf{w}) = -2X(\mathbf{y} - X^\top \mathbf{w}) = 0$$

4.6 The Normal Equations

So a necessary condition for $\mathbf{w}$ to be a minimizer or $RSS(\mathbf{w})$ is that $\mathbf{w}$ is a solution to the linear system

$$X(\mathbf{y} - X^\top \mathbf{w}) = 0$$

or, equivalently

$$X\mathbf{y} = XX^\top \mathbf{w}$$

These are the famous **Normal Equations**, a name that will become clearer later on.

4.6.1 Solving the Normal Equations

Let's *assume more samples than features, $m \geqslant d + 1$.* [10],

Let us also *assume that the linear system $X^\top \mathbf{w} = \mathbf{y}$ has at least one solution*[11] which implies that it either has 1 (unique) or ∞ solutions. As we know from linear algebra, it has a unique solution if and only if $dim(Ker(X^\top)) = 0$, because if $Ker(X^\top)$ contains any vector $\mathbf{w}' \neq \mathbf{0}$, then any vector of the form $\mathbf{w} + a\mathbf{w}'$ would also be a solution. Homework: Show that $dim(Ker(X^\top)) = 0$ if and only if $dim(Ker(X \cdot X^\top)) = 0$ and conclude that there exists a unique solution if and only if $X \cdot X^\top$ is invertible.

We first consider the case of a *unique solution.* We would like to solve our set of equations but in order to have a procedure that is generalizable for cases in which our assumptions do not hold, and for computational reasons, we prefer to invert matrices instead of using Gauss elimination. $X^\top$ is not a square matrix and therefore is not invertible, but $X \cdot X^\top$ is square and because we assume a unique solution, it is also invertible we can right-away solve for $\mathbf{w}$:

$$\mathbf{w} \quad = \quad (XX^\top)^{-1} X\mathbf{y}$$

So we can learn $\mathbf{w}$, i.e., learn the linear function, simply by using the above formula.

[10] This is a reasonable assumption in our case if we have, say, hundreds of customer features available to us (age, sex, income, previous item purchased and so on), but thousands (or millions) of customers in our samples. There can be other cases: Suppose, for example, that the customer features contain, among other things, all previously purchased items. Now consider Amazon in Israel. The total customer pool is of the order of millions, while Amazon offers (many) millions of products. So in such a case, it may very well be that $m < d$.

[11] This assumption might seem somewhat contradictory with the first one ($m \geqslant d + 1$): The more data we collect, m (which is the number of linear equations 'zipped' in $XX^\top \mathbf{w} = X\mathbf{y}$) grows, so the d numbers which are represented by $\mathbf{w}$ need to satisfy more and more equations. However, many of these equations are *dependent* and do not add new information to what we learn about the function f. In reality, the assumption that there is a (deterministic) function f that we can learn and then 'know everything' is never valid (e.g., due to f being too complicated and not in the hypothesis class, or due to random noise in the labels $\mathbf{y}$) and therefore, the larger m is, the better $\hat{f}$ we can find.

Consider now the second case, where $dim(Ker(X \cdot X^\top)) \neq 0$ and therefore $X \cdot X^\top$ is not invertible. We know that there is an infinite number of solutions, since $dim(Ker(X \cdot X^\top)) \neq 0$. but we can not invert $X \cdot X^\top$. On the other hand, we do not want to use Gauss elimination because it is too *numerically unstable* (more on that, soon). Even if there is an infinite number of solutions, we still need a way to find at least one. Instead of the standard method (Gauss elimination) we all learned in linear algebra courses, in the next section we will apply an easily-generalizable and *computationally-stable* technique, using the **Singular Value Decomposition** of matrices, or **SVD**.

4.7 Singular Value Decomposition

The **Singular Value Decomposition** or **SVD** is based on the following facts:

- *Any* real matrix, and in particular, our $d+1$-by-m matrix X, can be written as

  $$X = U \cdot \Sigma \cdot V^\top$$

  where U is an orthonormal $d+1$-by-$d+1$ matrix, Σ is a $d+1$-by-m diagonal matrix, and V is an orthonormal m-by-m matrix. For every matrix there can be many SVD's. In particular, there is an SVD where the diagonal elements of Σ, denoted by $\Sigma_{i,i} \equiv \sigma_i$, satisfy $\sigma_1 \geqslant \sigma_2 \geqslant \ldots, \geqslant \sigma_{d+1} \geqslant 0$. The σ_i's are called the **singular values** of X.

- The columns of U are eigenvectors of $XX^\top$. These columns are called the **left singular vectors** of X.

- The columns of V are eigenvectors of $X^\top X$. These are called the **right singular vectors** of X.

- $\sigma_1^2, \ldots, \sigma_{d+1}^2$ are the shared eigenvalues of both $X^\top X$ and $XX^\top$.

- The columns of U and V are ordered such that the i-th column corresponds to the eigenvalue σ_i^2. This is important because the unique choice of ordering of the singular values: $\sigma_1 \geqslant \ldots, \geqslant \sigma_{d+1} \geqslant 0$ in Σ puts constraints on the choice of U and V.

We stress that *any* real matrix has an SVD decomposition (or a lot of them) - whether or not it is diagonalizable, or even square! This is one of the reasons why SVD is *Incredibly useful* in learning and data science.

The SVD has many useful properties one of which is the following: $dim(Ker(X^\top)) = 0$ if and only if $\sigma_{d+1} > 0$ (that is, all the singular values are nonzero). So one can tell $dim(Ker)$ and $dim(Im)$ of a matrix simply by looking at its singular values.

Another useful property of SVD is (for proof, see homework):
Let $\Sigma^\dagger$ be a m-by-$d+1$ diagonal matrix with diagonal elements

$$\Sigma^\dagger_{i,i} = \begin{cases} 1/\sigma_i & \sigma_i > 0 \\ 0 & \sigma_i = 0 \end{cases}$$

We now define

$$\hat{\mathbf{w}} = U(\Sigma^\dagger)^\top V^\top \mathbf{y} = (X^\top)^\dagger \mathbf{y}$$

Theorem: $\hat{\mathbf{w}}$ is *always* a solution to the system $XX^\top \mathbf{w} = X\mathbf{y}$ independently of whether or not $XX^\top$ is invertible: If $XX^\top$ is invertible (the case of a unique solution), then $\hat{\mathbf{w}} = (XX^\top)^{-1}X\mathbf{y}$ so we recover the unique solution. Even if $XX^\top$ is not invertible, that is, even if there are ∞ solutions to the system of equations $XX^\top \mathbf{w} = X\mathbf{y}$, $\hat{\mathbf{w}}$ is still one of them, in fact a one with a *minimal norm*:

$$\|\hat{\mathbf{w}}\| = min\left\{\|\mathbf{w}\| \,:\, XX^\top \mathbf{w} = X\mathbf{y}\right\}$$

To conclude, SVD is useful for learning $\mathbf{w}$ because, among other things, it works regardless of whether there are 1 or ∞ solutions. Another major reason why SVD is useful for learning is because it is *numerically stable*, as explained below.

4.7.1 Making the SVD solution numerically stable

Computers don't calculate over $\mathbb{R}$, they use bits and more specifically they use floating-point arithmetics with very finite precision. A lot of non-trivial knowledge and care are required in order to understand how learning algorithms are actually implemented in software. *Any* learning algorithm comes down to a lot of calculus (e.g gradients) and linear algebra (e.g. inverses) implemented in software. You should care *deeply* about how algorithms are implemented and when they break numerically, as in the following example.

Sometimes $XX^\top$ is formally invertible but *close to singular*. This happens if columns of $X^\top$ are *almost* co-linear or if one column of $X^\top$ is *almost* spanned by other columns. In this case some singular values of X will be nonzero, but very small. When this happens, Gauss elimination may yield wildly incorrect results. Also, because of double-precision arithmetics, $1/\sigma_i$ will not be precise. The practical solution in such cases is to choose a **machine precision threshold**, ε and let

$$\Sigma^{\dagger,\varepsilon}_{i,i} = \begin{cases} 1/\sigma_i & \sigma_i > \varepsilon \\ 0 & \sigma_i \leqslant \varepsilon \end{cases}$$

4.8 How many samples do we need to learn a linear function?

The system $X^\top \mathbf{w} = \mathbf{y}$ has m equations in $d+1$ variables. If $m < d+1$ we have no hope of learning f - we don't have enough data. Our hypothesis class is a $d+1$-dimensional linear space: the larger d, the more complicated the hypothesis space. So the larger d, the more samples we need to learn it.

4.9 Summary - Noiseless Case

Before presenting a concrete example, let us summarize what we discussed so far about learning linear functions. When the training data is $(\mathbf{x}, f(\mathbf{x}))$ with f a linear function, we know how to learn f. We need at least $d+1$ training samples. We learn by solving the system $XX^\top \mathbf{w} = X\mathbf{y}$. This can be done in a numerically stable way using the SVD, regardless of whether or not the system has a unique solution or not. If there is a unique solution, SVD recovers it and in that case, this is the same solution we would get by simply inverting $XX^\top$ and multiplying both sides of the normal equations by that inverse. If there are infinitely many solutions, SVD will recover one which has a minimal norm.

5 The Linear Model - Noisy Case

As we mentioned before, the assumption that the labels y are *exactly* linear in the samples $\mathbf{x}$ is not realistic. The deviation from linearity can happen simply because the real deterministic function, f, is only approximately linear, but also because that f is obscured by random inaccuracies in the values of the samples, the labels or both. This is the case we would like to discuss.

5.1 Data Generation Model With Noise

So far we assumed that training data is created by a deterministic process[12], that is, there exits a function $f(x)$ where the training data is of the form

$$\Big(x_i\,,\, f(x_i)\Big) \qquad i = 1, \dots m$$

.

We now would like to consider a more general and much more realistic case where the training data is of the form

$$\Big(x_i\,,\, f(x_i) + z_i\Big) \qquad i = 1, \dots m$$

with $z_1, \dots z_m \overset{\text{iid}}{\sim} (0, \sigma^2)$, meaning that the noise generated by i.i.d random variables from some distribution with mean 0 and variance σ^2. So from now on, our treatment of learning will be probabilistic: Due to the random noise, even for identical $\mathbf{x}$'s, we may end up with different y's. In order to be able to learn we therefore adapt what we did so far in the deterministic case to the probabilistic (noisy) one.

Let us still assume the linear hypothesis class and that we have enough data to learn, namely $m \geqslant d + 1$. So, as before, there is an unknown $\mathbf{w}$ such that for every sample vector $\mathbf{x}_i$ in our data (that is, every column in our data matrix X):

$$y_i = \mathbf{x}_i^\top \mathbf{w} + z_i$$

Denoting the noise vector $\mathbf{z} = (z_1, \dots, z_m)^\top$ we have in matrix notation

$$\mathbf{y} = X^\top \mathbf{w} + \mathbf{z}$$

.

Note that the vector $\mathbf{y}$ is no longer necessarily [13] in $Im(X^\top)$ in which case the system $\mathbf{y} = X^\top \mathbf{w}$ has no solutions.

Let's use the square loss function as before, and learn by empirical risk minimization:

$$\mathbf{w}_S := \text{argmin}_{\mathbf{w}} \left\| y - X^\top \mathbf{w} \right\|$$

This means learning $h_S \in \mathcal{H}_{lin}$ by solving the normal equations. This makes a lot of sense. We have $\mathbf{y} \notin Im(X^\top)$ - because the noise "pushed" $\mathbf{y}$ out of $Im(X^\top)$. As we've seen, solving the normal equations is equivalent to projecting $\mathbf{y}$ back onto $Im(X^\top)$ (through finding a Least Squares approximation).

[12] We recall that by "Deterministic" we simply mean: if the same $\mathbf{x}$ appears again, it appears with the same $\mathbf{y}$.

[13] Unless, by an extreme accident, $\mathbf{z} \in Im(X^\top)$. Moreover, due to the noise term, for the *same* $\mathbf{x}$, one can have two *different* y's, which *guarantees* that there will be no solution

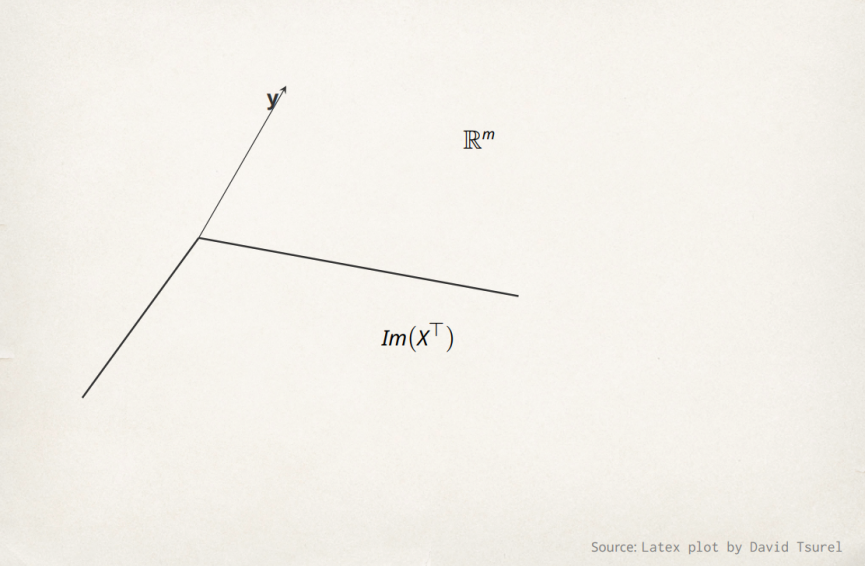


Figure 8: $\mathbf{y}$ no longer in $Im(X^\top)$

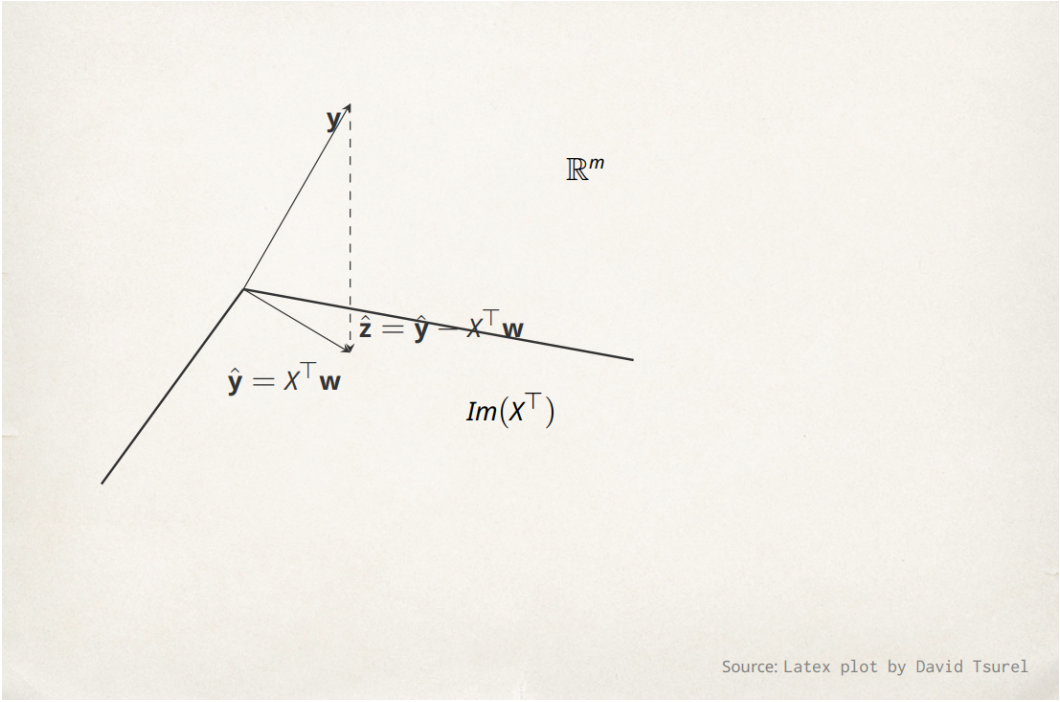


Figure 9: $\hat{y} \equiv X^\top \hat{w}$ is the projection of $\mathbf{y}$ onto $Im(X^\top)$

5.2 The Maximum Likelihood Principle

Assume further - for just a moment - that noise is Gaussian: $z_i \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2)$. This means that the i-th observation is independently distributed $y_i \sim \mathcal{N}(\mathbf{x}_i^\top \mathbf{w}, \sigma^2)$. Suppose we **knew** the weight vector $\mathbf{w}$, we could then ask the following question: The data matrix X is fixed. Given that we know $\mathbf{w}$, what is the probability to observe a responses vector $\mathbf{y}$?

Answer: The probability density is a product of Gaussian densities:

$$p(\mathbf{y}|\mathbf{w}) = \prod_{i=1}^{m} \left[\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\mathbf{x}_i^\top \mathbf{w} - y)^2}{2\sigma^2}} \right]$$

This is a question in probability: We know $\mathbf{w}$, what's the chance to observe $\mathbf{y}$? But we are actually interested here in the reverse question: we sampled $\mathbf{y}$, what's the most "likely" value of $\mathbf{w}$?

The answer is known as the **Maximum Likelihood** (ML) principle which suggests: let's choose $\mathbf{w}$ for which the probability density of getting the observed $\mathbf{y}$ is maximal. We write the **likelihood function** - now a function of $\mathbf{w}$, for fixed $\mathbf{y}$ the likelihood is

$$L(\mathbf{w} \,|\, \mathbf{y}) = \frac{1}{(2\pi\sigma^2)^{m/2}} \prod_{i=1}^{m} \left[e^{-\frac{(\mathbf{x}_i^\top \mathbf{w} - y)^2}{2\sigma^2}} \right]$$

The Maximum Likelihood Estimator (MLE) for $\mathbf{w}$ is thus

$$\hat{\mathbf{w}} := \text{argmax}_{\mathbf{w}} L(\mathbf{w} \,|\, \mathbf{y}) = \text{argmax}_{\mathbf{w}} \log L(\mathbf{w} \,|\, \mathbf{y}) = \text{argmin}_{\mathbf{w}} \sum_{i=1}^{m} (\mathbf{x}_i^\top \mathbf{w} - y_i)^2$$

This, by now, should look familiar: The MLE (assuming Gaussian noise) is just the Least Squares estimator we obtained by empirical risk minimization of square error loss.

5.3 Noise, Bias and Variance

Let us consider a concrete example of what we learnt so far.

5.3.1 Polynomial fitting

Consider the following special case called polynomial fitting[14]: We are given points $a_1, \ldots a_m \in \mathbb{R}$ and labels $y_1, \ldots, y_m$. We would like to learn a polynomial function $\mathbb{R} \to \mathbb{R}$ from the following Hypothesis class:

$$\mathcal{H}^d = \left\{ a \mapsto \sum_{k=0}^{d} w_k a^k \right\}$$

Denoting $\mathbf{x}_i = (1, a_i, a_i^2, \ldots, a_i^d)$, our hypothesis class is simply $\left\{ \mathbf{x} \mapsto \mathbf{x}^\top \mathbf{w} \,|\, \mathbf{w} \in \mathbb{R}^{d+1} \right\}$ The matrix X in this case is the **Vandermonde matrix** that you are probably familiar with from linear algebra courses. Since the a_i's are different from one another, it has full rank: $d+1$ if $d+1 \leqslant m$ and m if $d+1 > m$.

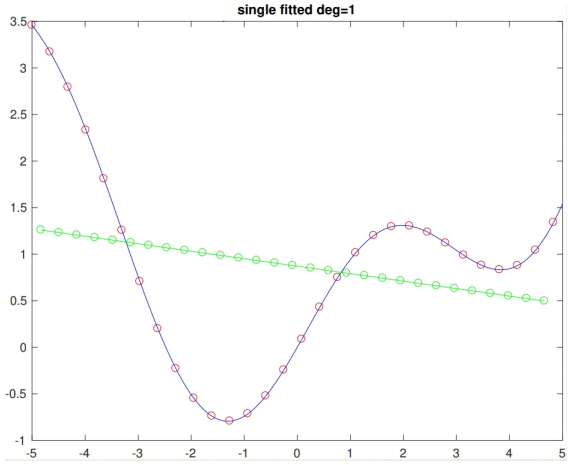
5.3.2 Polynomial fitting: no noise

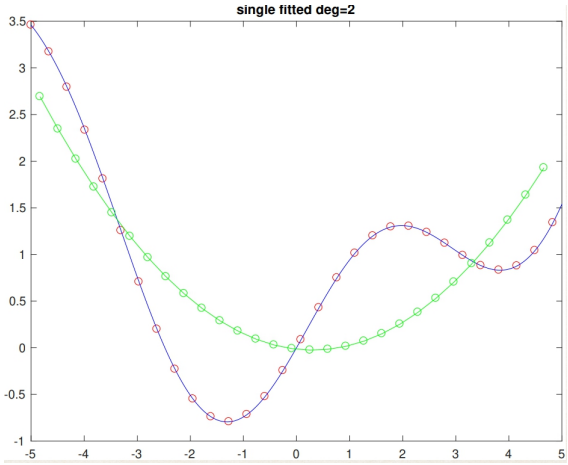
In the following figures we chose $a_1, \ldots, a_{30}$ (horizontal values of red circles). We then chose a "true" polynomial p (shown in blue curve) of degree 29 and calculated its values at these points - see the red circles. To fit a polynomial of degree d we used linear regression on the training data $\mathbf{x}_i = (1, a_i, a_i^2, \ldots, a_i^d)$ and $y_i = p(a_i)$, $i = 1, \ldots, 30$. The linear regression chose

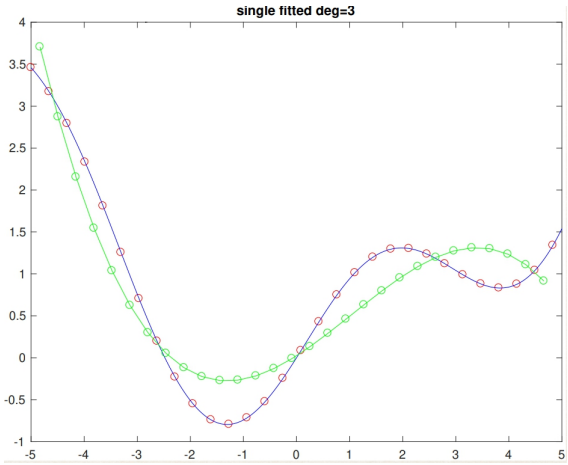
[14] In spite of the term "Polynomial fitting", and the form of $\mathcal{H}$, this model is in fact a specific case of the Linear Model. This is because we do *not* require the d features that we discussed in previous sections to be independent of one another

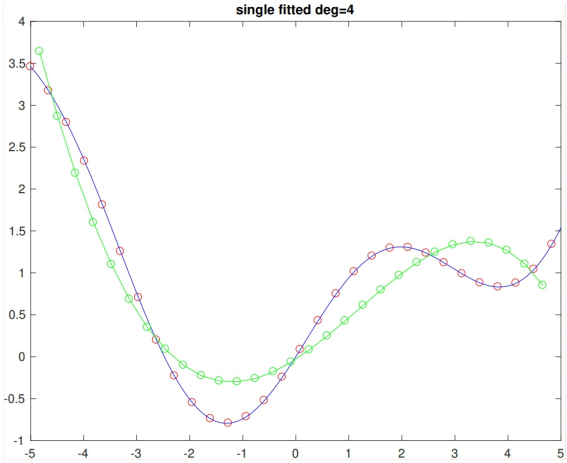
coefficients for polynomial $\hat{p}$ of degree d. The fitted polynomial is shown in the green curve. We evaluated the fitted polynomial $\hat{p}$ on the training data $a_1, \ldots, a_{30}$ (green circle).

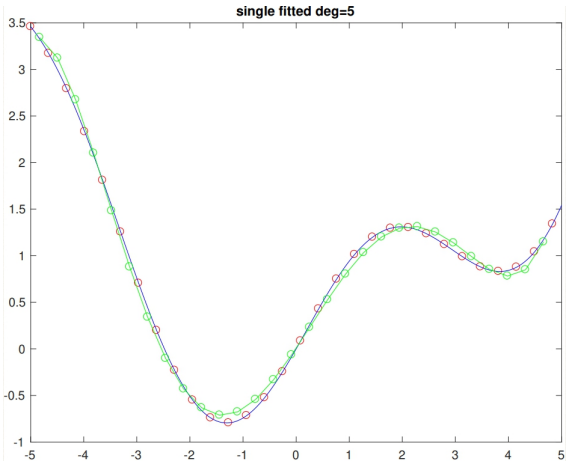
The hypothesis class in each slide consists of polynomials up to a given degree, indicated on the title. So the hypothesis class grows from slide to slide.

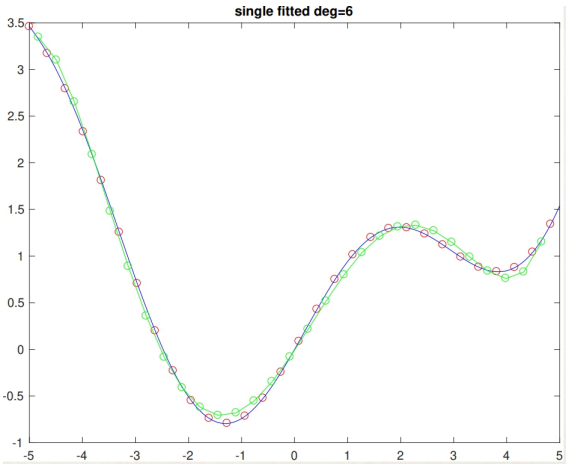

single fitted deg=1
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-5
-4
-3
-2
-1
0
1
2
3
4
5


single fitted deg=2
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-5
-4
-3
-2
-1
0
1
2
3
4
5


single fitted deg=3
4
3.5
3
2.5
2
1.5
1
0.5
0
-0.5
-1
-5
-4
-3
-2
-1
0
1
2
3
4
5

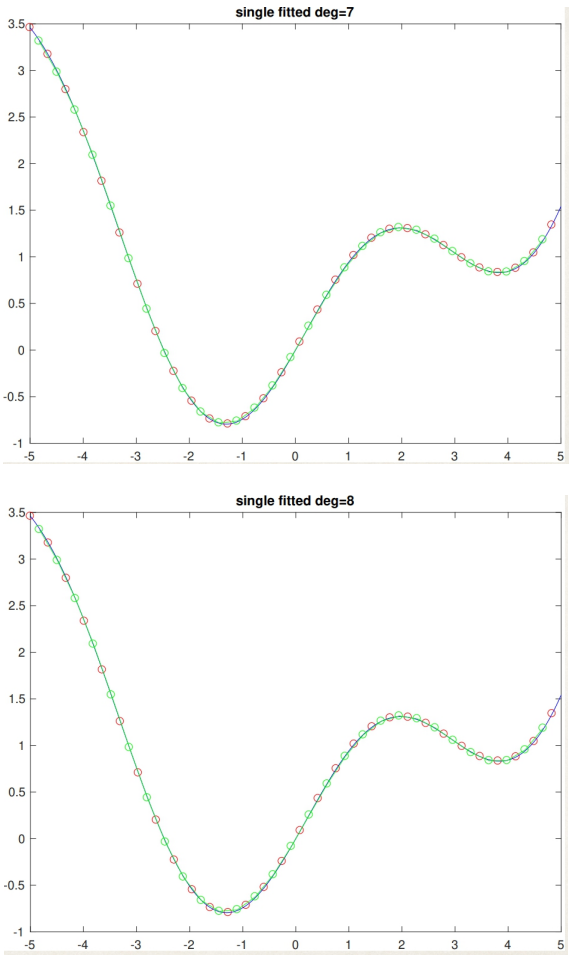






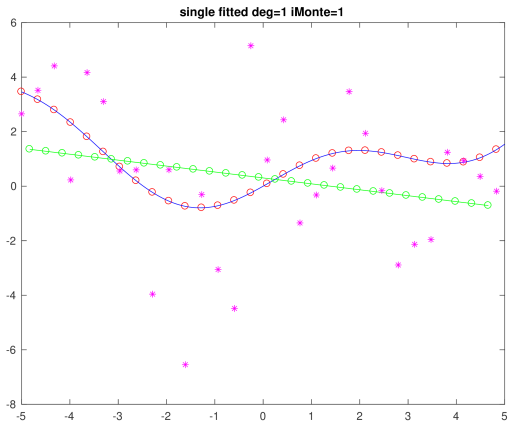
5.3.3 Bias

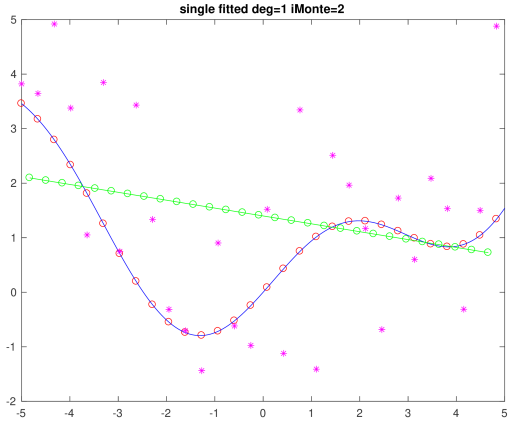
What can we learn from these figures? For small d, the hypothesis class $\mathcal{H}^d$ is too small. The true polynomial p of degree 29 cannot be described even by the "best approximation" to p from $\mathcal{H}^d$. Informally, **Bias** describes how well the "true" f can be approximated by our hypothesis class. Informally, the larger the hypothesis class, the smaller the bias.

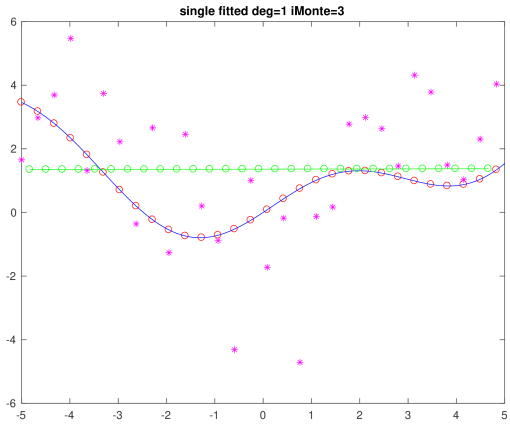


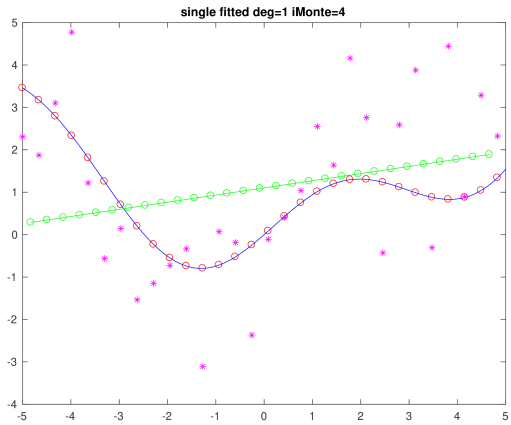
5.4 Polynomial fitting: with noise

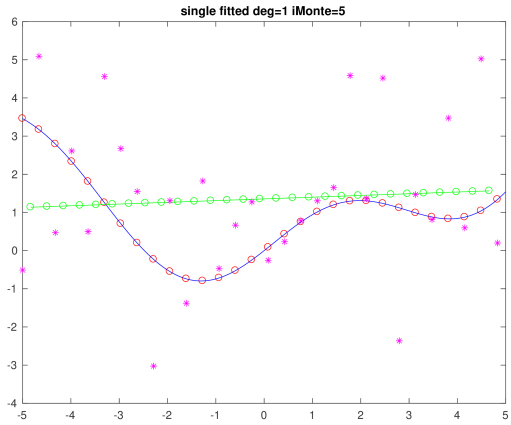
The next figures shows an evaluation of p on the red circles **and added i.i.d Gaussian noise** $z_1, \ldots, z_{30} \overset{\textbf{iid}}{\sim} \mathcal{N}(0, 4)$. The noisy values are shown in pink stars. To fit a polynomial of degree d we used linear regression with training data $\mathbf{x}_i = (1, a_i, a_i^2, \ldots, a_i^d)$ and $y_i = p(a_i) + z_i$, $i = 1, \ldots, 30$. The linear regression chose coefficients for polynomial $\hat{p}$ of degree d. The fitted polynomial is shown in green curve. We evaluated the fitted polynomial $\hat{p}$ on the training data $a_1, \ldots, a_{30}$ (green circle). We repeated this for several "monte carlo" iterations (different random seeds = different noise vectors). Each iteration has a different number shown on top. The only difference between slides (for same fitted degree d) is the noise vector

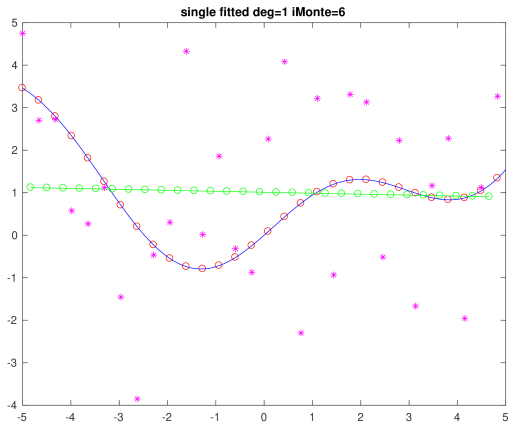

single fitted deg=1 iMonte=1


single fitted deg=1 iMonte=2


single fitted deg=1 iMonte=3



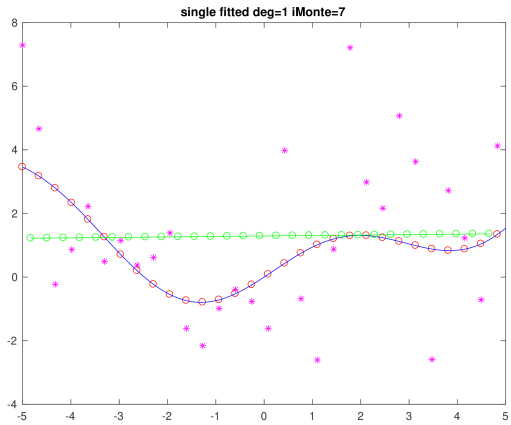


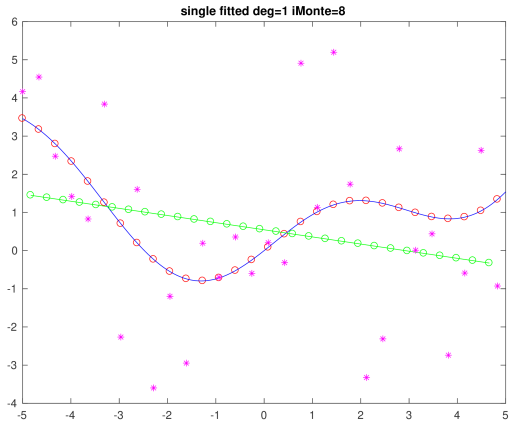


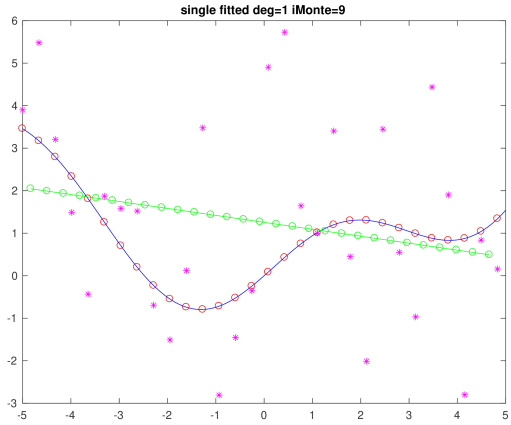
5.5 Variance

The above figures show that for different random draws of noise, the fitted polynomial $\hat{p}$ is different. That's expected: **When the labels y_i are random, the prediction rule we learn, h_S, is also random** What will cause h_S to have more variance? Clearly, the higher the noise level, the more variance in h_S. But is there anything else? What happens to the variance when we take a larger hypothesis class?

Let's have a look:



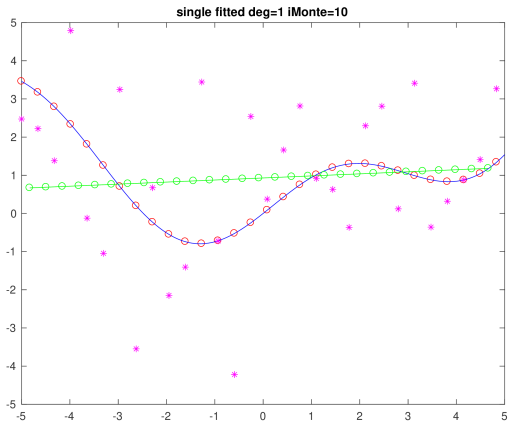


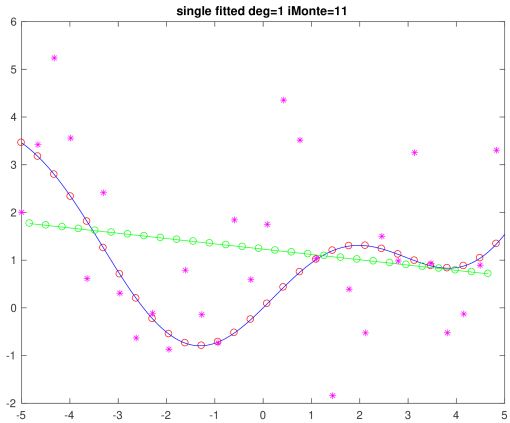


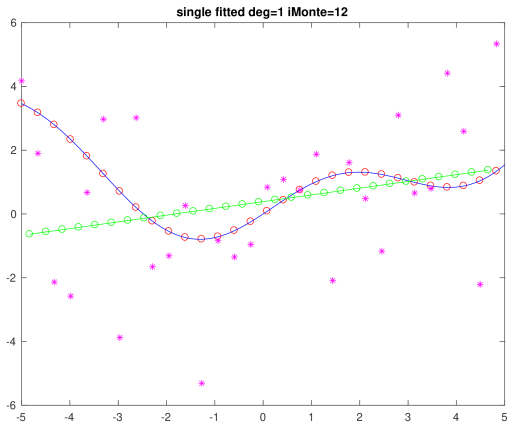
Informally, **Variance** is the variability in the learned prediction rule h_S. Informally, the larger the hypothesis class, the higher the variance because the learning algorithm "chases after the noise" and "sees shapes in the clouds" - uses the freedom allowed by a larger hypothesis class and **tries to describe the noise**

5.6 Variance in linear model: Geometrical Interpretation

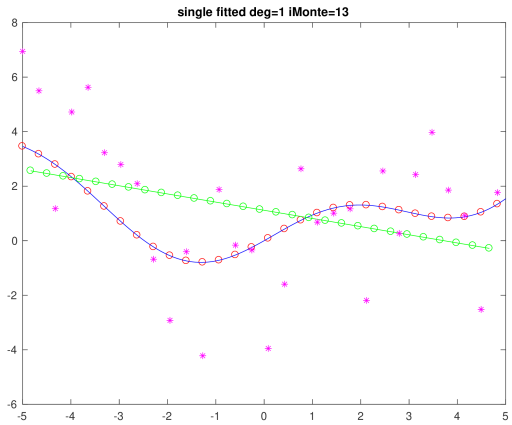
Assume $y_i = \mathbf{x}_i^\top \mathbf{w} + z_i$, $i = 1, \ldots, m$. To estimate $\mathbf{w}$ we project $\mathbf{y}$ onto $Im(X^\top)$. Since the noise is i.i.d so, roughly speaking, it adds the same variability in all directions. The larger $Im(X^\top)$ inside the space $\mathbb{R}^m$, the more noise is captured by the projection, and the more variability in

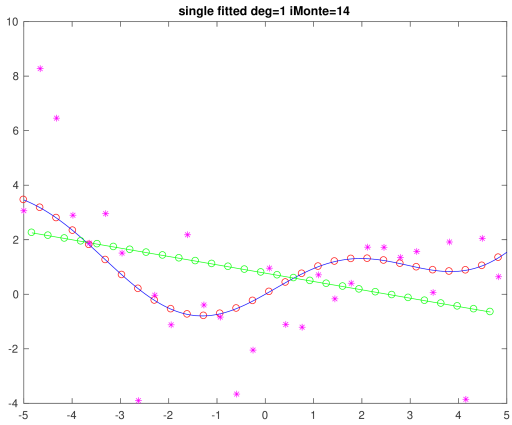


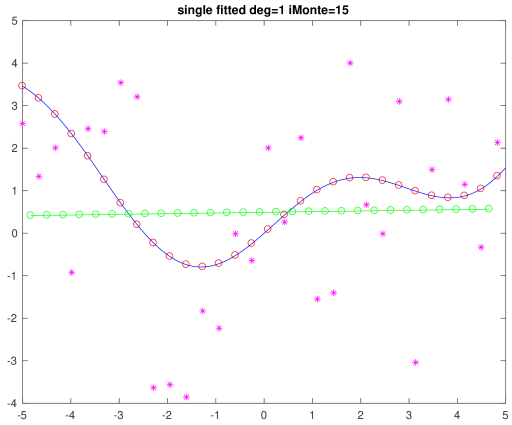




$\mathbf{w}_S$ is seen. In the extreme case $m = d + 1$ and $XX^\top$ invertible, $Im(X^\top)$ fills the whole space, so the projection does nothing. The noise has the largest possible influence on $\mathbf{w}_S$. We see that increasing training set size m reduces the variance of $\mathbf{w}_S$



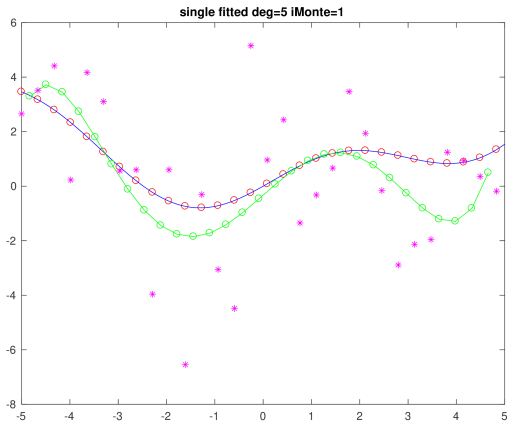


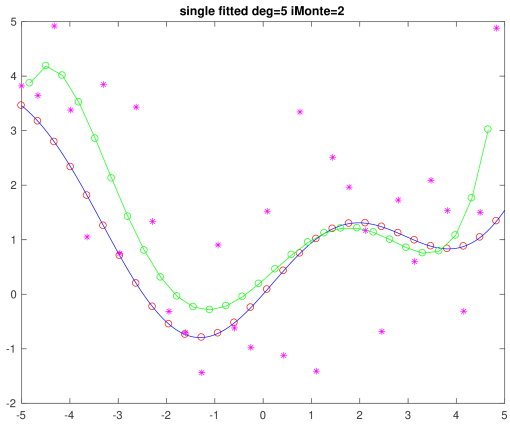


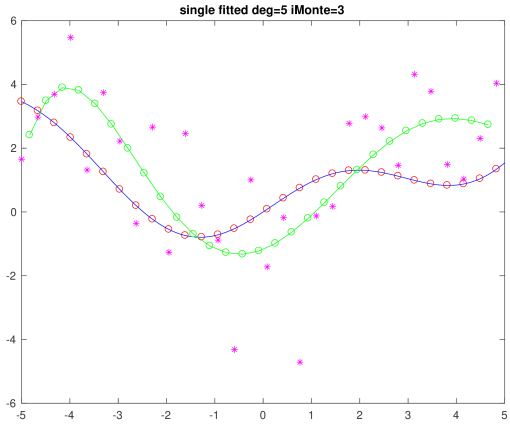
5.7 Bias-Variance Tradeoff

What we have seen is a general phenomenon in machine learning:

- A small hypothesis class ("low model complexity") will generally cause high bias and low variance
- A large hypothesis class ("high model complexity") will generally cause low bias and high variance



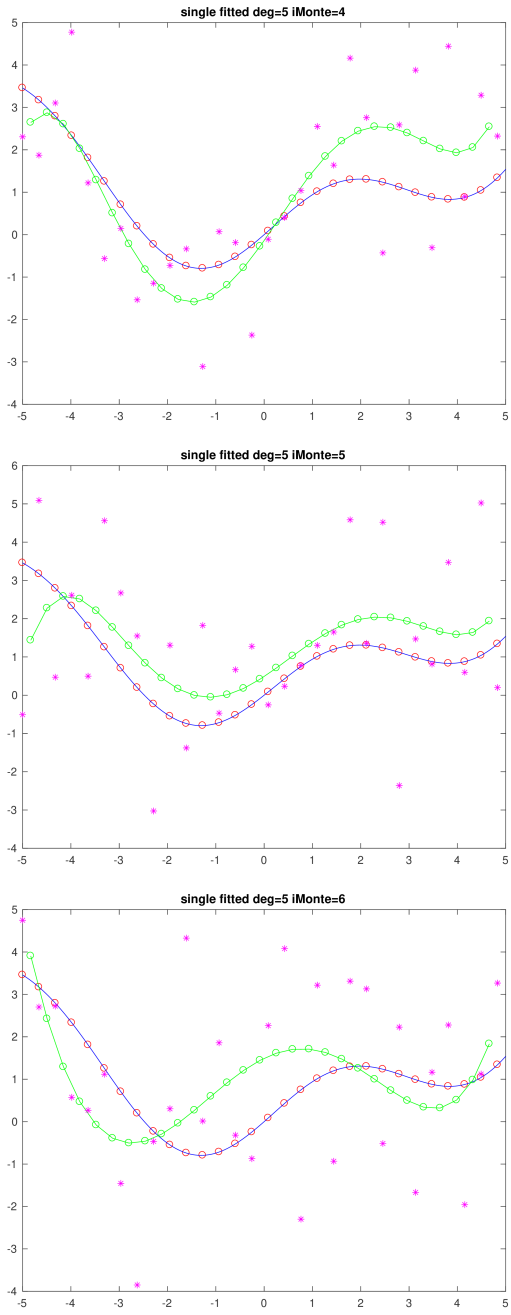




Lecture 3: Classification

6 Introduction

In this lecture we will become familiar with many methods for classification. Our linear regression lecture was a little of everything - theory and practical aspects. This will be a practical lecture - our goal is to be familiar with a few of the "classical" classification learning algorithm, understand the principles they are based on, and know how and when to use them. Sometimes we will only have limited understanding on **why** some learning algorithm performs better than others on some dataset. Sometimes we will have solid understanding based on theory;

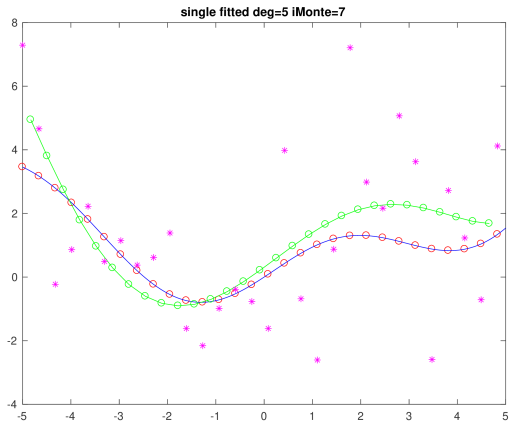


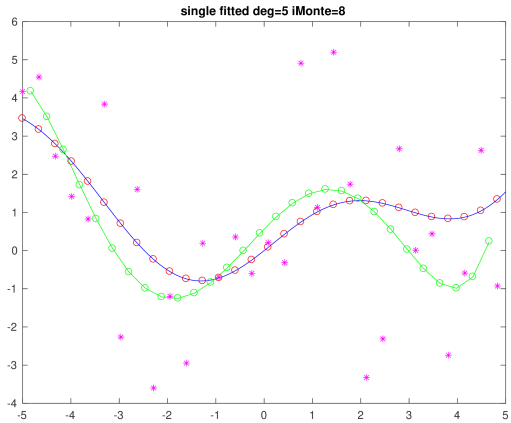
sometimes we won't understand at all.

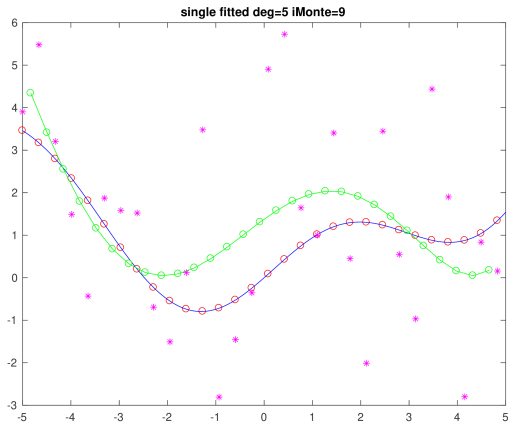
6.1 Textbook references

UML = Understanding Machine Learning; ESL = Elements of Statistical Learning 2nd ed. **recommended reading in bold**

- Half-spaces: **UML 9.1**, ESL 4.5
- Support Vector Machines: **UML 15.1, 15.2**, **ESL 12.2**
- Logistic regression: **ESL 4.4 (up to 4.4.3)**, UML 9.3



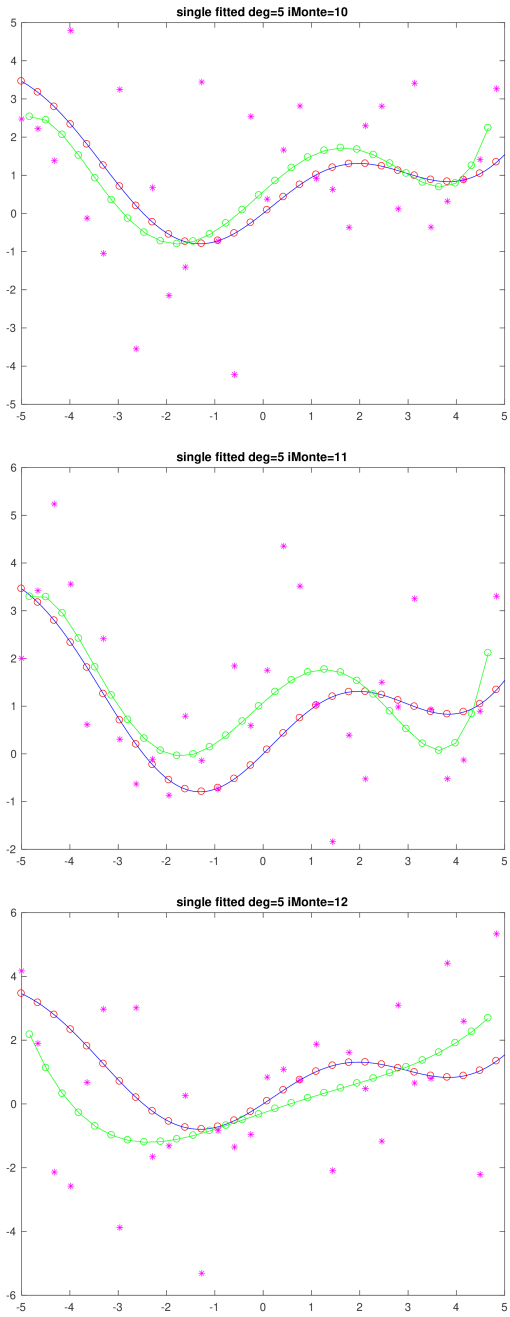




- Nearest Neighbors: **ESL 13.3**, UML 19
- Classification trees and random forests: **ESL 9.2**, UML 18.2

6.2 Some preliminaries

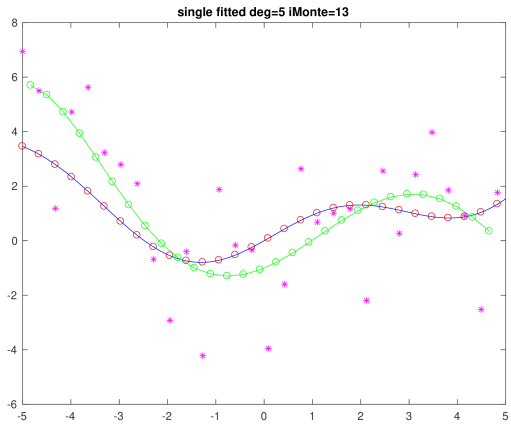
Recall that there are two "flavors" for supervised learning. When we are learning a function $f : \mathcal{X} \to \mathcal{Y}$ and $\mathcal{Y} = \mathbb{R}$, this is a **regression** problem. When $\mathcal{Y}$ is a finite set, this is a **classification** problem. We distinguish between classification problems (when $\mathcal{Y}$ is the simplest possible finite set, $\mathcal{Y} = \pm 1$) and multi-class classification problems, when $\mathcal{Y} = \{1, \ldots k\}$. In a binary classification problem (or just "classification"), we provide a yes/no prediction. In a

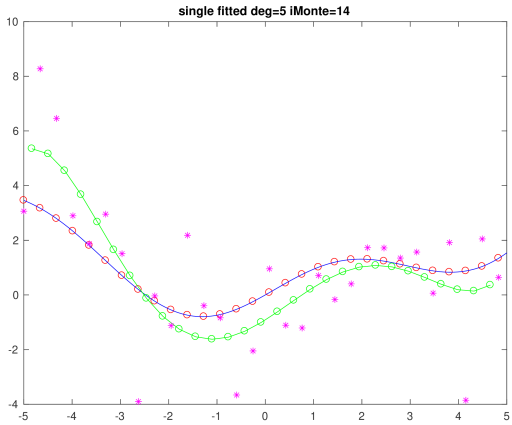


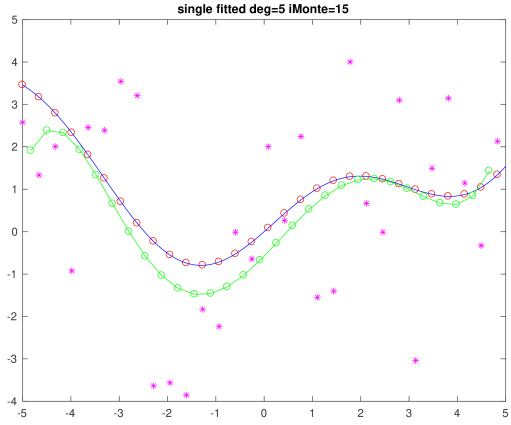
multi-class classification, we predict one of $k > 2$ classes. In this lecture we will only deal with binary classification. All the methods you will learn here can be generalized to k classes. Also, in this lecture we will only deal with the Euclidean sample space $\mathcal{X} = \mathbb{R}^d$, namely, each sample has d **features**. So in this lecture $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{\pm 1\}$. Some of the methods we will learn can be "persuaded" to work on other kinds of sample spaces $\mathcal{X}$, but this will not be part of this lecture.

6.3 Examples for classification problems

- Predict whether a patient will develop a certain medical condition, or not







- Predict whether a user will like a new product, or not
- Determine whether network traffic pattern is a cyber attack, or normal traffic
- Determine whether an art work is an original or forged
- Determine whether a given email is spam or not
- Detect fraud on credit card transactions
- Predict whether a loan applicant will default on the loan
- ...

6.4 Heart disease data

Here is an example classification problem to get us started. (See "Elements of Statistical Learning" section 4.4.2). It is believed that high blood pressure, smoking, family history, etc are related to heart disease. Can we predict, in a sample of people, who has developed a heart disease and who did not, based on this information?

The features collected are:

sbp	systolic blood pressure
tobacco	cumulative tobacco (kg)
ldl	low density lipoprotein cholesterol
adiposity	
famhist	family history of heart disease (yes/no)
typea	type-A behavior
obesity	
alcohol	current alcohol consumption
age	age at onset
chd	response, coronary heart disease (yes/no)

Here is some of the data:

	sbp	tobacco	ldl	adiposity	famhist	typea	obesity	alcohol	age	chd
1	160	12.00	5.73	23.11	Present	49	25.30	97.20	52	1
2	144	0.01	4.41	28.61	Absent	55	28.87	2.06	63	1
3	118	0.08	3.48	32.28	Present	52	29.14	3.81	46	0
4	170	7.50	6.41	38.03	Present	51	31.99	24.26	58	1
5	134	13.60	3.50	27.78	Present	60	25.99	57.34	49	1
6	132	6.20	6.47	36.21	Present	62	30.77	14.14	45	0
7	142	4.05	3.38	16.20	Absent	59	20.81	2.62	38	0
8	114	4.08	4.59	14.60	Present	62	23.11	6.72	58	1
9	114	0.00	3.83	19.40	Present	49	24.86	2.49	29	0
10	132	0.00	5.80	30.96	Present	69	30.11	0.00	53	1
11	206	6.00	2.95	32.27	Absent	72	26.81	56.06	60	1
12	134	14.10	4.44	22.39	Present	65	23.09	0.00	40	1
13	118	0.00	1.88	10.05	Absent	59	21.57	0.00	17	0
14	132	0.00	1.87	17.21	Absent	49	23.63	0.97	15	0
15	112	9.65	2.29	17.20	Present	54	23.53	0.68	53	0
16	117	1.53	2.44	28.95	Present	35	25.89	30.03	46	0
17	120	7.50	15.33	22.00	Absent	60	25.31	34.49	49	0

Figure 10: South African Heart Data from ESL

As you can see, some of the features are numerical and some categorical. You can download this data from the book's website and have fun trying different classifiers on it.

6.5 Plotting and imagining a feature space $\mathbb{R}^d$ with binary labeled data

Try to imagine a training sample for binary classification in $\mathbb{R}^d$, namely, points in $\mathbb{R}^d$ that come with a label in $\{1, -1\}$ (say). In the slides and handout, we can only plot examples in $\mathbb{R}^2$, like the one below, but you should always try to imagine a higher-dimensional case.

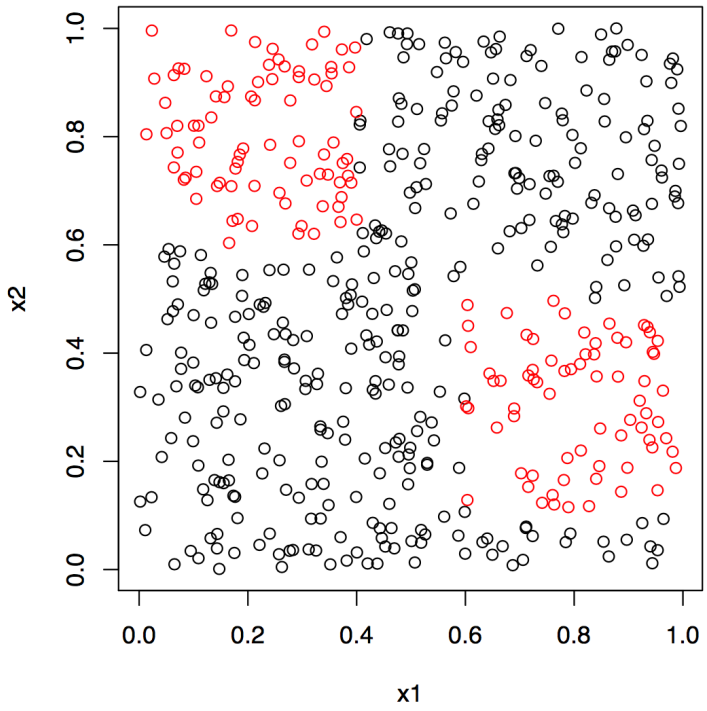


Figure 11: Classification training sample in $\mathbb{R}^2$

6.6 What loss function should we use?

Recall from the linear regression lecture that we decided to measure performance using the **square loss**. That was a reasonable choice for a regression problem. How shall we measure performance for **classification?**

- The first idea is **misclassification error**: simply count the samples in which the prediction and the true label did not agree. Given a prediction (classification) rule $h : \mathcal{X} \to \{\pm 1\}$, and a labeled sample $S = \{(x_i, y_i\}_{i=1}^m$ (such as a training sample), the misclassification (or accuracy) loss of h on this sample will be

$$L_S(h) := \sum_{i=1}^{m} \mathbf{1}_{y_i \neq h(x_i)} = |\{i | y_i \neq h(x_i)\}|$$

- Can there be any problems or issues with the misclassification loss? After all, it just counts the number of times h was wrong - the number of times h misclassified a sample.

- The main problem is that - in practice - there are two kinds of errors the classifier can make, and making each kind of error can have very different implications, or costs. So just counting how many errors in total the classifier made may not be a useful performance measure.

6.7 Type-I and Type-II errors

- **Example: Credit decisions.** Suppose we are building a classifier that predicts whether a bank customer seeking a loan is credit-worthy and will return a loan if given one. We

choose the labels such that -1 means "not credit worthy, deny loan" and 1 means "credit worthy, approve loan". What are the two kinds of **errors** our classifier can make?

- Let the true label of a sample be y_i, and the classifier-predicted label $\hat{y}_i$. If $y_i = -1$ and $\hat{y}_i = 1$, the classifier predicted that a non-credit-worthy customer will return the loan. If we act on this prediction, and the customer defaults on the loan, the bank loses all the loan sum. On the other hand, if $y_i = 1$ and $\hat{y}_i = -1$, the classifier predicted that a credit-worthy customer, which would have paid the interest and returned the loan in full, is not credit-worthy and should be denied the loan. If we act on this recommendation, the bank loses the interest it would have earned on the loan. **Which of the two errors is more serious? Which of the two errors costs more to the bank?** If we could choose which error we should avoid "at all costs" and which error we can "allow to happen", what will we choose?

- **Example: Drug safety.** Let's look at another example, an extreme example to help illustrate this point. We are building a classifier that predicts whether a certain drug is **safe to use** for a particular person, or **unsafe** / deadly / dangerous to use. We choose the labels such that -1 means "unsafe drug, do not use" and 1 means "safe drug, ok to use".

  What are the two kinds of errors the classifier can make? If $y_i = -1$ and $\hat{y}_i = 1$, the classifier recommends to give a drug to a patient, which can actually kill them (say). And if $y_i = 1$ and $\hat{y}_i = -1$, the classifier recommends that the patient should avoid a drug, which is actually safe to use. **Which of the two errors is more serious?** If we could choose which error we should avoid "at all costs" and which error we can "allow to happen", what will we choose?

- So we see that, depending on the context of the classification problem, the two kinds of errors can have very different costs. The "bad" error, which we would like to avoid at all costs, is called Type-I error in the statistics literature. The "not-so-bad" error is called Type-II error.

- In a classification problem we always choose one label in $\mathcal{Y} = \{\pm 1\}$ and call it "no effect" or "negative" and the other "effect" or "positive". Suppose we called -1 "negative" and called 1 "positive".

  Type-I and Type-II errors are defined as follows:

	$y_i = -1$	$y_i = 1$
$\hat{y}_i = -1$	·	Type-II error
$\hat{y}_i = 1$	Type-I error	·

- Choosing "negative" and "positive" defines which error is the Type-I error. For example, for the classification problem "Is the new drug safe to use?"
  We can assign the following meaning to the labels:
  $y = -1$: (negative) The new drug is safe
  $y = 1$: (positive) The new drug is dangerous
  Then a Type-I error means: we decided not to offer a safe drug

  But if we reverse the meaning to have

$y = -1$: (negative) The new drug is dangerous
$y = 1$: (positive) The new drug is safe
Then a Type-I error means: we decided to offer a dangerous drug. This is obviously the more serious of the two kinds of errors we can make.

- We always try to choose the "negative" and "positive" label such that the error we really want to avoid is the Type-I error - the error of taking a negative sample and by mistake predicting it is positive.
- Exercise: For the classification problem of "is this email spam or not" how would you choose the labels? What are the two errors a spam detector can make? which one is the one we really want to avoid? So, which of the labels "spam email" and "not spam, valid email" would you label "negative" and which is "positive"?

6.8 False Positive, False Negative and all that jazz

- Here is important terminology you will hear with regards to classification errors.
- Assume the true label is $y = -1$ (no effect, negative). Then
  - $\hat{y} = -1$ is **true negative**
  - $\hat{y} = 1$ is **false positive**
- Now assume the true label is $y = 1$ (effect, positive). Then
  - $\hat{y} = 1$ is **true positive**
  - $\hat{y} = -1$ is **false negative**
- Note: false negative and false positives are the misclassification errors. False positive is what we called Type-I error, False negative is what we called Type-II error. True negative and true positve are the two cases where the classifier is **correct**. It may take a little time to get used to this terminology. It's helpful to remember that something beginning with "false" is an error, and that the second word is the predicted label, not the true label. So that, for example, "false positive" is an **error**, where we **predicted** positive, so that the true label is actually **negative**.
- Some more terminology:
  - Denote P number of positives, N nuber of negatives
  - Denote TP and FP number of true and false positives
  - Denote TN and FN number of true and false negatives
  - Then
    - Error rate: $(FP + FN)/(P + N)$
    - Accuracy $(TP + TN)/(P + N)$
    - Precision: $TP/(TP + FP)$
    - Recall (sensitivity, true positive rate): TP/P
    - Specificity: TN/N
    - False positive rate: FP/N

6.9 Decision boundary

Let h be a binary classification rule in $\mathbb{R}^d$. (Suppose, for example, that we used a training sample to select h from some hypothesis class $\mathcal{H}$. We can feed any point $\mathbf{x} \in \mathbb{R}^d$ into h and get one of two classes. This means that $\mathbb{R}^d$ is a disjoint union of two sets:

$$\mathbb{R}^d = \{\mathbf{x}|h(\mathbf{x} = 1\} \biguplus \{\mathbf{x}|h(\mathbf{x} = 0\}$$

(if the class labels are, say, $\mathcal{Y} = \{0, 1\}$). These sets can be very simple (two half-spaces) or very complicated. The boundary between these two sets is called the **decision boundary**: a test sample on one side of the boundary will be classified to one class by h, and a test sample on the other side of the boundary will be classified to the other class.

Figure 12 is a simple (in fact linear) decision boundary.

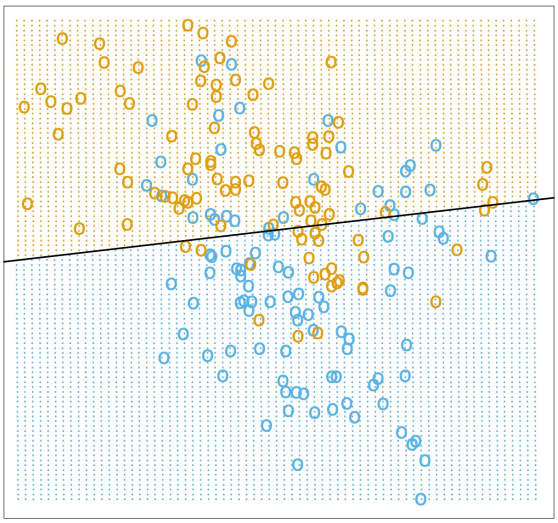


Figure 12: Decision boundary of a linear classifier (source: ESL)

Figure 13 is a more complicated decision boundary.

In learning a new classification method, it is useful to understand the **shape** of the decision boundaries, to observe them in simulation, to plot them for real data, etc. This can help us develop intuition when we work we that classifier, and also get a qualitative assessment of the bias and the variance of that classifier.

6.10 Our goal in this lecture

We will cover five very different classification algorithm. Our first goal is to be familiar with these methods, understand how they work, how and when to use them. Our second goal is to know **how to read about a new classifier.** There are hundreds of methods our there, and as an expert, it's best to know which questions to ask when you approach a new learning algorithm, and what to key points you want to make sure you understand about it.

Here is the list of questions we will ask about each method. Some of these questions will be explained as we go along.

- What is the hypothesis class $\mathcal{H}$? How does the decision boundary look like?

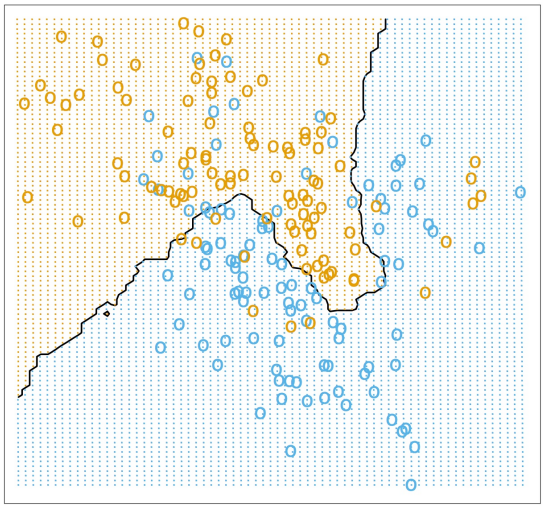


Figure 13: Decision boundary of a k-nearest-neighbor classifier (source: ESL

- What is the learning principle for training model (choosing $h_S \in \mathcal{H}$ based on training sample S)?
- How do we implement this principle computationally - how do we train the model and find $h_S \in \mathcal{H}$ in practice? What's the time complexity of this method?
- After training, how do we store trained model $h_S \in \mathcal{H}$? How much space is needed to store h_S?
- Given a trained model (namely, that we found $h_S \in \mathcal{H}$), how do we make predictions on new samples (namely, how do we compute $h_S(x)$ for a new sample $x \in \mathcal{X}$?) What is the time complexity of making a prediction for a single new sample?
- Is the learner interpretable?
- Does the learner provide estimated class probabilities?
- Is this a single model (a single hypothesis class $\mathcal{H}$), or actually a family of models (a family of hypothesis classes)? If a family, what are the parameters that choose the model from the family, and how do they affect the bias-variance tradeoff?
- When will we use this learning algorithm? What are its advantages and disadvantages?

7 The Half-Space classifier

Let's start with one of the simplest classifier imaginable. Our sample space is $\mathbb{R}^d$ and we have a training sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$. It will be convenient to work with the class labels $\mathcal{Y} = \{+1, -1\}$.

Our hypothesis class $\mathcal{H}_{half}$ is the set of half-space classifiers in $\mathbb{R}^d$. These are functions of the form

$$\mathbf{x} \mapsto \text{sign}\left(\langle \mathbf{x}\,,\, \mathbf{w} \rangle + b\right) .$$

Each such function is completely determines by the vector $\mathbf{w} \in \mathbb{R}^d$ and the scalar $b \in \mathbb{R}$. (This should remind you of the linear regression hypothesis class[15].)

To see why the function $h_{\mathbf{w},b}(\mathbf{x}) = \text{sign}\left(\langle \mathbf{x}\,,\,\mathbf{w}\rangle + b\right)$ is a half-space classifier, assume first that $b = 0$. Convince yourself that

$$\mathbb{R}^d = \left\{\mathbf{x} \in \mathbb{R}^d \,|\, \langle \mathbf{x}\,,\,\mathbf{w}\rangle > 0\right\} \biguplus \left\{\mathbf{x} \in \mathbb{R}^d \,|\, \langle \mathbf{x}\,,\,\mathbf{w}\rangle < 0\right\} \biguplus \left\{\mathbf{x} \in \mathbb{R}^d \,|\, \langle \mathbf{x}\,,\,\mathbf{w}\rangle = 0\right\}$$

and that these sets correspond to the open half space on one side of the hyperplane $\mathbf{w}^\perp = \left\{\mathbf{x} \in \mathbb{R}^d \,|\, \langle \mathbf{x}\,,\,\mathbf{w}\rangle = 0\right\}$, the open half space on the other side of that hyperplane, and points on the hyperplane itself. So each vector $\mathbf{w} \in \mathbb{R}^d$ defines a hyperplane $\mathbf{w}^\perp$ and divides $\mathbb{R}^d$ into two half-spaces.

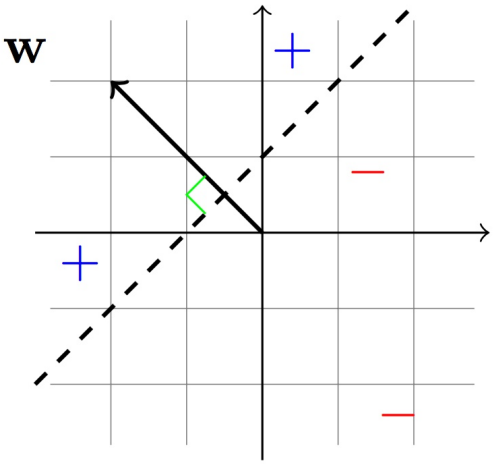


Figure 14: Half-space defined by a vector. Source: UML

The case $b = 0$ is called the **homogeneous** case, as the hyperplane $\mathbf{w}^\perp$ runs through the origin of $\mathbb{R}^d$. When $b \neq 0$, the hyperplane does not run through the origin.

7.1 Assumption - training sample is linearly separable

Let's start with the unreasonable assumption that the training samples are **linearly separable**, in the sense that there exists a hyperplane for which all training samples labeled 1 are on one side and all training samples labeled -1 are on the other side.

Mathematically, this means that we assume that, for our training sample S, there exist a vector $\mathbf{w} \in \mathbb{R}^d$ and a scalar $b \in \mathbb{R}$ such that $y_i \cdot \text{sign}\left(\langle \mathbf{x}\,,\,\mathbf{w}\rangle + b\right) = 1$ for all i, or equivalently

$$y_i \cdot \left(\langle \mathbf{x}\,,\,\mathbf{w}\rangle + b\right) > 0 \qquad i = 1, \ldots, m$$

since the inner product will be negative for samples with $y_i < 0$ and positive for samples with $y_i > 0$. (If we chose $\mathbf{w}$ pointing at the "wrong direction", we will have $y_i \cdot \text{sign}\left(\langle \mathbf{x}\,,\,\mathbf{w}\rangle + b\right) < 0$. In this case we just change $\mathbf{w}$ to $-\mathbf{w}$.)

[15] Indeed, While in linear regression we predicted a real-valued label with the class of linear functions (functions of the form $\mathbf{x} \mapsto \langle \mathbf{x}\,,\,\mathbf{w}\rangle + b$), now we predict a binary label with just a level-set of a linear function.

For simplicity, let's work with the homogeneous case $b = 0$. (We can always shift the entire training sample by a fixed vector if needed, to make sure that the separating hyperplane runs through the origin.)

So our hypothesis class is

$$\mathcal{H}_{half} = \left\{ h_{\mathbf{w}} : \mathbf{x} \mapsto \text{sign}\left(\langle \mathbf{x}\,,\, \mathbf{w} \rangle\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^d \right\}$$

and, just like in linear regression, training the model reduces to finding vector $\mathbf{w} \in \mathbb{R}^d$. Note that our assumption that the training sample is linearly separable is known as the **realizability assumption** - namely, **that the labels are generating by a function in our hypothesis class.**

7.2 Learning using ERM

Now that we have our hypothesis class, how do we train the model - namely, how do we choose $h \in \mathcal{H}_{half}$ given a training set?

Observe that, for a hypothesis $h_{\mathbf{w}} \in \mathcal{H}_{half}$, the misclassified training samples are exactly those samples with $y_i \cdot \text{sign}\left(\langle \mathbf{x}\,,\, \mathbf{w} \rangle\right) = -1$ or equivalently $y_i \cdot \langle \mathbf{x}\,,\, \mathbf{w} \rangle < 0$

The loss of $h_{\mathbf{w}}$ on the training sample S is therefore

$$L_S(h_{\mathbf{w}}) = \sum_{i=1}^{m} \mathbf{1}_{y_i \cdot \langle \mathbf{x}\,,\, \mathbf{w} \rangle < 0}$$

Since the training sample is linearly separable by assumption, clearly it makes sense to choose $h \in \mathcal{H}_{half}$ that perfectly separates the training sample. Such $h \in \mathcal{H}_{half}$ will have zero training loss, $L_S(h_{\mathbf{w}}) = 0$. In other words, any separating hyperplane $\mathbf{w}^{\perp}$ is going to correspond to an hypothesis $h_{\mathbf{w}}$ that minimizes the empirical risk $L_S(h_{\mathbf{w}})$. (By assumption, the minimal value will be 0). So we decide that the learning principle will again be **empirical risk minimization - the same principle we used for developing linear regression.**

7.3 Computationally implementing ERM for halfspace classifiers

The question is now if there is a computationally efficient way to find

$$\text{argmin}_{\mathbf{w} \in \mathbb{R}^d} L_S(h_{\mathbf{w}})\,.$$

By assumption there exists a vector $\mathbf{w_0} \in \mathbb{R}^d$ such that

$$y_i \cdot \langle \mathbf{x}\,,\, \mathbf{w_0} \rangle > 0 \qquad i = 1, \ldots, m$$

First observe that this implies that there also exists a (different) vector $\mathbf{w_1} \in \mathbb{R}^d$ such that

$$y_i \cdot \langle \mathbf{x}\,,\, \mathbf{w_1} \rangle \geqslant 1 \qquad i = 1, \ldots, m$$

To see why this is true, just normalize $\mathbf{w_0}$ by the smallest product, namely, define

$$\mathbf{w_1} := \frac{1}{\min_i \{ y_i \cdot \langle \mathbf{x}\,,\, \mathbf{w_0} \rangle \}} \cdot \mathbf{w_0}\,.$$

So it's enough to look for a vector $\mathbf{w}$ that satisfy $y_i \cdot \text{sign}\left(\langle \mathbf{x}\,,\, \mathbf{w} \rangle\right) \geqslant 1 \qquad i = 1, \ldots, m$

We are looking for a vector that satisfies m linear constrains. Of course we know how to do this - linear programming.

7.4 Commercial break: Convex optimization

Before we continue with half-spaces, let us break and briefly introduce convex optimization, since we will start using convex optimization tools and concepts.

- **An optimization problem** over $\mathbb{R}^d$ has the general form

$$\begin{aligned} &\text{minimize} && f_0(\mathbf{x}) \\ &\text{subject to} && f_i(\mathbf{x}) \leqslant b_i, \qquad i = 1 \dots n \end{aligned}$$

  where $\mathbf{x}$ is the **optimization variable**, $f_0 : \mathbb{R}^d \to \mathbb{R}$ is the **objective function**, and $f_i : \mathbb{R}^d \to \mathbb{R}$ $(i = 1, \dots, m)$ are the **constraint functions**. It is implicitly implied that the optimization happens over $dom(f_0) \subset \mathbb{R}^d$, the domain of f_0.

- **A convex function** is function $f : \mathbb{R}^d \to \mathbb{R}$ such that $dom(f)$ is a convex set in $\mathbb{R}^d$, and

$$f(\alpha\mathbf{x} + \beta\mathbf{z}) \leqslant \alpha f(\mathbf{x}) + \beta f(\mathbf{z})$$

  for all $\alpha, \beta \in \mathbb{R}$ and all $\mathbf{x}, \mathbf{z} \in dom(f)$.

- **A convex optimization problem** is an optimization problem as above in which $f_0, f_1, \dots, f_n$ are all convex functions.

- **A linear programming problem (or a linear program)** is a special case of a convex optimization problem, in which $f_0, f_1, \dots, f_n$ are all **linear** functions.

In general, optimization problems are hard to solve computationally. We take special interest in **convex optimization** problems since there have a unique solution, and that solution can be found in computationally tractable ways. A great deal is known about **convex optimization algorithms**, which are iterative numerical algorithms that converge to the solution of a convex optimization problem. There are general solvers, which will solve a convex problem in the general form above, and there are specialized solvers for specific types, or families, of convex optimization problems. A specialized solver is typically preferred, as it leverages some particular structure of the problem to solve it more efficiently, using less space, etc. One example for specialized solvers you've seen in Algorithms course are specialized solvers for linear programs.

Why is convex optimization interesting for machine learning? In supervised learning, we would like to choose a hypothesis $h \in \mathcal{H}$ from our selected hypothesis class, based on some learning principle (such as ERM). Many learning principles are formulated as optimization problems, namely, the h our learning algorithm chooses is given as the minimizer of some quantity (such as empirical risk). So implementation of the learning algorithm needs to solve an optimization problem.

Sometimes, our hypothesis class is equivalent to a Euclidean space (for example, in linear regression we saw that linear functions are determined by the weight vector and an intercept, so the hypothesis class of linear functions was equivalent to the Euclidean space $\mathbb{R}^{d+1}$, and we've just seen the same happens for the hypothesis class of half-space classifiers). When this happens, our learning principle reduces to solving an optimization problem, namely, the hypothesis we choose $h \in \mathcal{H}$ is found as a minimum over $\mathbb{R}^d$ or a subset of $\mathbb{R}^d$ of some objective function, usually a loss function. When this objective is convex, we can use convex optimization algorithms to implement our learning algorithm efficiently.

(We will have a future lecture dedicated to theory and algorithms of convex optimization.)

7.5 Solving ERM for half-space by linear programming

Back to the half-space classifier. We saw that a hyperplane $\mathbf{w}^{\perp}$ minimizing ERM is a solution to the following linear program:

$$\begin{aligned} &\text{minimize} && 0 \\ &\text{subject to} && y_i \cdot \langle \mathbf{x}_i \, , \, \mathbf{w} \rangle \geqslant 1 \qquad i = 1, \ldots, m \end{aligned}$$

Such an optimization problem (with trivial objective) is called a **feasibility** problem - we're just looking for any vector which satisfies the constrains.

So, one way to implement ERM for the half-space classifier is using linear programming. There is another famous way, known as the **Perceptron** algorithm. This is an iterative algorithm with a guaranteed rate of convergence to a solution. You'll see this algorithm in the recitation.

7.6 What about a training sample that's not linearly separable?

When the sample training is not linearly separable, the optimization problem above has no solution. We can still try to implement the ERM principle (find a hyperplane with minimal number of misclassification errors) however unfortunately in this case the minimization problem

$$\text{argmin}_{\mathbf{w} \in \mathbb{R}^d} L_S(h_{\mathbf{w}}) \, .$$

is known to be computationally hard.

7.7 Summary

- Hypothesis class $\mathcal{H}$: Half-spaces
- Learning principle for training model (choosing $f \in \mathcal{H}$): Separating hyperplane chosen with Empirical Risk Minimization (ERM) for misclassification loss
- Computational implementation of learning principle: Linear programming, Perceptron
- How to store trained model: store the vector $\mathbf{w}$ perpendicular to the hyperplane defining the half-space
- When to use: As a simple baseline. Otherwise never.

See if you can fill in these yourself:

- How to make predictions on new samples?
- Time complexity for training, and for predicting on a new sample?

8 Support Vector Machines

We encountered two problems with the half-space classifier.

- Even if the training sample is linearly separable, the half-space separating is not unique. Which one should we choose? (see figure - which of the hyperplanes should we choose?)

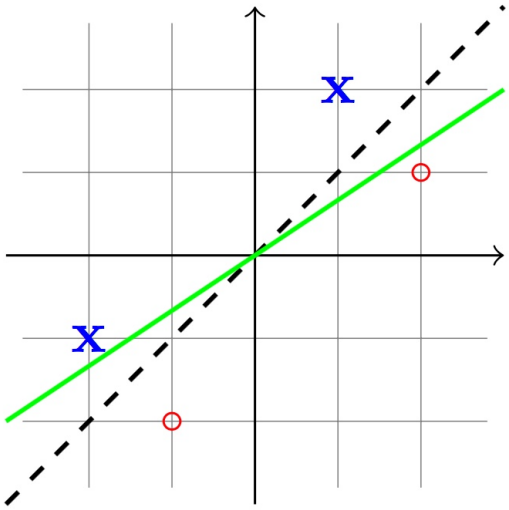


Figure 15: Which separating hyperplane should we choose? Source: UML

- A much more severe problem is that we chose to work with the ERM principle to select the hypothesis h, but when the training sample is not linearly separable, implementing ERM is computationally hard. So the half-space classifier is not very useful in practice.

We'll now continue with the same hypothesis class if half-spaces. We won't assume that they are defined by hyperplanes that go through the origin. So our hypothesis class is now

$$\mathcal{H}_{SVM} = \left\{ h_{\mathbf{w}} : \mathbf{x} \mapsto \text{sign}\left(\langle \mathbf{x}\,,\, \mathbf{w} \rangle + b\right) \;\middle|\; \mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R} \right\}$$

But we won't use the ERM principle. We will introduce a totally different learning principle, one which solves both problems above. It gives us a unique hyperplane, and it can be implemented computationally - efficiently - even when the training sample is not linearly separable. You'll cover SVM again in the recitation, so this will part of the lecture will be relatively fast and a little less detailed.

8.1 A new learning principle: Maximum Margin

For a hyperplane $(\mathbf{w}, b)$ (defined by the vector $\mathbf{w} \in \mathbb{R}^d$ and a scalar b) and point $\mathbf{x} \in \mathbb{R}^d$, define the **distance** between $(\mathbf{w}, b)$ and $\mathbf{w}$ by

$$D((\mathbf{w}, b), \mathbf{x}) := \min_{\mathbf{v}:\langle \mathbf{v}\,,\,\mathbf{w}\rangle + b = 0} \|\mathbf{x} - \mathbf{v}\|$$

namely the Euclidean distance between $\mathbf{x}$ and the closest point on the hyperplane $(\mathbf{w}, b)$.

Now let's define the **margin** of a hyperplane, with respect to a training sample $S = \{(\mathbf{x}_i, y_i)\}$. The margin $M\left((\mathbf{w}, b), S\right)$ is simply the smallest distance between the hyperplane and any sample point $\mathbf{x}_i$:

$$M((\mathbf{w}, b), S) := \min_{1 \leqslant i \leqslant m} D((\mathbf{w}, b), \mathbf{x})\,.$$

Obviously, a hyperplane with a large margin is preferred - since, if it separates the training samples, the larger the margin, the better the separation.

Our new learning principle is: We will choose $h_{\mathbf{w},b} \in \mathcal{H}_{SVM}$ that has the **largest margin** with respect to our training sample S.

The vectors closest to the hyperplane determine the margin. They are called **support vectors**, hence this learner's name. (Calling it a support vector **machine** has no justified reason, but is just cool.)

8.2 Hard SVM

Let's start with the **realizable case**, namely make the same assumption we had for the half-plane classifier - that the training sample is linearly separable. (We'll remove this soon.) To implement our learning principle of maximal margin, we need to search, among all the hyperplane separating S, for the hyperplane with maximum margin. Namely, the hypothesis $h_{\mathbf{w},b} \in \mathcal{H}_{SVM}$ our learner will choose is the solution to the following optimization problem

$$\begin{aligned} &\text{maximize} && M((\mathbf{w},b),S) \\ &\text{subject to} && y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 \qquad i = 1, \ldots, m \end{aligned}$$

The optimization variables are $\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$. (Compare this to the linear program above for half-spaces: we kept the same constrains, which ensure that the hyperplane chosen will separate the training sample, but, instead of a trivial objective, we have the margin!) We don't worry about "maximize" instead of "minimize" as we can just multiply the objective by -1.

Is this a convex optimization problem? We hope so, since if yes, it is computationally tractable.

Exercise. If $\|\mathbf{w}\| = 1$ then $D((\mathbf{w},b) = |\langle \mathbf{x} \, , \, \mathbf{w} \rangle + b|$. (The solution is in "Understanding Machine Learning" claim 15.1).

We don't mind describing the hyperplane using a unit vector - it really doesn't matter. So our optimization problem is equivalent to:

$$\begin{aligned} &\text{maximize} && \min_{1 \leqslant i \leqslant m} |\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b| \\ &\text{subject to} && y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 \qquad i = 1, \ldots, m \end{aligned}$$

Well, is this convex? In the recitation you'll see that this problem is equivalent to the following optimization problem:

$$\begin{aligned} &\text{minimize} && \|\mathbf{w}\|^2 \\ &\text{subject to} && y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 \qquad i = 1, \ldots, m \end{aligned}$$

So, good news - you can see that in this latest optimization problem, the objective is convex (the norm function is convex by triangle inequality) and, as before, the constrains are linear. A convex optimization problem in which the objective is a quadratic form of the optimization variable, and the constrains are linear functions, is called **a Quadratic Program** (QP). This is

one of those special families of convex optimization problems that we mentioned. There are specialized solvers for Quadratic Programs that are much more efficient that general convex solvers.

This version of SVM is called **Hard SVM** (for "hard margin") and is only suitable for a separable training sample - indeed, if the training sample is not separable, the optimization problem above has no solutions as for any candidate $\mathbf{w}, b$), at least one of he constrains

$$y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1$$

cannot be satisfied.

8.3 Soft SVM

This last sentence gives us an idea. We can't really expect that the training sample will be linearly separable, meaning that we can't expect **all** the constrains

$$y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1$$

to be satisfied simultaneously. So we would like to allow this constrain to be violated - for as few training samples as possible, of course. Recall that if $y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) < 0$, this means that the sample $\mathbf{x}_i$ is on the "wrong side" of the hyperplane. Similarly, convince yourself that if there exists some $\xi_i > 0$ such that

$$y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 - \xi_i$$

this means that the sample $\mathbf{x}_i$ is on the wrong side of the **margin** by a proportional amount of ξ (try to understand mathematically the exact meaning of this sentence.)

So in order to allow training samples to violate the constrains "a little", and end up on the wrong side of the margin, we change the Hard-SVM optimization problem to:

$$\begin{aligned} &\text{minimize} \quad \|\mathbf{w}\|^2 \\ &\text{subject to} \quad \begin{cases} y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 - \xi_i \text{ and } \xi_i \geqslant 0 & i = 1, \ldots, m \\ \frac{1}{m} \sum_{i=1}^{m} \xi \leqslant C \end{cases} \end{aligned}$$

where $C > 0$ is a constant that we specify. The variables ξ_1, ξ_m are new auxiliary variables that we introduced. This is still a Quadratic Program.

The larger the constant C we choose, the more violations of the margin we allow. One the one hand, we want to allow "noisy" samples to violate the margin, so that the hyperplane will ignore them; on the other hand, if we allow to many violations, we lose touch with the training sample and its structure. This is exactly the bias-variance tradeoff: basically, the larger C, the more freedom the learner has to "chase after the training sample".

Another possibility for allowing violations is as follows:

$$\begin{aligned} &\text{minimize} \quad \lambda \|\mathbf{w}\|^2 + \frac{1}{m} \sum_{i=1}^{m} \xi_i \\ &\text{subject to} \quad y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 - \xi_i \text{ and } \xi_i \geqslant 0 \end{aligned}$$

Here, instead of specifying the constant C, we specify a constant λ that is included in the optimization problem's objective. The larger λ, the less sensitive the solution will be to the term $\frac{1}{m}\sum_{i=1}^{m}\xi_i$, and will allow more violations. The smaller λ, the more sensitive, and will allow less violations. If we choose to work with this optimization problem to choose h, the constant λ also moves us along different members of a family of learners, each with a different bias-variance tradeoff. λ is known as a **regularization parameter** - we will have a lecture about regularization later in the course, and will understand more about learners based on optimization problems that contain a regularization term.

8.4 A family of learners

In Soft SVM, we encounter something new and important: the soft SVM algorithm requires a parameter C or λ. The hypothesis produced by the learner - for a given training sample - depends heavily on the parameter. So in fact, Soft-SVM is not a single learning algorithm, but a family of learning algorithms. Changing C

8.5 When is SVM useful?

Since the hypothesis class of SVM is half-spaces, it is generally a classifier with high bias and low variance. SVM is useful in high dimensions, when the dimension d is large. In the recitation, you will see a method to "lift" a sample into a space with higher dimensions, where it can become linearly separable or almost linearly separable - and then SVM can be useful.

8.6 Summary - Soft SVM

- Hypothesis class $\mathcal{H}$: Half-spaces
- Learning principle for training model (choosing $f \in \mathcal{H}$): Linear separator with Maximum Margin
- Computational implementation of learning principle: Quadratic Program
- How to make predictions on new samples:
- Interpretable: No
- Estimates class probabilities: No
- Family of models: Yes, indexed by the regularization parameter $\lambda \in [0, \infty)$.
- Time complexity for training, and for predicting on a new sample:
- How to store trained model: store the vector $\mathbf{w}$ perpendicular to the separating half-space
- When to use: (i) As a simple baseline (ii) after embedding data in high-dimensional space (kernelization) - see next recitation.

9 Logistic Regression

9.1 A probabilistic model for noisy labels

- One way to think about linear regression with Gaussian errors: Assumed $y = f(\mathbf{x})$ for f a linear function in the sample vector $\mathbf{x}$. With noise, each label y_i is drawn from a random variable of the form $y_i = \langle \mathbf{x_i}, \mathbf{w} \rangle + z_i$ with $z_i \sim \mathcal{N}(0, \sigma^2)$. Equivalently $y_i \sim \mathcal{N}(\langle \mathbf{x_i}, \mathbf{w} \rangle, \sigma^2)$. We want to learn the vector $\mathbf{w}$ (or equivalently, the linear function $\mathbf{x} \mapsto f(\mathbf{x})$. In other words, we assumed that each sample $(\mathbf{x}, y)$ is such that the **expected value** of the label y is linear in $\mathbf{x}$.

- For classification on $\mathcal{X} = \mathbb{R}^d$, we have samples $(\mathbf{x}, y)$ with $\mathbf{x} \in \mathbb{R}^d$ and $y \in \{\pm 1\}$. When discussing logistic regression it is convenient to use labels $\{0, 1\}$ instead. Let's assume a probabilistic model similar to that of linear regression: it's natural to use **Bernoulli** random variables. Can we want to assume $y_i \sim Ber(p_i)$, for some $p_i \in [0, 1]$ that's related somehow to $\mathbf{x}_i$. This models noise: each sample is a result of some "inner" tendency to be 0 or 1, which we called p_i, but then a coin is flipped to determine y_i. It could be that p_i is very small and by chance we drew $y_i = 1$, for example - but that would be rare. So this is a good model for labels in $\{0, 1\}$ that come with noise.

- How shall we connect p_i to $\mathbf{x}_i$? We can't assume a linear function $\langle \mathbf{x_i}, \mathbf{w} \rangle$ for some $\mathbf{w} \in \mathbb{R}^{d+1}$, as in linear regression, since p_i is restricted to $[0, 1]$. So it would be natural to choose a **link** function $\phi : \mathbb{R} \to [0, 1]$ that is monotone increasing, and maps $(-\infty, \infty)$ bijectively to $(0, 1)$, and assume that $p_i = \phi(\langle \mathbf{x_i}, \mathbf{w} \rangle)$ for some $\mathbf{w} \in \mathbb{R}^{d+1}$.

- Why is $\mathbf{w} \in \mathbb{R}^{d+1}$ and not $\mathbf{w} \in \mathbb{R}^d$? just like in linear regression, we need an **intercept**. So we pad the sample vectors $\mathbf{w}$ with a 1 in the 0-th entry, so that $\langle \mathbf{x_i}, \mathbf{w} \rangle = \sum_{j=0}^{d} x_j w_j = w_0 + \sum_{j=1}^{d} x_j w_j$ where $\mathbf{x} = (1, x_1, \ldots, x_d)$ and $\mathbf{w} = (w_0, w_1, \ldots, w_d)$.

- So our model is that labels are independent of each other, and that there exists a vector $\mathbf{w}$ such that $y_i \sim Ber(\phi(\langle \mathbf{x_i}, \mathbf{w} \rangle))$. This hypothesis class is indexed by the vector $\mathbf{w} \in \mathbb{R}^{d+1}$ - and just like linear regression, learning comes down to finding $\mathbf{w}$.

- What function shall we use for ϕ? There are a few well known choices in the literature. Honestly it doesn't matter all that much, as long as the function $\phi : \mathbb{R} \to [0, 1]$ that is monotone increasing, and maps $(-\infty, \infty)$ bijectively to $(0, 1)$ (so that it's invertible). We would like ϕ to be smooth, so that we can use derivatives in any optimization that will be needed to train the model and choose $\mathbf{w}$ based on the training sample.

- The ϕ we will use is the famous **logistic function**

$$\pi(x) = \frac{e^x}{1 + e^x} .$$

  (verify that it is smooth, indeed analytic, monotone increasing, and maps $(-\infty, \infty)$ bijectively to $(0, 1)$).

9.2 The hypothesis class

- So our hypothesis class is simply

$$\mathcal{H}^d_{logistic} = \{\mathbf{x} \mapsto \pi(\langle \mathbf{x}, \mathbf{w} \rangle)\} .$$

But wait, something is not right. We are in a classification lecture, where $\mathcal{Y} = \{0, 1\}$, but this hypothesis class contains functions $\mathbb{R}^d \to [0, 1]$, not $\mathbb{R}^d \to \{0, 1\}$. That is correct. Note that since $\{0, 1\} \subset [0, 1]$, we can use our classification training sample to choose a function in $\mathcal{H}^d_{logistic}$. So we'll be able to train the model. But how will we predict on a new sample? If our learner chose some $h \in \mathcal{H}^d_{logistic}$, and equivalently some $\mathbf{w} \in \mathbb{R}^{d+1}$, how do we predict a label $\{0, 1\}$ for some sample $\mathbf{x}$? The value $h(\mathbf{x})$ is in $[0, 1]$ and may be, for example, 0.7. And we need a classifier, so need a value in $\{0, 1\}$, not in $[0, 1]$. The answer is that we think about $h(\mathbf{x})$ as an estimate of the probability that the label corresponding to $\mathbf{x}$ is 1. If $h(\mathbf{x}) = 0.1$, the label is likely 0. If $h(\mathbf{x}) = 0.9$, the label is likely 1. We will need to choose a **cutoff** α and our class prediction will be

$$\hat{y} := \begin{cases} 1 & h(\mathbf{x}) > \alpha \\ 0 & h(\mathbf{x}) \leqslant \alpha \end{cases} .$$

More on how to choose the cutoff α - soon. This is an important subject.

9.3 The learning principle: maximum likelihood

- Back to logistic regression. So we have our hypothesis class $\mathcal{H}^d_{logistic}$. What learning principle shall we use- how will we choose the function $h \in \mathcal{H}^d_{logistic}$, and, equivalently, the vector $\mathbf{w} \in \mathbb{R}^{d+1}$, from a training sample $S = \{(\mathbf{x}_i, y_i\}_{i=1}^m$? We will use the **maximum likelihood principle** (seen in the linear regression lecture).

- Assuming that $y_i \sim Ber(\pi(\langle \mathbf{x_i}\,,\, \mathbf{w}\rangle))$ where π is the logistic function, and assuming the labels y_i are all independent, we can write the **joint density** of the labels vector $\mathbf{y} = (y_1, \ldots, y_m) \in \{0, 1\}^m$. Let $Y = (Y_1, \ldots, Y_m)$ be a random vector taking values in $\{0, 1\}^m$ with independent entries, where $Y_i \sim Ber(\pi(\langle \mathbf{x_i}\,,\, \mathbf{w}\rangle))$. The joint distribution of Y is parametrized by $\mathbf{w}$, and

$$Prob(Y = \mathbf{y}|\mathbf{w}) = \prod_{i=1}^{m} p_i(\mathbf{w})^{y_i}(1 - p_i(\mathbf{w})^{1-y_i}$$

where

$$p_i = \pi(\langle \mathbf{x_i}\,,\, \mathbf{w}\rangle) = \frac{exp(\langle \mathbf{x_i}\,,\, \mathbf{w}\rangle}{1 + exp(\langle \mathbf{x_i}\,,\, \mathbf{w}\rangle} .$$

Now, applying the **maximum likelihood principle**, we'll turn this density into a likelihood by **fixing** the observes values $\mathbf{y}_i$ and looking at it as a function of the unknown parameter vector $\mathbf{w}$ (instead of looking at it as a probability density function with known parameter $\mathcal{V}w$, aimed to calculate the probability of observing certain values of y_i.) So we will choose the vector $\mathbf{w}$ for which the likelihood is maximal. Write

$$L(\mathbf{w}|\mathbf{y}) = \prod_{i=1}^{m} p_i(\mathbf{w})^{y_i}(1 - p_i(\mathbf{w})^{1-y_i}$$

for the likelihood (don't confuse this L with loss - the letter L is used for both likelihood and loss in statistical learning.) Since the log function is monotone increasing, we can maximize the log-likelihood $\ell(\mathbf{w}|\mathbf{y}) := \log(L(\mathbf{w}|\mathbf{y})$:

$$\text{argmax}_{\mathbf{w}\in\mathbb{R}^{d+1}} L(\mathbf{w}|\mathbf{y}) = \text{argmax}_{\mathbf{w}\in\mathbb{R}^{d+1}} \ell(\mathbf{w}|\mathbf{y}) .$$

Write for convenience $\beta_i = \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle$. Let's expand the log-likelihood:

$$\begin{aligned}
\ell(\mathbf{w}|\mathbf{y}) &= \sum_{i=1}^{m} \left[y_i \log p_i(\mathbf{w}) + (1-y_i)\log(1-p_i(\mathbf{w}))\right] = \\
&= \sum_{i=1}^{m} \left[y_i \log\left(\frac{e^{\beta_i}}{1+e^{\beta_i}}\right) + (1-y_i)\log\left(\frac{1}{1+e^{\beta_i}}\right)\right] = \\
&= \sum_{i=1}^{m} \left[y_i\beta_i - \log\left(1+e^{\beta_i}\right)\right] = \\
&= \sum_{i=1}^{m} \left[y_i\langle \mathbf{x_i}\,,\,\mathbf{w}\rangle - \log\left(1+e^{\langle \mathbf{x_i}\,,\,\mathbf{w}\rangle}\right)\right]
\end{aligned}$$

So applying the maximum likelihood principle means that, based on the training sample S, we choose the function $h \in \mathcal{H}^d_{logistic}$, and, equivalently, the vector $\mathbf{w} \in \mathbb{R}^{d+1}$, by finding

$$\hat{\mathbf{w}} := \underset{\mathbf{w}\in\mathbb{R}^{d+1}}{\operatorname{argmax}} \sum_{i=1}^{m} \left[y_i\langle \mathbf{x_i}\,,\,\mathbf{w}\rangle - \log\left(1+e^{\langle \mathbf{x_i}\,,\,\mathbf{w}\rangle}\right)\right].$$

- Remember that in linear regression we first derived the learner using Empirical Risk Minimization (ERM) on the square loss $L(y,\hat{y}) = (y-\hat{y})^2$, and then saw that the same learner (least squares learner) can be obtained using the maximum likelihood principle if we assume additive Gaussian errors? Well, here in logistic regression we can go the other way around. We just derived the logistic regression learner using the maximum likelihood principle assuming y_i are Bernoulli distributed. We can in face arrive at the same learner if we applied the ERM using a strange loss (see "Understanding Machine Learning" 9.3).

9.4 Computational implementation

- Computationally, how do we maximize the log-likelihood? One of the advantages of choosing the logistic function π for ϕ above, is that the resulting log-likelihood is a **concave** function of the optimization variable $\mathcal{V}w$, so we can instead look for the minimum of the minus log-likelihood, which is convex. There are general ways to minimize convex functions - we will have a whole lecture on that. But, since logistic regression is so famous and so useful, people have studied in great detail how to optimize numerically (i.e on a computer) this particular log-likelihood. While there is no closed form for the maximizer $\hat{\mathbf{w}}$, and there is an iterative algorithm that usually converges quickly to $\hat{\mathbf{w}}$ based on **Newton-Raphson** iterations. It is recommended to read about this algorithm in "Elements of Statistical Learning 2nd ed." 4.4.1. In practice, unless we have a special reason to write our own solver, we'll fit logistic regression models with a learning software package. You should be familiar with `glmnet`, a package for the statistical programming language `R` (there are python bindings) that can solve large logistic regression problems very quickly. It also supports ℓ_1 **regularized logistic regression**, which is a famous, useful classifier we will see later in the course. In other software packages, you might find logistic regression under a function called `glm`. GLM stands for "generalized linear model" - logistic regression is one kind of something called a generalized linear model (more information in statistics courses.)

9.5 Interpretability

One important property that a classifier may or may not have is "explainability" or "interpretability". In general, there are two different things people mean when they talk about interpretability of a learning algorithm:

9.5.1 Which features were important

When working with $\mathcal{X} = \mathbb{R}^d$, on real data, we gather many features. We might think that some of them may not be important for prediction. So, after we ran the learner on the training sample and obtained the rule $h_S \in \mathcal{H}$, we may ask which features were used by the learner and which were not.

9.5.2 Why was this label predicted?

When a classifier predicts a label y_i for a sample $\mathbf{x}_i$, we may ask: why? Why was this label predicted? This may be important, for example, if the prediction was wrong and we would like to understand why it was wrong. (Indeed, debugging a machine learning algorithm in production – understanding why some crucial predictions are wrong – is a difficult and often overlooked task.) It's like "lifting the hood" and looking into the inner workings of the classifier.

9.5.3 Example: Interpretability of linear regression

Linear regression, for example, is a learner that is highly interpretable. When we ask "which features were important" we just look for the large entries of the fitted vector $\mathbf{w}$. When we ask "why was the label y_i predicted" we just look at the weights we fitted at the training stage. We know the weights. We know which weight is for which feature. So we have a good understanding of the prediction $\mathbf{x} \mapsto \langle \mathbf{x}, \mathbf{w} \rangle$. We can see that, for example, a large weight was multiplied by a large feature (an entry of $\mathbf{x}$) and had a large effect on the predicted label. And so on.

9.5.4 Interpretability of logistic regression

The same is true for logistic regression. When we have a prediction $\pi(\langle \mathbf{x}, \mathbf{w} \rangle) \in [0, 1]$ for a sample $\mathbf{x}$, we can ask "why this prediction", and get a satisfying answer. We can look at the weight vector $\mathbf{w}$ that we fitted in the training, and see which large (positive or negative) entries of $\mathbf{x}$ were multiplied by large weights, and whether negative and positive terms canceled out in the inner product, and so on. And when we ask "which features were important", we again look at the entries of the fitted weight vector $\mathbf{w}$. If some entries in the weight vector $\mathbf{w}$ are very small, we deduce that these features did not help predict the labels in the training set.

So from now on, when we look at a new classification, we can ask whether it's interpretable or not, and if it is, ask how to explain decisions / predictions made on test samples.

9.6 How to make predictions on a new sample: working with estimated class probabilities and choosing the cutoff

One important remaining issue is that in logistic regression we gave training data with samples in $\{0, 1\}$ but produced an hypothesis that takes values in $[0, 1]$. These values are "estimated class probabilities" - the "probability" or "likelihood" that the sample is in class 1 and not class 0. How to turn this into a class prediction in $\{0, 1\}$?

The following is important for logistic regression, as well as any other classification algorithm that produces an "estimated class probability" instead of a class. We actually like it when a classifier gives us an "estimated class probability" because we don't just get a prediction of class $\{0,1\}$, but also some estimate of how **certain** the classifier is. For example, a classifier can predict class 1, but if we knew that the estimated class probability is 0.6 (so that the classifier isn't really sure), we might do something different than if, say, the estimated class probability was 0.99.

This is especially important when we think about Type-I and Type-II errors. Recall that in most classification problems, of the two errors the classifier can make, there is one that we would really like to avoid. Having an estimated class probability can help us "tune" the prediction so have more avoidance of one kind of error, and be more relaxed about the other kind of error.

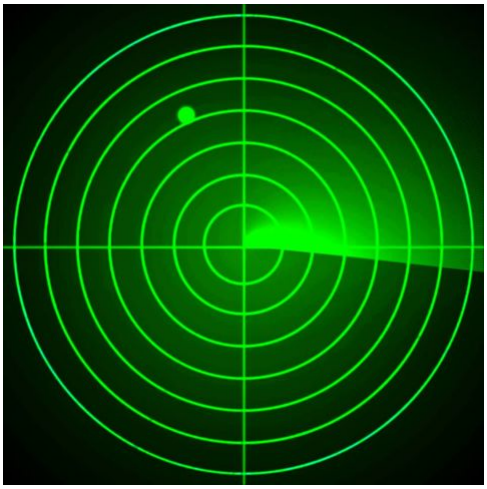
So suppose we trained a logistic regression and produced some $h \in \mathcal{H}^d_{logistic}$ Suppose that the label 0 is the **negative** label and 1 is the **positive** label, (with meaning chosen such that **false-positive**, namely the Type-I error, is the error we really don't want to make, of the two possible errors). We need to choose a cutoff $\alpha \in [0,1]$ such that the actual class prediction we make will be

$$\hat{y} := \begin{cases} 1 & h(\mathbf{x}) > \alpha \\ 0 & h(\mathbf{x}) \leqslant \alpha \end{cases}.$$

How shall we choose it? There's an important **trade-off** here. If we set α to be very high (near 1) we will tend to predict 0. So we will be **less** likely to have false-positives (which is the error we really don't want to make) - and that's good - but at the same time will be **more** likely to have false-negatives (which are positive samples that we misclassified as negatives). So if we set α too high, we might have very low false-positive rate but "miss" (misclassify) most of the positive samples. On the other extreme, if we set α to be very low (near 0) we will tend to predict 1. So we will be **more** likely to have false-positives - and that's bad - but at the same time will be **less** likely to have false-negatives. So if we set α too low, we might have a high false-positive rate but "catch" (correctly classify) most of the positive samples.

So we see that in changing α in $[0,1]$ we are moving along a trade-off between the chance to make a Type-I error (make a false-positive) on the one hand, and the chance to "catch", or correctly classify, a positive sample. This was first studied in the World War II, when radar was invented. The designer of the radar had to choose when to put a green dot on the radar, indicating a target detected there. Sometime radar waves would bounce off back from clouds or birds, and the designer had to choose a **threshold** α. If the radar pulse returning is stronger than α, the radar screen would show a green dot. If weaker than α, no dot. Now, if α is set too low (say $\alpha = 0.1$), the screen would be full of a thousand green dots - since any bird or cloud (with, say, $h(\mathbf{x} = 0.2)$ would be classified as **positive**, a target. So that the radar will be full of false positives, false targets, and will be useless. On the other hand, if α is set too high (say $\alpha = 0.9$) then enemy airplanes (with, say, $h(\mathbf{x} = 0.8)$ will not appear on the screen, since they will be classified by mistake as birds, and the radar again would be useless.

The radar engineers developed a way to look at this tradeoff, which is still popular to this day in machine learning. After training the logistic regression model (so we have $h \in \mathcal{H}^d_{logistic}$) we make a grid of values of α in $[0,1]$. For each value of α we create a classifier by thresholding h at α, and calculate the number of Type-I and Type-II errors the classifier makes - preferably over a test sample that was not used for training the logistic regression (why?). We plot a parametric curve of TPR (true positive rate) against FPR (false positive rate) when α is the parameter. This curve is called the **Receiver Operating Characteristic (ROC)** curve. It is continuous, increasing and goes from $(0,0)$ in the FPR-TPR plane (for $\alpha = 0$ we classify



everything as negative, so no false positives and not true positives) to $(1, 1)$ (for $\alpha = 1$ we classify everything as positive, so false positive rate is 1 - we make every possible Type-I error - and also true positive rate is 1 - we "catch" all the positive samples).

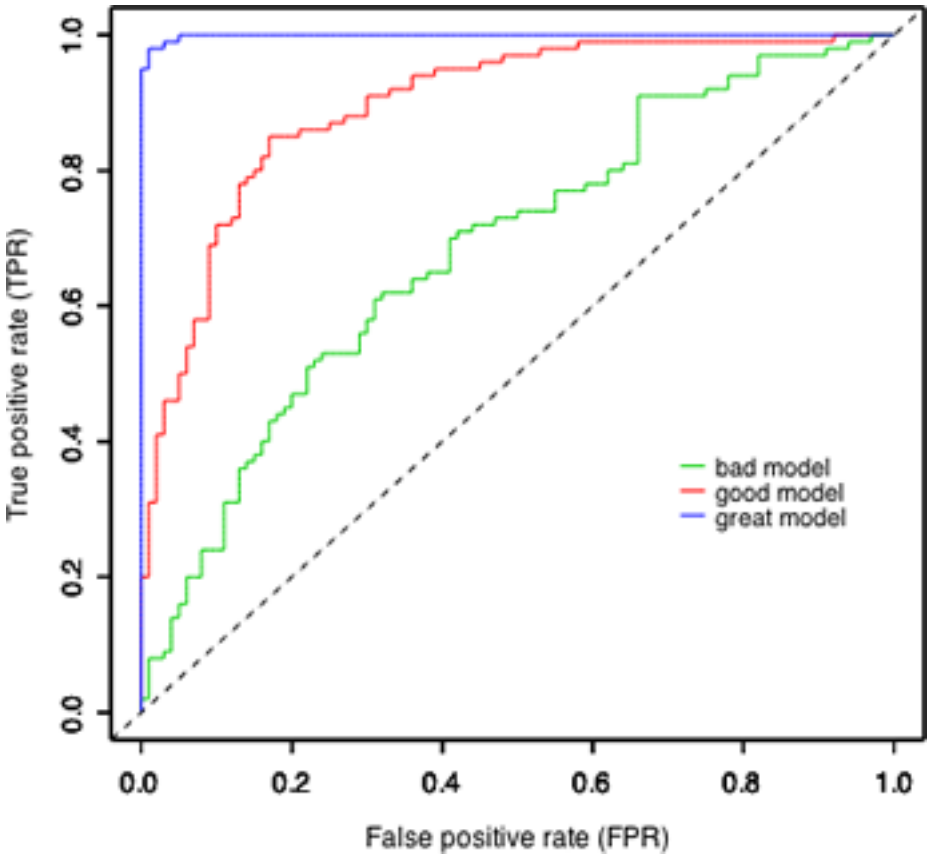


Figure 16: ROC curves of three trained models (source:jxieeducation.com)

Convince yourself that if the ROC curve is a linear line from $(0, 0)$ to $(1, 1)$, the classifier is just a random guess. If a classifier has an ROC curve that is closed to this linear line, it's a poor classifier. Now convince yourself that if the ROC curve rises sharply from $(0, 0)$, for

example makes a "jump" to $(FPR = 0.1, TPR = 0.9)$, it's a good classifier - we are able to correctly detect 0.9 of the positive samples at the price of 0.1 false positive rate.

Plotting the ROC curve of a classifier has a few different uses:

- **Tuning α.** It allows us to see the tradeoff, provided by the classifier, between Type-I errors and correct detection of positive samples, so we can choose the tuning of α we would like to work with for the actual prediction - for using logistic regression to actually classify new samples.

- **AUC - a performance measure for the tradeoff itself**. It allows us to define a performance measure for the logistic regression classifier known as **Area Under the Curve** (AUC). This performance measure evaluates the prediction rule h we chose without having to decide on α - it measures the quality of the **tradeoff** provided by h, a tradeoff from which we must choose a specific point in order to actually classify new samples. AUC is simply the **definite integral** of the ROC curve on the segment $[0, 1]$ - the area under the AUC curve. As mentioned above, AUC around $1/2$ means that h is poor - more specifically, that the **tradeoff** provided by h is poor. AUC is bounded from above by 1, so an AUC close to 1 means h offers an excellent trade-off, and in this case we expect to be able to find a cutoff α that gives a classifier with very few false-positives and very high detection rate (true positive rate).

- **Comparing candidate rules.** Suppose that we have two or candidate rules h_1, h_2 (or more). For example, maybe we trained logistic regression on the same training sample with different features, or maybe we have one rule that comes from logistic regression and a different rule (that also gives estimated class probabilities), and we're wondering which one to use. Now we have a problem - we can't turn h_i into an actual classification rule without choosing a cutoff α_i, but would like to compare h_1 to h_2 without committing to a cutoff - to compare the tradeoff offered by h_1 to that offered by h_2. It is very useful here to plot the two ROC curves of h_1 and h_2 on a single axis - and visually compare the tradeoffs they offer.

As you've probably understood, the need to choose a threshold α and the ROC curve are **not** special to logistic regression. We need to choose α, and can plot ROC curve, for any classification algorithm that produces "estimated class probabilities" rather than class predictions in $\{0, 1\}$.

9.7 Summary

- Hypothesis class:
$$\mathcal{H}^d_{logistic} = \{\mathbf{x} \mapsto \pi(\langle \mathbf{x}\,,\, \mathbf{w}\rangle)\}\ .$$

- Learning principle for training model (choosing $f \in \mathcal{H}$): Maximum Likelihood

- Computational implementation of learning principle: Specialized iterative method based on Netwon-Raphson iterations, or a general convex solver (pay attention to effective rank of regression matrix - just like in linear regression. Near-singular regression matrix X will cause numerical problems.)

- How to make predictions on new samples: with $\mathbf{w}$ the fitted weight vector:
$$\hat{y} := \begin{cases} 1 & \pi(\langle \mathbf{x}\,,\, \mathbf{w}\rangle) > \alpha \\ 0 & \pi(\langle \mathbf{x}\,,\, \mathbf{w}\rangle) \leqslant \alpha \end{cases},$$

where $0 < \alpha < 1$ is a **cutoff** chosen to give a good tradeoff between false positive rate and true positive rate (a point on the ROC curve).

- Interpretable: Yes - using fitted regression coefficients
- Estimates class probabilities: Yes
- Family of models: Yes, as we add or remove features from regression matrix X. Also possible to add **regularization** to control bias-variance for a fixed regression matrix X - see future lectures.
- Time complexity for training, and for predicting on a new sample:
- How to store trained model: store the regression weight vector $\mathbf{w}$
- When to use: Always try this, especially when classes are more or less balanced

10 Nearest Neighbors

And now for something completely different. **Nearest neighbor** classification is a popular, simple, effective learner that's simple to understand - but (depending on the dimension and training sample size) may not so easy to implement. It is sometimes surprisingly effective. And it is very different from all the other methods in this lecture.

10.1 No hypothesis class

Wait, what? Yes, this classifier is **not** based on our paradigm of hypothesis class, learning principle, and all that. **A nearest neighbor classifier has no training stage.**

Every other classification algorithm in this lecture has an hypothesis class, and a training stage in which we use the training sample S to choose some $h_S \in \mathcal{H}$. **After the training stage, we can in principle discard the training sample** and just store the chosen hypothesis h_S in order to make predictions on new samples.

Nearest neighbors classifiers are different. There is no hypothesis class and no training stage. We must keep the entire training sample S to make new predictions: the algorithm just tells us how to use S to make a prediction on a new sample.

10.2 Prediction with k-nearest neighbors

The algorithm need two parameters: an integer value k, and a distance function ρ on $\mathbb{R}^d$. We can use the square Euclidean norm $\rho(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|$. Or we can use a **weighted** square norm that gives more importance to some features: $\rho(\mathbf{x}, \mathbf{z}) = \sum_{j=1}^{d} \omega_j (x_j - z_j)^2$, for some fixed weight vector $(\omega_1, \ldots, \omega_d)$ and where x_j is the j-th coordinate of the vector $\mathbf{x} \in \mathbb{R}^d$.

So, what's the algorithm? Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$ be our training sample and let $\mathbf{x}$ be the test sample we wish to classify.

The algorithm finds the k training samples closet (with respect to the distance ρ provided) to $\mathbf{x}$ and predicts by majority vote of the labels of these k (training samples) nearest neighbors.

Formally,

- Let $\pi = (\pi_1, \ldots, \pi_m)$ be a permutation of $(1, \ldots, m)$ such that

$$\rho(\mathbf{x}, \mathbf{x}_{\pi(1)}) \leqslant \rho(\mathbf{x}, \mathbf{x}_{\pi(2)}) \leqslant \ldots \rho(\mathbf{x}, \mathbf{x}_{\pi(m)}) .$$

  So that $\mathbf{x}_{\pi(1)}, \ldots, \mathbf{x}_{\pi(k)}$ are the k-nearest-neighbors of $\mathbf{x}$ among the samples in S, with respect to the distance distance ρ.

- The predicted class is

$$\text{argmax}_{y \in \{0,1\}} \sum_{i=1}^{k} \mathbf{1}_{y_{\pi(i)}=y}$$

10.3 Choosing k

In fact, nearest-neighbors is a family of learning algorithms, one for each k. The possible values for k are between 1 (1-nearest-neighbor) and m.

What happens at the extremes $k = 1$, $k = m$?

- When $k = 1$, the test point just copies the label of its nearest neighbor in the training set. The decision boundaries are known as **Voronoi cells**. The classifier has very low bias and very high variance (why?)

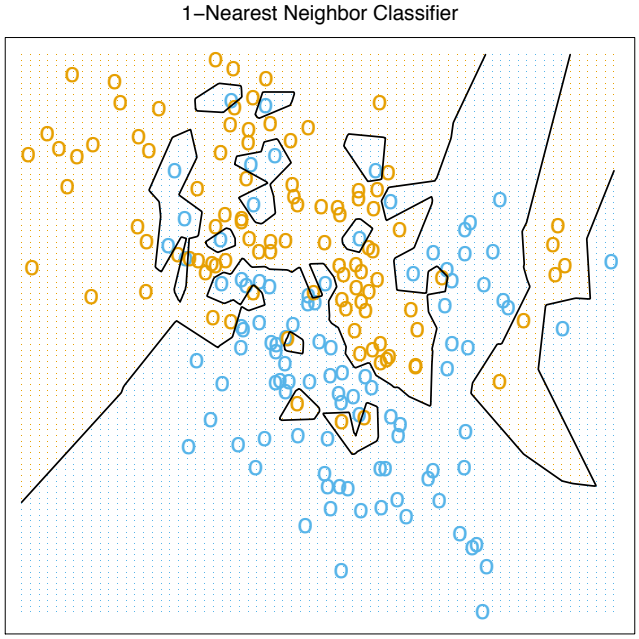


Figure 17: Decision boundary for a 1-NN classifier (source: ESL)

- When $k = m$, the classifier predicts a single class for every point in $\mathbb{R}^d$ - the majority vote of all the training sample. Bias is (very, very) large, variance is zero.

And in less extreme values of k, what do we get?

- When $k = 3$ (say) the prediction is a majority vote of the 3 nearest neighbors of the test sample. (It's good to take an uneven k so that there will always be a majority vote.) The

classifier has low bias and high variance. To see why, think how many training samples will need to change their class labels in order for a test sample to receive a different prediction. If the vote for some test sample is 2-1, it's enough that **one** test sample will change its label, and the prediction will change.

- When $k = m/3$ (say) the prediction is a majority vote of **a third** of the training sample points. The classification rule is constant on large parts of $\mathbb{R}^d$. Bias is high, variance is low. (Again, think how many training samples need to change their class for the prediction to change.)

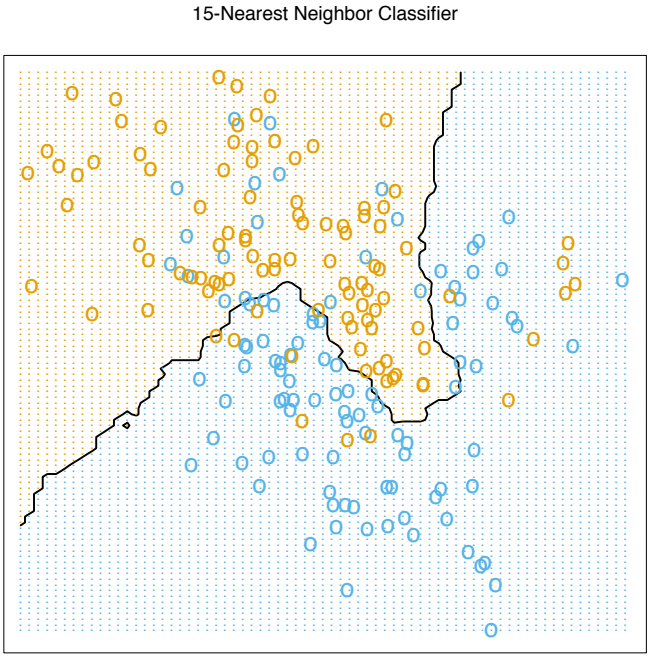


Figure 18: Decision boundary for a 15-NN classifier (source: ESL)

As we change k we change the bias-variance tradeoff. Typically we will see something like this for the test error of a k-NN classifier, as misclassification error is plotted over k:

We will talk about this in more detail later in the course. This is an example of the **general phenomenon** for most learning algorithms, which we have mentioned several times before:

10.4 Computational implementation of k-nearest neighbors

Implementing a k-nearest-neighbors classifier is very easy on small datasets, and not so easy when the dimension d and/or the sample size m are large. There are generally three types of implementation approaches:

- **Brute force implementation**: We keep the entire training sample S in storage during the entire prediction process. For each new test sample $\mathbf{x} \in \mathbb{R}^d$, we calculate $\|\mathbf{x} - \mathbf{x}_i\|$ for all $1 \leqslant i \leqslant m$ and partially sort to find the k smallest distances. (What's the time complexity? And how much space is needed?)

- **Exact nearest neighbors search with preprocessed data structure**: If ρ is the Euclidean norm, algorithms such as `kd-tree` and others can be used to pre-process the

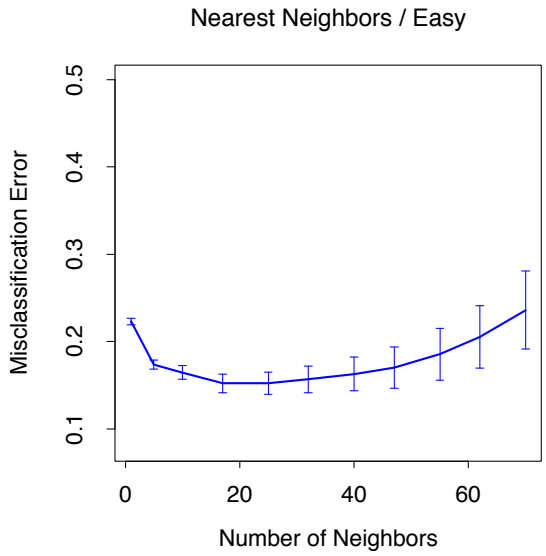


Figure 19: Test error of a k-NN classifier over k (source: ESL)

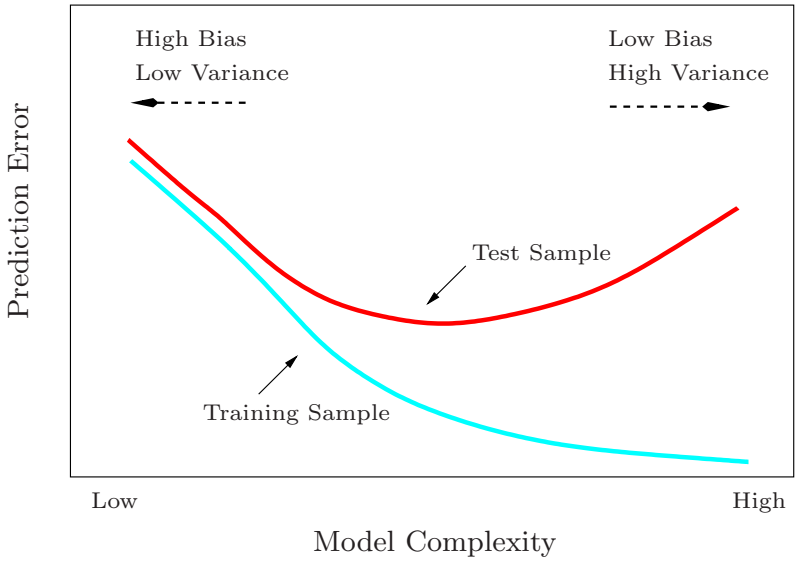


Figure 20: General shape of a bias-variance tradeoff (source: ESL)

training sample and construct a **special data structure**. (This is kind of a training stage, even though we don't come up with a hypothesis, just cache computations regarding the training sample so our prediction will be faster.) We don't need to keep the training sample S, just the data structure. Then at prediction time the data structure is used to quickly locate the k nearest neighbors of the test sample $\mathbf{x}$.

- **Fast randomized nearest neighbors search**: beyond our scope in this course.

10.5 Other sample spaces

Nearest neighbors is the only classification algorithm we learn in this lecture, which can work on sample spaces other than $\mathcal{X} = \mathbb{R}^d$. Indeed, we just need a distance function ρ on $\mathcal{X}$. The implementation may be challenging, as all the tricks that are available to speed up computation in $\mathbb{R}^d$ are not available generally.

10.6 Summary

- Hypothesis class $\mathcal{H}$: None ("model free")
- Learning principle for training model (choosing $f \in \mathcal{H}$): none - no model
- Computational implementation: Brute force nearest neighbor search; kd-tree and similar exact methods which speed up prediction on new samples; fast randomized nearest neighbors
- How to make predictions on new samples: if using brute force, calculate distance to every point in sample
- Interpretable: No
- Estimates class probabilities: No (unless using with many neighbors)
- Family of models: Yes, according to k (using k-nearest neighbors)
- Time complexity for training, and for predicting on a new sample:
- How to store trained model: no trained model. Must store the entire training sample S (or preprocessed data structrue)
- When to use: Always try when implementation possible. Especially when you're out of ideas.

11 Classification Trees

One of the most useful algorithms for classification **and** regression known is the **CART Random Forest.** CART stands for Classification and Regression Trees. A random forest, is, well, a large collection of trees. Over a few lectures, we will to develop the full CART Random Forest algorithm in three steps:

1. **Growing the classification tree** (A regression tree is very similar - if you understand classification trees you understand regression trees.)
2. **Pruning the tree** using a regularization principle for precise control over bias-variance tradeoff.
3. **Bagging trees** to create a random forest.

As we will see, are different ways to **grow** and **prune** a classification tree. We will see a method for growing and pruning called CART. This lecture will focus on the first step. We'll see **pruning** and **bagging** in later lectures.

So let's grow a CART classification tree. It is an interesting classifier in and of itself, but the real reason to learn it is to understand the much more powerful (and popular) Random Forest classifier.

In this section we'll use the label set $\mathcal{Y} = \{0, 1\}$.

11.1 Tree-induced, axis-parallel partitions of $\mathbb{R}^d$

We've already seen two classifiers that use piecewise-constant prediction rules: half-spaces and SVM. In both the hypothesis class is a half-space, where we predict class 1 (say) on one side of a hyper-plane, and 0 (say) on the other side.

Now we would like to use more complicated piecewise-constant prediction rules (more complicated hypotheses.) Let's consider a rule that partitions the sample space $\mathbb{R}^d$ into **axis-parallel boxes, or "hyper-rectangles"**, and it each box we predict either 1 or -1. The learner's task would be to use the training sample to "chop" the sample space $\mathcal{X}$ into a disjoint union of axis-parallel boxes, and to assign a class prediction to each box.

To make our hypothesis class a little smaller and a little simpler, let's focus on disjoint unions of boxes that are obtained by iteratively chopping an existing box into two smaller boxes along one of the axes. More specifically:

- We start with the whole sample space $\mathbb{R}^d$
- We chop $\mathbb{R}^d$ into two axis-parallel "boxes" (a half-space is also a box in our terminology) by selecting one of the d coordinates, say the coordinate i_1 ($1 \leqslant i_1 \leqslant d$) and a value $t_1 \in \mathbb{R}$. The two boxes will be $B_+ = \{\mathbf{x} \in \mathbb{R} \,|\, x_{i_1} > t_1\}$ and $B_- = \{\mathbf{x} \in \mathbb{R} \,|\, x_{i_1} \leqslant t_1\}$. When then chop B_+ by again selecting a coordinate $i_{2,1}$ and a value $t_{2,1}$ (such that $t_{2,1} > t_1$ so we are chopping inside the box B_+) and Define $B_{++} = \left\{\mathbf{x} \in B_+ \,|\, x_{i_{2,1}} > t_{2,1}\right\}$ and $B_{+-} = \left\{\mathbf{x} \in B_+ \,|\, x_{i_{2,1}} \leqslant t_{2,1}\right\}$. Similarly we choose a coordinate $i_{2,2}$ and a value $i_{2,2}$ and chop B_- into B_{-+} and B_{--}. Now we have $\mathbb{R} = B_{++} \biguplus B_{+-} \biguplus B_{-+} \biguplus B_{--}$. We can stop here, or we can chop any (or all) of the existing boxes into smaller axis-align boxes. We may, for example, choose to only chop B_{++} further, and leave the others as they are. Finally we stop chopping, and are left with a partition of $\mathbb{R}$ into a disjoint union of axis-aligned boxes. We call such a partition a Tree Partition.

  Note that the partitions obtained this way are special - most partitions of $\mathbb{R}^d$ into axis-aligned boxes are not Tree Partitions, namely, cannot be constructed by such a top-down iterative chopping procedure.

11.2 The Regression Tree hypothesis class

The hypothesis class $\mathcal{H}_{CT}$ we will consider consists of piecewise-constant functions, that assign a class prediction (1 or 0) to each box in a Tree Partition. Unless we restrict it somehow, the class contains all piecewise-constant functions supported on all Tree Partitions of $\mathbb{R}^d$ (to any number of boxes). Formally, for a Tree Partition $\mathbb{R}^d = \biguplus_{j=1}^N B_j$ of $\mathbb{R}^d$ into N boxes, and label assignments $c_j \in \{0, 1\}$ ($j = 1, \ldots, N$) assigning label c_j to box B_j, the hypothesis $h \in \mathcal{H}_{CT}$ is a function $h : \mathbb{R}^d \to \{0, 1\}$ defined by

$$h(\mathbf{x}) = \sum_{j=1}^{N} c_j \mathbf{1}_{B_j}(\mathbf{x})$$

where $\mathbf{1}_{B_j}$ is the indicator function of B_j.

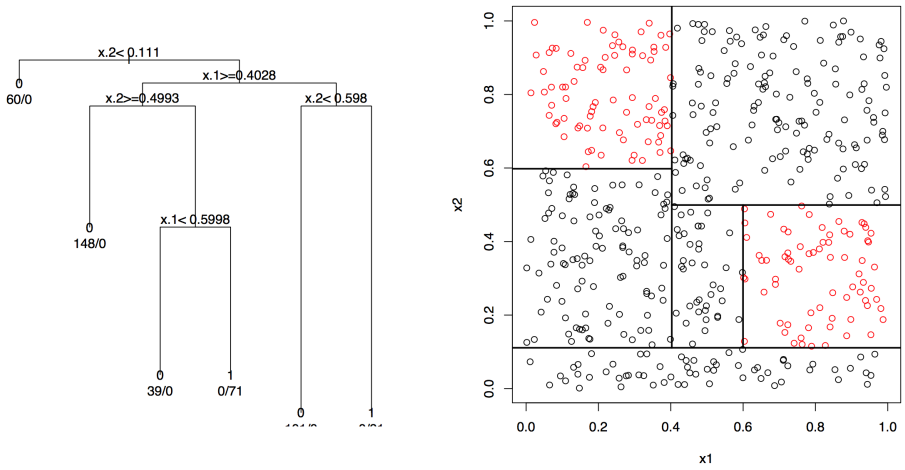


Figure 21: A tree and its induced partition in $\mathbb{R}^2$

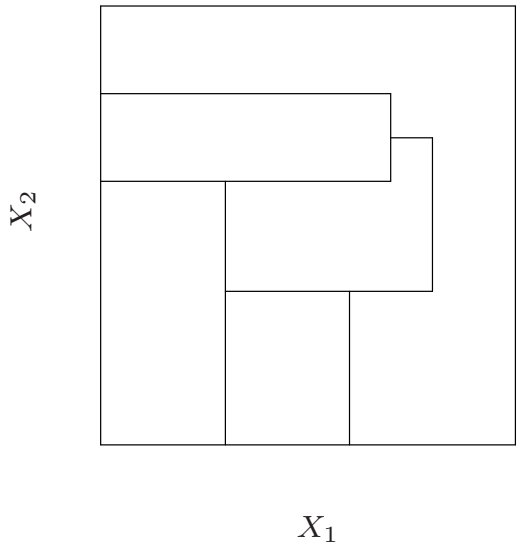


Figure 22: A partition into axis-aligned boxes that is **not** a tree partition. (Source: ESL)

11.3 Any hypothesis in $\mathcal{H}_{CT}$ corresponds to a Decision Tree, and vice-versa

Let's talk about something else for a second, which we turn out to be closely related to the hypothesis class $\mathcal{H}_{CT}$ we just defined.

Outside computer science and machine learning, there is a time-honored way to classify, or to **make decisions.** Suppose someone comes into a hospital emergency room. The first step of triage is to determine - fast - whether they are in a life-threatening medical emergency, or else they can wait in line and receive treatment in a little while. The triage

uses a sequence of yes/no questions, such as: Is the patient conscious yes/no?
- If not conscious: Classify as **emergency**
- If yes conscious: Is the patient's pulse < 40 beats per minute?

  – If yes, pulse < 40: Classify as **emergency**

  – If no, pulse > 40: Is the patient's pulse > 130 beats per minute?

    * If yes, pulse > 130: Is the patient's systolic blood pressure $< 80mmHg$?

      · If yes, blood pressure $< 80mmHg$, classify as **emergency**

      · If no, blood pressure $> 80mmHg$: Is the patient's systolic blood pressure $> 140mmHg$?

        - If yes, blood pressure $> 140mmHg$, classify as **emergency**

        - If no, blood pressure $< 140mmHg$, classify as **not emergency**

    * If no, pulse < 130: classify as **not emergency**

This is a **decision tree** that uses three features: **conscious** (a binary categorical feature), **pulse** (a numerical feature) and **blood pressure** (also a numerical feature). See if you can we write a diagram for this decision tree in the shape of a tree, where every node is a question, and every leaf is a decision / classification. The root of the tree is the first question ("conscious yes/no?").

Now observe that **every function in our Classification Trees hypothesis class $\mathcal{H}_{CT}$ is equivalent to a decision tree**. In the notations of the generic example above, the first question is: "$x_{i_1} > t_1$ - yes/no?"; If yes, we ask the second question "$x_{i_{2,1}} > t_{2,1}$ - yes/no?". If no, we ask the second question "$x_{i_{2,2}} > t_{2,2}$ - yes/no?". And so on, until there are no more splits and we have reached a box over which the function in $\mathcal{H}_{CT}$ is constant. If the constant value is $+1$, we classify / predict class 1. If the constant value is 0, we predict class 0.

This is why our hypothesis class is called - classification **trees**.

11.4 How *not* to grow a classification tree

Having defined our hypothesis class, the next question is of course - what learning principle shall we use, namely, how shall be select $h_S \in \mathcal{H}_{CT}$ based on a training sample S. We have to work with some loss - for example, let us work with misclassification loss - simply counting misclassifications. But as you will see, in what follows you can replace this loss with any other loss function you prefer.

Let's start from the end. Suppose that we have already somehow decided to use a certain Tree Partition of $\mathbb{R}^d$ that consists of N disjoint boxes, $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$. Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ be our training sample. If the predicted label assigned to box B_j is $\hat{y}(B_j) \in \{\pm 1\}$ then the number of misclassification errors that are incurred by the training samples that fall inside B_j is of course $\sum_{\mathbf{x}_i \in B_j} \mathbf{1}_{y_i = \hat{y}(B_j)}$. Let's apply the ERM principle - and try to minimize the misclassification errors on our training set.

For any box B in a tree partition, and a sample $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$, and a label $y \in \{0, 1\}$, define

$$P_y^S(B) := \frac{1}{n_S(B)} \sum_{\mathbf{x}_i \in B} \mathbf{1}_{y_i = y} \,,$$

where $n_S(B) = \left| \{1 \leqslant i \leqslant m | \mathbf{x}_i \in B\} \right|$ is just the number of training samples in S that fall inside the box B.

The label that will minimize the empirical risk for those training samples that fall inside the box B is of course the **majority vote**

$$\hat{y}_S(B) = \underset{y\in\{0,1\}}{\operatorname{argmax}} P_y^S(B)$$

- checking which label has more samples inside B - this is the label for which we make less misclassification errors inside the box B.

So, adopting the ERM principle, the labeling assignment, for a given tree partition $\mathbb{R}^d = \biguplus_{j=1}^N B_j$, the assignment that minimizes the misclassification for the training sample S is given by labeling the box B_j with $\hat{y}_S(B_j)$.

It follows that, for our training sample S, for each Tree Partition there is a unique label assignment - and therefore a unique classification tree (hypothesis) $h \in \mathcal{H}_{CT}$ that minimizes the empirical risk. It seems, therefore, that all we have to do is look for the tree that minimizes ERM $h \in \mathcal{H}_{CT}$, namely find

$$\operatorname{argmin}_{h\in\mathcal{H}_{CT}} L_S(h)$$

where $L_S(h)$ is the misclassification error of the candidate classification tree h on the training sample S.

But wait. If we don't limit the number of levels of the tree, we know which tree will minimize the empirical risk - the tree with so many levels that each sample in $\mathbf{x}_i \in S$ finds itself alone in a box, and that box is (of course) labeled with the label y_i. Convince yourself that such a classification tree has empirical risk $L_S(h) = 0$. But this tree won't generalize well to new samples. (Why? - this is the very definition overfitting: we used the fact that $\mathcal{H}_{CT}$ is very large to find $h \in \mathcal{H}_{CT}$ that fits the training sample S **exactly**. So our learner has huge **variance**.)

The solution? We should of course limit the number of levels in the classification tree. Let $\mathcal{H}_{CT}^k$ denote the hypothesis class of classification trees with at most k levels. Note that we now have **a family of hypothesis classes** - one for each k - not just one hypothesis class. Convince yourself that k controls the size of the hypothesis class and therefore controls a bias-variance tradeoff:

- For small k, $\mathcal{H}_{CT}^k$ is small, and therefore the ERM learner over $\mathcal{H}_{CT}^k$, namely the learner that chooses $h_S \in \mathcal{H}_{CT}^k$ with the lowest empirical risk, will have very high bias (only very simple functions on $\mathbb{R}^d$ can be approximated by classification trees with just a few levels) and very low variance (the boxes are very large, so labels assigned to each box are based on majority vote of typically many training samples. So changing a few training samples will barely change the selected hypothesis h_S.)

- For large k, $\mathcal{H}_{CT}^k$ is large, and therefore the ERM learner over $\mathcal{H}_{CT}^k$ will have low bias (we can approximate even complicated functions on $\mathbb{R}^d$ with classification trees with many levels), and high variance (the boxes are small, so labels assigned to each box are based on majority vote of typically just a few samples. So changing a few training samples could change the label assignment of many boxes.)

Ok, so it seems we have a solution: We will choose a reasonable value for k, namely, select a specific hypothesis class among the family of all possible classification tree hypothesis classes, and use ERM, namely return

$$\operatorname{argmin}_{h\in\mathcal{H}_{CT}^k} L_S(h)\,.$$

11.5 How to grow a classification tree

Houston, we have a problem. How do we find the minimizer $\text{argmin}_{h \in \mathcal{H}^k_{CT}} L_S(h)$ computationally? Namely, how do we implement the ERM principle over $\mathcal{H}^k_{CT}$? So far, the examples we've seen of learners based on the ERM learning principle lead to computationally tractable problems: linear regression was based on ERM, and we had a closed form expression for the minimizer. Half-space classifier was based on ERM, and lead to a simple convex optimization problem. But now? the search space $\mathcal{H}^k_{CT}$ is large (how many different hypothesis are in $\mathcal{H}^k_{CT}$?) and has no Euclidean or other structure that can be used. To find ERM, it seems we would have to use brute force search, which is infeasible. (In fact, one can prove that implementing ERM on $\mathcal{H}^k_{CT}$ is an NP-hard problem with respect to the training sample size[16].

What do we do? this is the first time that we come fact to face with the bitter truth that while the ERM principle is nice, it is often impossible to implement - especially when the hypothesis class on which we are working has no Euclidean structure.

So we must resort to **heuristics**. This is where it gets interesting. While the definition of classification trees, and of $\mathcal{H}^k_{CT}$, as well as how to assign prediction labels to boxes in a Tree Partition given a training sample, are all canonical, there are several different heuristic approaches to how to proceed and "grow a classification tree" - namely, choose $h_S \in \mathcal{H}^k_{CT}$ - in practice.

One approach, that came out of the statistical learning community, is known as **Classification and Regression Trees** (CART). It is described in detail in "Elements of Statistical Learning 2nd ed." section 9.2. Another set of approach came out of the computer science machine learning community, are known by the weird names **ID3**, **C4.5** and **C5.0**. The simplest of these, ID3, is described in "Understanding machine learning" section 18.2. While these approaches differ in details, all of these are basically greedy heuristic that build a tree top-down and then "prune" it (merge some of the boxes back into larger boxes).

Here, we will develop the CART heuristic, which is quite similar to C5.0, and has very well polished and efficient software implementation in software packages.

11.6 Growing classification Trees a-la CART

CART consists of two stages: **growing** the tree, which results in a tree that is a little too large, and then **pruning** the tree to bring it down to the most effective size. We start with how to grow the tree and will discuss pruning in a later lecture.

Suppose we chose k to be the maximal tree depth (we will discuss how to choose k in a later lecture). The heuristic to grow a full classification tree with at most k levels will proceed top-down, starting from $\mathbb{R}^d$ and progressively chopping each box into two boxes. We don't chop a box under one of two conditions: (i) the maximum number of levels k has been reached; or (ii) the box has reached a minimal number of training samples that was pre-determined. (As we will see, it makes no sense to chop a box with very few training samples in it.) If no minimal number of sample was specified, we anyway have to stop if a box only has one training sample. Each existing box B is chopped into two boxes as follows:

- for each coordinate $i \in \{1, \ldots d\}$, and each potential chopping value $t \in \mathbb{R}$, let $g_i(t)$ be the the lowest empirical misclassification risk (with respect to the training sample S) incurred

[16] By reduction from "three-dimensional matching", see Hyafil and Rivest, "Constructing Optimal Binary Decision Trees is NP-Complete", *Information Processing Letters* **5**(1), 1976

by chopping the box B at value t along coordinate i. Calculate the value $g_i(t)$ for each i and t as follows:

  - For each $t \in \mathbb{R}$ (such that t is a valid chopping point for B along coordinate i) let $B_{+,t} := \{\mathbf{x} \in \mathbb{R} \,|\, x_i > t\}$ and $B_{-,t} := \{\mathbf{x} \in \mathbb{R} \,|\, x_i \leqslant t\}$ by the two boxes obtained from B by chopping the box B along coordinate i at the value t.
  - Now let $\hat{y}_S(B_{\pm,t})$ as defined above be the class assignment for box $B_{\pm,t}$ that minimizes the empirical misclassification risk (with respect to training sample S) in that box. And let $P^S_{\hat{y}_S(B_{\pm,t})}(B_{\pm,t})$ be that optimal empirical misclassification risk incurred by these class assignments.
  - Now, let $g_i(t) = P^S_{\hat{y}_S(B_{+,t})}(B_{+,t}) + P^S_{\hat{y}_S(B_{-,t})}(B_{-,t})$. Convince yourself that this is the best empirical risk incurred by chopping the box B at value t along coordinate i.

- Let $t_i := argmin_{t \in \mathbb{R}}\ g_i(t)$ be the **best** chopping point along coordinate i. Calculate t_i for each coordinate $i = 1, \ldots d$.
- Let $i_* := \text{argmin}_{1 \leqslant i \leqslant d}\, g_i(t_i)$ be the **best** coordinate along which to chop B.
- Now chop B along coordinate i_* at the value t_{i_*}.

All this was just a formal way of communicating the very simple idea used by the CART heuristic to grow a full tree: (i) chop each box again and again until you have reached k levels in the tree, or reached a box with too few training samples; and (ii) chop each box along the **best** coordinate, at the **best** value to chop, and whenever you chop give each half-box the **best** class assignment, in the sense of misclassification error on the training sample.

What's the computational complexity of this stage at the heuristic?

- First, convince yourself that each split only takes $O(md)$ steps. To see why, observe that for a given coordinate, we don't actually have to scan over all possible chopping values $t \in \mathbb{R}$, but only in one value per training sample (why?). And then we scan over d coordinates to find the best one.
- So, growing a classification tree with at most k levels, using the CART heuristic, will take $O(md \cdot 2^k)$ steps. But we have an upper bound on k, that comes from the training sample size m.
- So what's the time complexity of growing a classification tree with CART using a training sample of size m?

11.7 Why to prune a classification tree

Pruning a tree means cutting off unnecessary branches. The tree obtain when we're done with the "growing" stage of CART may be too large. A tree too large means some of the boxes are too small, so we are not in an optimal point on the bias-variance tradeoff (see discussion above about k the maximal tree depth). It could help reduce the generalization error to merge some of the boxes together, so that the majority votes to determine the box label assignment would be based on larger sets of training samples. Merging two boxes is equivalent, from the decision tree perspective, to merging to leaves together and removing the node between them. Hence, "pruning". We will complete the CART heuristic in a future lecture when we learn how CART prunes a classification tree after growing it.

11.8 How to make predictions on a new sample

Once we have finished running the learner and have chosen a hypothesis $h_S \in \mathcal{H}_{CT}$ (a classification tree), making a prediction on a new sample $\mathbf{x} \in \mathbb{R}^d$ is very simple: we just run along the tree from top to bottom and answer each question using $\mathbf{x} \in \mathbb{R}^d$ until we get to a leaf. The leaf identifies the box to which the point $\mathbf{x} \in \mathbb{R}^d$ belongs, and therefore determines the predicted label.

11.9 Interpretability

One of the great advantages of a classification tree is that it's so **interpretable**. This is possible the most interpretable classifier.

- To understand which features where important in the classification process, we just look at the nodes (the splits) and see which features the classification tree algorithm chose to split on. A feature that never appeared in any split has not been useful for classification of the training sample. A feature that appears once or more (remember that the algorithm can choose to split on some feature again and again in different areas of $\mathbb{R}^d$) has been useful.
- To understand why a new sample was classified the way it was classified, we just follow the tree from top to bottom, and see how each answer to each question went.

11.10 Summary- Classification Trees

- Hypothesis class $\mathcal{H}_{CT}^k$: Piecewise-constant functions induced by Tree Partitions (axis-aligned rectangles) of depth at most k, where k is given parameter
- Learning principle for training model (choosing $f \in \mathcal{H}$): ERM
- Computational implementation of learning principle: Top-down greedy heuristics such as CART (implementing ERM is NP-hard)
- How to make predictions on new samples: Go top-down along the tree until a leaf is reached.
- Interpretable: Yes - just read the tree
- Estimates class probabilities: No
- Family of models: Yes, indexed by k the maximal tree depth. (Later when we talk about pruning we will see a more delicate way to control the hypothesis class size.)
- Time complexity for training, and for predicting on a new sample:
- How to store trained model: For each node in the tree, store the split information (which coordinate i was used to split, and which value t was used to split). And, store the class assignment given §for each leaf in the tree.
- When to use: As a simple baseline, or to get a highly interpretable rule that is easy to explain and easy to plot. Otherwise, classification trees are used to construct "random forests" which are very effective very popular classifiers. (more or forests, later.)

Lecture 4: PAC Theory of Statistical Learning, Part I

12 Introduction

(Recommended reading for this week and next week: Understanding Machine Learning, chapters 2-6).

Example: Immunotherapy. We are working on a new medical treatment and we would like to predict for each patient whether the treatment will work for her or not. Available to us is a data set which is shown in Figure 1 below. On the basis of this data we need a no/yes (0 or 1) prediction in order to decide whether or not to apply the treatment. Let us formulate the

sex	age	Time	Number_o	Area	induration	Result_of_Treatment
1	22	2.25	14	51	50	1
1	15	3	2	900	70	1
1	16	10.5	2	100	25	1
1	27	4.5	9	80	30	1
1	20	8	6	45	8	1
1	15	5	3	84	7	1
1	35	9.75	2	8	6	1
2	28	7.5	4	9	2	1
2	19	6	2	225	8	1
2	32	12	6	35	5	0
2	33	6.25	2	30	3	1
2	17	5.75	12	25	7	1
2	15	1.75	1	49	7	0
2	15	5.5	12	48	7	1
2	16	10	7	143	6	1
2	33	9.25	2	150	8	1
2	26	7.75	6	6	5	1
2	23	7.5	10	43	3	1
2	15	6.5	19	56	7	1
2	26	6.75	2	6	6	1
1	22	1.25	3	47	3	1

Taken from UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]

Figure 23: Immunotherapy Data Set

ingredients of a general problem of this type.

- The individual object, $\mathbf{x}$, that we wish to label. In our case, each such object is a vector $\mathbf{x}$ of d real numbers describing d features of a patient, as shown in the figure.
- The set of all $\mathbf{x}$'s is called the **Domain set,** $\mathcal{X}$. We can take, for example, $\mathcal{X} = \mathbb{R}^d$ (We could also take a smaller $\mathcal{X}$ since age must be positive and sex can have only two values).
- **Label set,** $\mathcal{Y}$: set of possible labels. In our case we can take, for example, $\mathcal{Y} = \{0, 1\}$, where 0 corresponds to a recommendation for not giving the treatment and 1 for giving it.
- **A prediction rule,** $h : \mathcal{X} \to \mathcal{Y}$: used to label future examples. This function is called a **predictor**, a **hypothesis**, or a **classifier**.

So the learner's input (e.g., the data in the figure) is called **Training data,** and consisting of m already-labelled objects $S = ((\mathbf{x}_1, y_1), ..., (\mathbf{x}_m, y_m)) \in (\mathcal{X} \times \mathcal{Y})^m$, and the learner's output is the prediction rule, $h : \mathcal{X} \to \mathcal{Y}$. So a learner is a map $\mathcal{A} : S \mapsto h$. The goal of the learner is to produce an h that will be correct (as much as possible) on future examples. So we need to have a way to quantify this, namely, to measure the quality of prediction of a candidate rule h with respect to future examples.

13 A Theoretical framework for learning

The most basic questions in machine learning are: Is anything learnable? what can be learned and what cannot? When learning is possible, how many training samples do we need to learn? When learning is possible, how do we learn? In this lecture we develop the **PAC theory of learning**, a famous theory that gives (within its assumptions and framework) a complete answer to this question - for **batch supervised learning**.

13.1 A Data-generation Model

To formal mathematical answers to such questions, we must make formal mathematical assumptions on how our training and test samples are generated. From now on, we assume the following:

- There is a distribution $\mathcal{D}$ over the examples $\mathcal{X}$. That is, each $\mathbf{x}_i$ is sampled according to $\mathcal{D}$ and therefore each sequence S, of m $\mathbf{x}$'s, has a different probability to appear.
- The $\mathbf{x}$'s are **Independently Identically Distributed (i.i.d.)**. In particular, the training set S consists of $(x_i, f(x_i))_{i=1..m}$, where the $\mathbf{x}_i$'s were sampled independently according to $\mathcal{D}$, and our test set, will be made of future examples: $x \in \mathcal{X}$, again to be sampled according to $\mathcal{D}$ independently of each other and of those in the training set.
- There is a function f which is the correct classifier, i.e., for every $i \in [m]$, $y_i = f(x_i)$ For now, we assume that f is deterministic (no noise): $y = f(x)$ and we will call it the "PAC model". In following lectures we'll add noise and call it the "Agnostic PAC model".

Compare the above to first learning problem we saw - the Linear Model: There, we had samples[17] $\mathbf{x}_1, .., \mathbf{x}_m \in \mathbb{R}^d$ fixed (given) and all of them received equal importance, for example, in their contribution to the global error (the loss function). In the framework we are developing now, the samples $\{x_i\}$ are i.i.d samples according to $\mathcal{D}$, i.e., they have different probabilities to appear and therefore may have a different weights in the loss function. In the previous lecture we started with assumption $y_i = f(\mathbf{x}_i)$ where f is deterministic and linear. We then assumed *noise* $y_i = f(x_i) + z_i$. Next lecture we will add noise also to our current model.

13.2 Classifiers

We will focus on focus on **classifiers** until further notice. However, many principles you'll see today are true for regression problems as well.

For a classifier, we define the **Generalization Error** as:

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{P}}_{x\sim\mathcal{D}}[h(x) \neq f(x)] \stackrel{\text{def}}{=} \mathcal{D}\left(\{x \in \mathcal{X} : h(x) \neq f(x)\}\right) .$$

where $\mathcal{D}$ and f are unknowns. The generalization error is also called the **risk**, or the **true error**. Recall that $\mathcal{D}$ is a distribution over $\mathcal{X}$, that is, for a given $A \subset \mathcal{X}$, the value of $\mathcal{D}(A)$ is the probability to see some $x \in A$.

Note: Recall that you should be critical about an error measure that counts total number of misclassification error of a classifier. Recall that there are two kinds of errors a classifier can make. they are called Type-I and Type-II errors and one is usually much worse than the other[18].

[17] the sample space there was $\mathcal{X} = \mathbb{R}^d$

[18] For now, we only note that, assuming $\mathcal{Y} = \{0, 1\}$, in the above definition of the true error, $h(x) \neq f(x)$ may

13.3 The framework so far

Make sure you understand the theoretical framework we have developed so far. Our task is to design a learning algorithm (a "learner"), which we denote by $\mathcal{A}$. For a given sample size m, $\mathcal{A}$ is a map from the training sample

$$S = ((x_1, y_1), .., (x_m, y_m)) \in (\mathcal{X} \times \mathcal{Y})^m$$

where each $\mathbf{x}_i$ to the set of all possible rules $\mathcal{X} \to \mathcal{Y}$.

We're thinking of classification problems now, so in this lecture $\mathcal{Y} = \{\pm 1\}$, but everything here can be generalized. The **data generation model** we assume is probabilistic, in fact i.i.d: We assume that there is an unknown distribution $\mathcal{D}$ over $\mathcal{X}$ (so that $(\mathcal{X}, \mathcal{D})$ is a probability space - we won't pay attention to sigma-algebras, measurable sets and all that jazz). We assume that each sample - both in training set and it test set - is sampled according to $\mathcal{D}$ independently from any other sample before of after it. For the labels y, we assume there is an unknown function $f : \mathcal{X} \to \mathcal{Y}$ such that for each sample $x \in \mathcal{X}$, the corresponding label is $y = f(x)$. In particular, for our training data, we have $y_i = f(x_i)$ for $i = 1, \ldots, m$. Finally, for a candidate prediction rule $h : \mathcal{X} \to \mathcal{Y}$ that our learner may produce, will measure the generalization performance of h - how well it will perform on future unseen samples - by simply using the expected misclassification rate

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{P}}_{x \sim \mathcal{D}}\left[h(x) \neq f(x)\right].$$

13.4 Our goal in this lecture

Our goal in this lecture will be to understand, in detail, the following two definitions.

Definition: PAC learnability and sample complexity

1. A hypothesis class $\mathcal{H}$ is **PAC Learnable** if there exists a function $\tilde{m}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following property: For every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$ that satisfies $L_{\mathcal{D},f}(h^*) = 0$ for some $h^* \in \mathcal{H}$, when running the learning algorithm $\mathcal{A}$ on $m \geqslant \tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f, the algorithm returns an hypothesis $h_S = \mathcal{A}(S)$ such that, with probability of at least $1 - \delta$ (over the choice of the training samples), we have $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$.

2. For a PAC learnable hypothesis class, we define the **Sample Complexity** of $\mathcal{H}$ for specified ϵ, δ as the minimal number of samples $\tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ required for the definition to hold with respect to ϵ, δ. The Sample Complexity function of $\mathcal{H}$ is denoted $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$.

Definition: VC-Dimension
Let $\mathcal{H} \subset \{\pm 1\}^{\mathcal{H}}$ be an hypothesis class. For a subset $C \subset \mathcal{X}$ let $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to C, namely, $\mathcal{H}_C = \{h_C : h \in \mathcal{H}\}$, where for $h : \mathcal{X} \to \mathcal{Y}$, $h_C : C \to \mathcal{Y}$ is the function such that $h_C(x) = h(x)$ for every $x \in C$. Define the **VC-dimension** of $\mathcal{H}$ by

$$VCdim(\mathcal{H}) := \max\{|C| \;\Big|\; C \subset \mathcal{X} \text{ and } |\mathcal{H}_C| = 2^{|C|}\}.$$

refer either to $h(x) = 1, f(x) = 0$ or to $h(x) = 0, f(x) = 1$. According context of the actual learning problem at hand, one of these two cases will correspond to a **False Positive** event and the other to a **False negative** event

Note that $VCdim(\mathcal{H}) \leqslant \infty$.

The Fundamental Theorem of Statistical Learning.
These two definitions are interesting since, if we adopt PAC learnability as our notion of learnability, then we have a **complete characterization of when it is possible (and impossible) to learn, in terms of the VC-dimension, as well as the exact minimal training sample size we need in order to learn, and a "universal" learner that successfully learns when enough training data is available.** In short, we have a full theory of batch learning - a full theory of when it is possible to generalize from a training sample to new samples, and how to do it.

This result is sometimes known as "the fundamental theorem of statistical learning" and states the following (roughly):

- An hypothesis class $\mathcal{H}$ is PAC-learnable **if and only** if $VCdim(\mathcal{H}) < infty$
- The sample complexity of an hypothesis class with finite VC-dimension is roughly

$$m_H(\epsilon, \delta) \sim \frac{VCdim(\mathcal{H}) + \log(1/\delta)}{\epsilon}$$

- The ERM rule achieves this minimum, namely, when learning is possible, ERM learns with a minimial number of examples.

Since the definitions of PAC-learnability and VC-dimension are quite complicated, we will take the rest of the lecture to slowly unpack them. So now, let's put ourselves in the framework as it appears in Section 13.3, and advance from there in baby steps.

13.5 Learning as a Game - first attempt

This framework in Section 13.3 can be thought of as a **game** between us and Nature, with a random payoff. The game proceeds as follows. The number of training samples m is given in advanced as a game parameter.

We move first. We choose a learner $\mathcal{A}$ that trains on m samples. This is our strategy. Nature moves second. Nature chooses a distribution $\mathcal{D}$ and a labeling function f. This is Nature's strategy. Importantly, Nature knows the strategy we chose when she chooses her strategy. To calculate the game's payoff, an i.i.d sample S of length m is drawn according to the distribution $\mathcal{D}$ that Nature chose, is labeled according to the function f that Nature chose, and is fed into the learner $\mathcal{A}$ that we chose to obtain the prediction rule $h_S := \mathcal{A}(S)$ (the notation h_S helps us remember that the prediction rule we learn, h_S, strongly depends on the random sample S that we drew). The payoff is $L_{\mathcal{D},f}(h)$. We can think that we want it to be as small as possible, and maybe Nature is an adversary that wants it to be as large as possible. Note that the payoff is random as it depends on the sample S that was drawn. If we're unlucky, and the sample S is "bad", namely, does not represent $\mathcal{D}$ very well, then the rule h_S our learner produces might not generalize well and the random loss (for that draw of S) will be high.

Now you might ask: what's the best strategy for us (how to best design $\mathcal{A}$)? Remember that Nature will know what we chose, and can try to be "cruel", namely, to choose $\mathcal{D}$ and f that

our learner did not prepare well for. Also, there's always a chance that we will draw a "lousy" sample S, namely a misleading sample that will confuse our learner and will cause it to output a rule h_S that will not generalize well. Is there a way to "defend" ourselves against "cruel" strategies $\mathcal{D}, f$ that Nature might play, and against "unlucky" draws of a training sample S? Is there anything at all that can be said in this generality about the problem of learning (= the problem of generalizing from training samples to new samples)?

Let's define this as **The Learning Game (first version)**: Fix the sample size m. Let's view our framework as a game between us and Nature, with random payoff.

- We choose a strategy (namely, a learner) $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{Y}^{\mathcal{X}}$
- Nature knows our strategy, and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X}$, and a label function $f : \mathcal{X} \to \mathcal{Y}$.
- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$
- The payoff is $L_{\mathcal{D},f}(h_S)$, namely, the expected fraction of misclassification errors h_S will make on data drawn i.i.d according to $\mathcal{D}$ and labeled according to f. **The payoff is random since S is random and therefore h_S is random.**
- We are going to assume Nature is "cruel" and does her best to win. So we'll look for learners $\mathcal{A}$ that have a **guaranteed maximal loss $L_{\mathcal{D},f}(h)$ for any** strategy $\mathcal{D}, f$ that Nature might play.

14 Probably correct & Approximately correct learners

We now can ask two questions. Recall that the loss is random - it depends on the draw of training sample S.

First question:

- Is there a strategy that will guarantee some upper bound $0 \leqslant \epsilon < 1$ on the loss with probability 1 (namely, almost surely for any sample S that might be drawn according to $\mathcal{D}$?)

Answer to first question: **No.**

- Choose $0 \leqslant \epsilon < 1$. The learner cannot hope to produce, regardless of how Nature plays, and with probability 1, a rule h with $L_{\mathcal{D},f}(h) \leqslant \epsilon$.
- Reason: There's always a (small) probability to get a completely "pathological" training sample S that does not represent $\mathcal{D}$ at all. The resulting rule h_S can be wrong on most of $\mathcal{X}$. If S is really bad, h_S can have a loss as high as 1, higher than any ϵ.
- **Example:**

- take $\mathcal{X} = \{x_1, x_2\}$. Fix γ. Our learner $\mathcal{A}$ must specify what to predict on a point that was not seen in the training set. WLOG we choose to predict + on a point we have not seen in the training set. Now, Nature plays $\mathcal{D}$ with $\mathcal{D}(\{x_1\}) = 1 - \gamma$, $\mathcal{D}(\{x_2\}) = \gamma$, and chooses a labeling function $f(x_1) = f(x_2) = -$. With probability γ^m, the training sample S only contains x_2. How does $h_S = \mathcal{A}(S)$ classify x_1 in this event? since there's no training data there, we will predict $h_S(x_1) = +$. As as result, on a random test point drawn according to $\mathcal{D}$, unfortunately h_S will be wrong with probability $1-\gamma$. So $L_{\mathcal{D},f}(h) = (1-\gamma)$. Now make γ small enough to have $1-\gamma > \epsilon$.
- In conclusion, even on a sample space $\mathcal{X}$ with only two points, for any fixed $\epsilon > 0$, for every strategy $\mathcal{A}$ we might play, Nature has a strategy $\mathcal{D}, f$, such that with some probability δ that depends only on ϵ, over the choice of training samples of length m, we have $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$.
- As ϵ was arbitrary, this means that for any arbitrarily "bad" loss ϵ, Nature can cause the game to end, with strictly positive probability, with a loss at least ϵ.
- What went wrong? In this example, with probability γ^m (which is really very tiny) we get a "lousy" training sample S, one that does not represent $\mathcal{D}$ and good enough, and does not allow us to generalize.

- **Claim:** For any desired upper bound $\epsilon < 1$ on the loss, no learner $\mathcal{A}$ can guarantee $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$ with probability 1 against **any** strategy of Nature.
- **Relaxation:** We allow the learner $\mathcal{A}$ to "fail completely" (produce h_S with potentially huge loss) with probability δ, where $\delta \in (0, 1)$ is specified in the conditions of the game.
- **Bottom line:** We must allow a small probability for "pathological" sample S, from which it is impossible to generalize.
- **Definition:** When a learner $\mathcal{A}$ has an arbitrary large loss with probability at most δ (for some $\delta > 0$), we say that $\mathcal{A}$ is **Probably** correct, with **confidence** δ.

So, we have no choice but to make a huge sacrifice - we must allow our learner to **fail completely** with probability at most δ, where δ is a fixed parameter. We can only hope for an upper bound on the loss (an upper bound that will hold regardless of how Nature plays) - with probability at least $1 - \delta$.

This leads us to the second question that we now ask:

Second question:

- On the event of a non-pathological sample (the event with probability at least $1 - \delta$), can we please have perfect accuracy? Namely, is there a strategy $\mathcal{A}$ for us with **zero loss** (w.p. at least $1 - \delta$) regardless of how Nature plays?

Answer to second question: **No.**

- Choose $\delta > 0$ (probability of complete failure - getting an arbitrarily large loss). The learner cannot hope to produce, regardless of how Nature plays, and with probability at least $1 - \delta$, a rule h with $L_{\mathcal{D},f}(h_S) = 0$.
- Reason: $L_{\mathcal{D},f}(h) = 0$ means that $\mathcal{D}\{h_S(x) = f(x)\} = 1$ (h_S is never wrong, almost surely with respect to $\mathcal{D}$). If Nature plays $\mathcal{D}$ that gives a tiny mass to some $x \in \mathcal{X}$, with high probability, the S will not include x, and there, h_S has no idea what to do. So it cannot generalize perfectly to x.

- **Example:**
  - Fix confidence $\delta > 0$. Our learner will be allowed to fail completely (have an arbitrary large loss) with probability δ. As before, fix γ. Our learner $\mathcal{A}$ must specify what to predict on a point that was not seen in the training set. WLOG we choose to predict $+$ on a point we have not seen in the training set. As before, Nature plays $\mathcal{D}$ with $\mathcal{D}(\{x_1\}) = 1-\gamma$, $\mathcal{D}(\{x_2\}) = \gamma$, and chooses a labeling function $f(x_1) = f(x_2) = -$. Now, with probability $(1-\gamma)^m$, S does not include x_2. How does $h_S = \mathcal{A}(S)$ classify x_2 in this event? since there's no training data there, we will predict $h_S(x_2) = +$. As as result, on a random test point drawn according to $\mathcal{D}$, unfortunately h_S will be wrong with probability γ. So $L_{\mathcal{D},f}(h) = \gamma > 0$. Now make γ small enough to have $(1-\gamma)^m > \delta$.
  - In conclusion, for any fixed confidence $\delta > 0$, for every strategy $\mathcal{A}$ we might play, Nature has a strategy $\mathcal{D}, f$, such that with probability strictly more that δ over the choice of training samples of length m, we have $L_{\mathcal{D},f}(h_S)$ strictly positive and bounded away from 0.
  - What went wrong? In this example, with probability $(1-\gamma)^m$ (which is large - we got a "typical" training sample), we "miss" the rare point x_2 in our training sample, and cannot have perfect generalization.
- **Claim:** For any confidence δ, no learner can hope to produce a perfect rule (a rule h_S with $L_{\mathcal{D},f}(h_S) = 0$) with probability at least $1-\delta$ regardless of Nature's strategy.
- **Relaxation:** We must be satisfied with an **approximately correct** learning rule: A rule h_S with $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$, where ϵ is specified in the conditions of the game.
- **Bottom line:** Even a "typical" ("non-pathological") training sample S may miss small areas in $\mathcal{X}$. On these areas the resulting h_S will be wrong in the worst case. We have to allow h_S to be wrong sometimes (to make some generalization errors) even when the training sample is "typical".
- **Definition:** When a learner $\mathcal{A}$ has an upper bound $\epsilon > 0$ on the loss (for some ϵ), we say that $\mathcal{A}$ is **Approximately** correct
  with **accuracy** ϵ.

Now we can define a **Probably Approximately Correct** learner.

Definition: When a learner $\mathcal{A}$ has an upper bound $\epsilon > 0$ on the loss with probabilty at least $1-\delta$, for some $\delta > 0$ and $\epsilon > 0$, but and can have an arbitrary large loss with probability at most δ, we say that $\mathcal{A}$ is **Probably Approximately** correct with accuracy ϵ and confidence δ.

It's important to understand the difference between **accuracy** ϵ and **confidence** δ: *First*, we draw the training sample S at random. The learning algorithm runs on this random input and its prediction is therefore random. If S is by chance "weird" (not representing $\mathcal{D}$ well), the rule h produced will be "wrong", namely, it won't generalize well. The number δ is the probability of failure due to a "weird" sample S. *Second*, we test the rule h on new data. New data is also random. $L_{\mathcal{D},f}(h)$ is the expected number of errors h will make. The number ϵ refers to the accuracy of h.

14.1 The game - for Probably Approximately correct learners

Since we can only hope to build a Probably Approximately correct learner, we will update the game definition a little. The sample size m will no longer be fixed. Instead, the accuracy ϵ and confidence δ, which our learner is required to achieve, are specified as game parameters. We get to decide on m in our strategy. Note that there is subtle nuance in notation now: when we write $\mathcal{A}$ for the learner, we actually mean a **sequence** of learners - one for each m. It would be better to write $\mathcal{A}_m : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{Y}^{\mathcal{X}}$, but we won't bother writing $\mathcal{A}_m$ and will continue to use $\mathcal{A}$.

So here is our updated game - (**The Learning Game (second version):** Fix desired accuracy $\epsilon > 0$ and confidence $\delta > 0$. We play a game against Nature, with random payoff.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{Y}^{\mathcal{X}}$. Both m and $\mathcal{A}$ can depend on (ϵ, δ).
- Nature knows our strategy, and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X}$, and a label function $f : \mathcal{X} \to \mathcal{Y}$. That is, Nature's strategy can depend on (ϵ, δ) specified, and also on the $m, \mathcal{A}$ we chose.
- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$
- The payoff is $L_{\mathcal{D},f}(h_S)$. It is random since S is random and therefore h_S is random.
- We are going to assume Nature is "cruel" and does her best to win. So we'll look for learners $\mathcal{A}$ that have a **guaranteed maximal loss** $L_{\mathcal{D},f}(h)$ **for any** strategy $\mathcal{D}, f$ that Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D},f}(h_S) \leqslant \epsilon\}$. If this probability is found to be larger than $1 - \delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ϵ and confidence δ - against Nature's best strategy - **we say that we've been successful (with regards to the parameters ϵ, δ)**.

If you don't like the game perspective, here our current definition of our learning challenge: The learner doesn't know $\mathcal{D}$ and f. The learner receives an accuracy parameter ϵ and a confidence parameter δ. It then ask for training data, S, containing $m(\epsilon, \delta)$ examples (that is, the number of examples can depend on the value of ϵ and δ, but it can not depend on the unknown $\mathcal{D}$ or f). Finally, the learner should output a hypothesis h_S, that depends only on ϵ, δ and the training sample S drawn, such that with probability of at least $1 - \delta$ it holds that $L_{\mathcal{D},f}(h) \leqslant \epsilon$. That is, the learner should be Probably Approximately correct, with the specified accuracy ϵ and confidence δ.

Since we can now choose the sample size m, and data costs money, we want m to be as small as possible - as long as we still design a probably approximately correct learner. We sense that there must be some trade-off between ϵ, δ and $m(\epsilon, \delta)$ The figure below shows schematically the connection between m,δ, and ϵ.

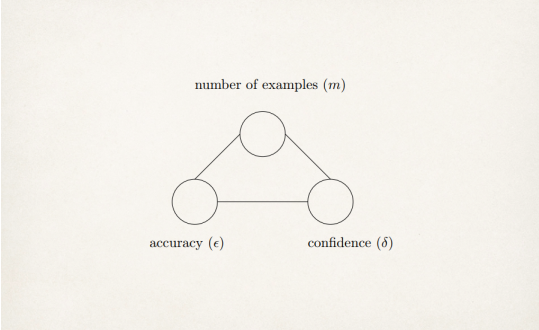


Figure 24: Connection between m,δ, and ϵ.

15 No Free Lunch and Hypothesis Classes

15.1 No Free Lunch!

Turns out that, unfortunately, we cannot in general be successful against Nature even in this second version of our game - for any parameters ϵ, δ... Why? If we know nothing about D and f, and if there are too many possibilities for f, then no matter how large our sample size, m, we can not be confident-enough that we can find an accurate-enough h_S. Let's understand why.

The two examples above (which demonstrated we can't hope to get a learner that enjoys confidence $\delta = 0$ or accuracy $\epsilon = 0$) only needed a sample space $\mathcal{X}$ with two points. Now let's consider a sample space with a countably infinite number of points.

Example.

- Suppose that $|\mathcal{X}| = \infty$. Recall that we move first and choose m, and that Nature knows the m we chose. So, let our choice of m be fixed. We must choose what to predict on a point we have not seen in the training sample. Denote our choice by $g(x)$. That is, our learner $\mathcal{A}$ specifies that if a point $x \in \mathcal{X}$ was **not** observed in our training data, then the rule that $\mathcal{A}$ outputs will predict $h_S(x) = g(x)$.

- Now, Nature picks some finite set $C \subset \mathcal{X}$ with $|C| > 2m$, and chooses $\mathcal{D}$ to be uniform over C. For the labeling function f, Nature plays a sinister move, and chooses $f(x) = -g(x)$ for all $x \in \mathcal{X}$. (The opposite of what h_s will predict on unseen points!)

- Now, let S be a training sample. Let $supp(S) \subset C$ denote all points $x \in \mathcal{X}$ observed in the particular training sample S. Since $|supp(S)| \leqslant m$ and since $\mathcal{D}$ is uniform over C, we have $\mathcal{D}(\{x \in \mathcal{X} \backslash supp(S)\}) \geqslant 1/2$. In other words, the probability that a new test point drawn according to $\mathcal{D}$ will be unseen in S is at least $1/2$.

- Now, as the game demands, we feed the sample S into $\mathcal{A}$ and obtain $h_S = \mathcal{A}(S)$. What is the loss? Regardless of what h_S predicts on points in the training sample S, since a test point has probability $\geqslant 1/2$ to be in the unseen part $\mathcal{X} \backslash supp(S)$, with probability at least $1/2$, the rule h_S will predict $h_S(x) = g(x)$ - and will be wrong, since Nature played the function $f(x) = -g(x)$. We conclude that, on the particular training sample S, we must have $L_{\mathcal{D},f}(h_S) \geqslant 1/2$.

- But this happens for **every** training sample S. So, Nature played a strategy for which, with probability 1 over the choice of training samples (according to $\mathcal{D}$), the game results in loss $L_{\mathcal{D},f}(h_S) \geqslant 1/2$.
- So, if we were looking to find a learner $\mathcal{A}$ that will be Probably Approximately correct (to some specified ϵ and δ regardless of Nature's strategy $\mathcal{D}, f$, we find that we cannot. And asking for a larger training sample won't help - if we increase m, Nature will just choose a larger set C and a distribution $\mathcal{D}$ uniform over that larger C.
- What went wrong? Nature could choose **any labeling function at all** f that she wanted, and we tried to learn (to generalize = to accurately predict) f from a sample that was too small for the number of possible functions we needed to choose from. We find we just cannot design a Probably Approximately correct learner if the set of possible labeling functions is "too large".

This is known as a **"No Free Lunch" Theorem**: without assuming anything in advance on f, without an prior information on the labeling function we are trying to learn, we find that learning is impossible. Equivalently, if the set of possibilities for the labeling function f is too large, then Nature can be cruel and play a function that we can't learn - the larger the sample size we choose, the more complicated the function f that Nature playes. So if the set of possibilities for the labeling function f is too large, learning is impossible (in the sense of the game that we defined).

Important Note! The argument given above is in fact wrong (can you find where?). While this was not a proof of a No Free Lunch Theorem, it gives all the crucial bits of intuition, and we will need them later. So we will call it a "proof" in quotes.

Here is an actual, formal No Free Lunch theorem. (There are many theorems that can be called by that name - basically any theorem that shows that without some prior knowledge on the labeling function, when there are "too many" possibilities for f, learning it is impossible.) You can read the proof in the Understanding Machine Learning book - Theorem 5.1, and exercise 3 in section 5.5. (it is not so easy to read and uses Agnostic PAC, a notion we will only get to later.)

Theorem 4 (No Free Lunch) *$\mathcal{X}$ be an infinite sample domain, $|\mathcal{X}| = \infty$. Fix $\epsilon < 1/2$. There always exists some $\delta > 0$ so that, for every learner $\mathcal{A}$ and training set size m, there exists a distribution $\mathcal{D}(x)$ over $\mathcal{X}$ and a function $f : \mathcal{X} \to \mathcal{Y}$, such that with probability of at least δ over the generation of a training sample S of size m drawn i.i.d from $\mathcal{D}$, we have $L_{\mathcal{D},f}(h_S) \geqslant \epsilon$ where $h_S = \mathcal{A}(S)$.*

15.2 We need hypothesis classes

So, in order to be able to learn, the learner **must** receive enough prior knowledge about the function f. This implies that we should assume that the target f comes from some **hypothesis class**, $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$.

Realizability Assumption. Suppose that an hypothesis class $\mathcal{H}$ is specified for our game. The **realizable case** is when Nature must play a function $f \in \mathcal{H}$. Actually, we don't care if

Nature plays a function f that is not in $\mathcal{H}$ as long as f is $\mathcal{D}$-almost surely identical to a function $h^* \in \mathcal{H}$. (This is because we will never see samples in $\mathcal{X}$ where $f(x) \neq h(x)$ - not in the training and not in the test samples.) So the formal mathematical **Realizability Assumption** is this: Nature plays a function f such that there exists $h^* \in \mathcal{H}$ with $L_{\mathcal{D},f}(h^*) = 0$.

The learner is given $\mathcal{H}$ before learning starts, and will only output $h_S \in \mathcal{H}$. In other words, for a training sample of size m the learner (learning algorithm) is a map $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$ such that $\mathcal{A} : S \mapsto h \in \mathcal{H}$. (As before, we will continue to abuse notation and write $\mathcal{A}$ instead of $\mathcal{A}_m$, and when we say "the learning algorithm $\mathcal{A}$" we will sometimes mean "a sequence of learners $\{\mathcal{A}_m\}_{m=1}^{\infty}$, one for each possible sample size").

We've just seen that if $|\mathcal{X}| = \infty$ then $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$ is "too large to learn". And, in a previous section we've seen that the learner gets to specify the training sample size m. The questions that arise now are:

- What are the "small enough" hypothesis classes $\mathcal{H}$ for which we **can** find a Probably Approximately correct learner? And what are the "too large" hypothesis classes $\mathcal{H}$ for which we **cannot**? Can we characterize exactly the hypothesis classes for which it is possible to learn?

- Assume we have a "small enough" $\mathcal{H}$. This means that for every ϵ, δ we have at least one strategy $m, \mathcal{A}$ such that $\mathcal{A}$ is Probably Approximately correct (with accuracy ϵ and confidence δ) no matter how Nature plays. This means that for every ϵ, δ there is a **minimal number of training samples**: maybe some learners are wasteful and need more training data than others, but for every ϵ, δ there is the absolutely minimal training sample size taht allows us to choose a Probably Approximately correct learner. Can we characterize this minimal function $m_{\mathcal{H}}(\epsilon, \delta)$? Is there a connection between the "size" of the hypothesis class $\mathcal{H}$ and $m_{\mathcal{H}}(\epsilon, \delta)$?

- Assume we have a "small enough" $\mathcal{H}$. Can we characterize explicitly a learner $\mathcal{A}$ that always succeeds in learning functions from $\mathcal{H}$? And how many training samples m does $\mathcal{A}$ need to always succeed in learning a function from $\mathcal{H}$ (always be Probably Approximately correct, no matter how Nature plays)? Can we find the **most training-data efficient** learner, namely a learner that can succeed with the minimal number of samples $m_{\mathcal{H}}(\epsilon, \delta)$ mentioned above?

15.3 Updating the game one last time

So here is our final version of the game - (**The Learning Game (third version):** Fix desired accuracy $\epsilon > 0$ and confidence $\delta > 0$. Fix an hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$. We play a game against Nature, with random payoff.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{H}$. Both m and $\mathcal{A}$ can depend on (ϵ, δ).

- Nature knows our strategy, and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X}$, and a label function **from the specified hypothesis class** $f \in \mathcal{H}$. That is, Nature's strategy can depend on $\epsilon, \delta, \mathcal{H}$ specified, and also on the $m, \mathcal{A}$ we chose.

- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f

- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$. Note that $h_S \in \mathcal{H}$.
- The payoff is $L_{\mathcal{D},f}(h_S)$. It is random since S is random and therefore h_S is random.
- We are going to assume Nature is "cruel" and does her best to win. So we'll look for learners $\mathcal{A}$ that have a **guaranteed maximal loss** $L_{\mathcal{D},f}(h)$ **for any** strategy $\mathcal{D}, f$ that Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D},f}(h_S) \leqslant \epsilon\}$. If this probability is found to be larger than $1-\delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ϵ and confidence δ - against Nature's best strategy - **we say that we've been successful (with regards to the parameters ϵ, δ) and hypothesis class $\mathcal{H}$.**

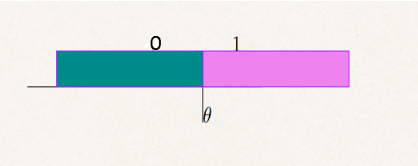
15.4 Example: Threshold functions

We saw above that if $|\mathcal{X}| = \infty$ and $\mathcal{H}$ is the class of all functions from $\mathcal{X}$ to Y, namely $\mathcal{H} = \mathcal{Y}^{\mathcal{X}}$, we can't be successful in the the third version of the learning game. Well, maybe it's just impossible to learn when $|\mathcal{X}| = \infty$? Fortunately, when the hypothesis class is "small enough", it is possible to be successful in the third version of the learning game. (Otherwise we wouldn't be having a course in machine learning.) In the following example we will have an infinite (in fact uncountably infinite) sample space, and a very simple hypothesis class. We will see that we can be successful in the third version of the game, for any ϵ, δ specified.

Consider the domain $\mathcal{X} = \mathbb{R}$, i.e., there is only one feature. We work with classification, and use the label set is $\mathcal{Y} = \{0, 1\}$ (instead of $\{+, -\}$). Define the hypothesis class of **Threshold functions:** over $\mathbb{R}$:

$$\mathcal{H}_{th} = \{x \mapsto h_\theta(x) : \theta \in \mathbb{R} \bigcup \pm\infty\}$$

where $h_\theta(x) = 0$ for $x \leqslant \theta$ and $h_\theta(x) = 1$ for $x > \theta$. (We define $h_\infty(x) = 0$ for all x, and $h_{-\infty}(x) = 1$ for all $x \in \mathbb{R}$).



What's the meaning of this definition? We are classifying points on the real line. Up until a certain unknown point, the points are in class 0. Beyond that point, they are in class 1. Nature chooses the true cutoff threshold θ, and a distribution $\mathcal{D}$ over the real line. We would like to receive a training sample S of labeled points, and successfully predict the label of future examples. In this case, our job is to determine, as accurately as possible, the unknown cutoff θ. (It is recommended to take some time and understand all the details of this example!)

Now that we know our hypothesis class to be $\mathcal{H}_{th}$, we should specify our strategy, which consists of the number of samples m we need, and a learning algorithm $\mathcal{A}$ that will process training sample and produce a decision rule. As before let $S = ((x_1, y_1), ..., (x_m, y_m))$ be the training set. As we will see, our choice of learner will not depend on the ϵ, δ specified, but our choice of m will certainly depend on them.

Learning algorithm

Let's start with a suggested learning algorithm. The training data may take the form shown in the three figures below, but not the form shown in the fourth figure, which is forbidden because it does not obey the Realizability Assumption (the assumption that Nature chooses $h \in \mathcal{H}$).

Figure 25: Possible training data for $\mathcal{H}_{th}$

Figure 26: Possible training data for $\mathcal{H}_{th}$

Figure 27: Possible training data for $\mathcal{H}_{th}$

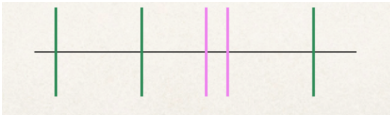


Figure 28: Forbidden training data for $\mathcal{H}_{th}$ - violates the Realizability Assumption

We suggest the following learning algorithm: return hypothesis $h_{\theta_{alg}}(x)$ with

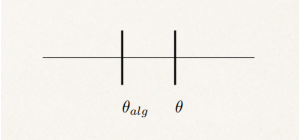
$$\theta_{alg} = \max_{y_i=0} x_i$$

. If $y_i = 1$ for all $i = 1 \dots m$ then return $\theta_{alg} = -\infty$, namely, return a rule that classifies all points to 1. Similarly, if $y_i = 0$ for all $i = 1 \dots m$ then return $\theta_{alg} = +\infty$, namely, return a rule that classifies all points to 0.

Number of Samples

Let $0 < \epsilon, \delta$ be the accuracy and confidence specified in the game. What should be out choice of m? Namely, how many samples are needed to guarantee that the true error is at most ϵ with probability at least $1 - \delta$?

Claim 1 *Fix $0 < \epsilon, \delta$. If $m \geqslant \frac{\log(1/\delta)}{\epsilon}$ then for any distribution $\mathcal{D}$ over the real line, and any choice of labeling threshold function $f_\theta \in \mathcal{H}_{th}$, with probability of at least $1 - \delta$ (over the choice of training sample S of size m), the loss $L_{\mathcal{D},\theta}(h_{\theta_{alg}})$ of the algorithm for learning threshold functions is at most ϵ.*

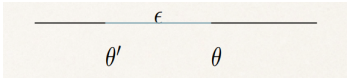


Proof: Fix distribution $\mathcal{D}$ over the domain set. Fix correct hypothesis $f_\theta \in \mathcal{H}_{th}$.
From the properties of the algorithm we know that:

$$\theta_{alg} < \theta$$

Note that the prediction rule produced by the algorithm will be correct for test samples $x < \theta_{alg}$ and $x > \theta$ and is incorrect for $\theta_{alg} < x < \theta$.

If $\mathcal{D}(\{x : -\infty < x < \theta\}) < \epsilon$ then $\mathcal{D}(\{x : \theta_{alg} < x < \theta\}) < \epsilon$ and therefore the true error is *always* (that is, with probability 1, no matter what S or m is) smaller than ϵ and we are done. We will therefore assume that $\mathcal{D}(\{x : -\infty < x < \theta\}) \geqslant \epsilon$ and define θ' such that $\mathcal{D}(\{x : \theta' < x < \theta\}) = \epsilon$.



Note that if there is $(x, y) \in S$ with $\theta' \leqslant x \leqslant \theta$ then the true error is at most ϵ and the probability not to get such a test sample is $(1-\epsilon)^m$. Using $1 - \epsilon \leqslant e^{-\epsilon}$, we see that the term $e^{-\epsilon m}$ would be smaller than δ if $m \geqslant \frac{\log(1/\delta)}{\epsilon}$.

15.4.1 Threshold functions - conclusion

We saw that, for $\mathcal{X} = \mathbb{R}$, $\mathcal{Y} = \{0, 1\}$ and $\mathcal{H} = \mathcal{H}_{th}$ (the hypothesis class of threshold functions), we have a strategy (choice of sample size $m = m(\delta, \epsilon)$ and learning algorithm $\mathcal{A}$) that is **always successful against Nature**, for any values ϵ, δ specified. In other words, for any ϵ, δ, our strategy (which depends on δ, ϵ) specified) is a Probably Approximately correct learner regardless of how Nature plays.

We recall that for $\mathcal{X} = \mathbb{R}$, $\mathcal{Y} = 0, 1$ and $\mathcal{H} = \mathbb{R} \to \mathbb{R}$ there were values of ϵ, δ for which we **could not be successful** regardless of how we played (in fact, we could not be successful for any $\epsilon < 1/2$). So whether we can be successful for any values ϵ, δ seems to be a property of the hypothesis class we choose.

This brings us to the famous definition of a **Probably Approximately Correct (PAC) learnable hypothesis class**.

16 PAC learning

Definition 17 *1. A hypothesis class $\mathcal{H}$ is* **PAC Learnable** *if there exists a function $\tilde{m}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following property: For every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$ that satisfies $L_{\mathcal{D},f}(h^*) = 0$ for some $h^* \in \mathcal{H}$, when running the learning algorithm $\mathcal{A}$ on*

$m \geqslant \tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ *i.i.d. examples generated by* $\mathcal{D}$ *and labeled by* f*, the algorithm returns an hypothesis* $h_S = \mathcal{A}(S)$ *such that, with probability of at least* $1 - \delta$ *(over the choice of the training samples), we have* $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$.

2. *For a PAC learnable hypothesis class, we define the* **Sample Complexity** *of* $\mathcal{H}$ *for specified* ϵ, δ *as the minimal number of samples* $\tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ *required for the definition to hold with respect to* ϵ, δ*. The Sample Complexity function of* $\mathcal{H}$ *is denoted* $m_{\mathcal{H}}(0,1)^2 \to \mathbb{N}$.

Note that PAC learnability is only one possible definition of learning that can account for the fundamental limitations on accuracy and confidence. We could, for example, settle for a specific, "good enough", values of δ, and ϵ instead of requiring that the condition that $m > m(\epsilon, \delta)$ implies $L_{\mathcal{D},f}(h) \leqslant \epsilon$, to hold, for **any** $\delta, \epsilon < 1$.

16.1 Finite hypothesis classes are PAC learnable

In order to understand more about when PAC learning is possible (namely, which hypothesis classes are PAC learnable) let us first consider the case where $\mathcal{H}$ is a *finite* hypothesis class. Finite hypothesis classes can be huge: for example, take $\mathcal{H}$ is all the functions from $\mathcal{X}$ to $\mathcal{Y}$ that can be implemented using a Python program of length at most b, for b fixed and large. Or, take $\mathcal{H}$ to be all the functions from $\mathcal{X}$ to $\mathcal{Y}$ where $|\mathcal{X}|$ and $|\mathcal{Y}|$ are finite.

16.1.1 Empirical Risk Minimization

You might expect that there would be nothing to say in this generality, namely, that in order to design a successful learning algorithm we must pay attention to the details of the specific $\mathcal{X}$ and finite $\mathcal{H}$ at hand. Now comes a big surprise. It turns out that **there is a simple learner that is always successful on finite hypothesis classes** (and on many other hypothesis classes as we will see later).

The idea behind this amazing learning is very simple and natural: try to be as correct as possible on the training data!

Formally, given a training set $S = (x_1, y_1), \ldots, (x_m, y_m)$ we define the **empirical risk** of a candidate prediction rule $h \in \mathcal{H}$ by

$$L_S(h) = \frac{1}{m} |\{i : h(x_i) \neq y_i\}| .$$

Our amazing learning algorithm is simple: on a training sample S, it returns $h \in \mathcal{H}$ that **minimizes the empirical risk** $L_S(h)$**.** In other words,

$$\mathcal{A}_{ERM} : S \mapsto \text{argmin}_{h \in \mathcal{H}} L_S(h) .$$

The minimum may not be unique, in which case the algorithm returns one of the minimizers. Our amazing learner is therefore called **Empirical Risk Minimization (ERM)** learner. We give this important learner its own special notation and denote it by $ERM_{\mathcal{H}}$ instead of Ac_{ERM}.

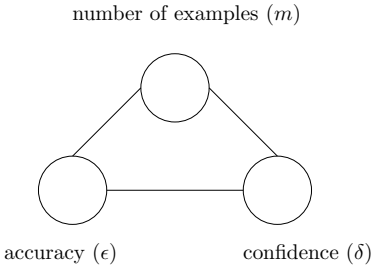
But wait, how do we know that there is a minimum? Note that $L_S(h) \geqslant 0$, and we are minimizing over a finite class, so there is a minimum. In fact, under the assumption that Nature plays $f \in \mathcal{H}$, we know that for any training sample S, $L_S(f) = 0$ for the particular labeling function that Nature chose. So that the lower bound 0 is achievable. In other words, under our assumption that $f \in \mathcal{H}$, the ERM learner will always return a rule h with $y_i = h(x_i)$ for $i = 1, \ldots, m$. Such a rule is called **Consistent** - it is consistent with the training sample.

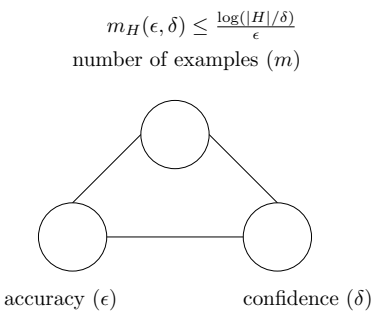
16.1.2 Learning Finite Classes

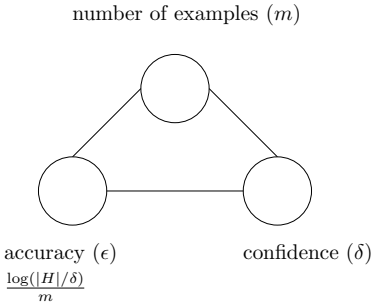
Our main observation concerning finite classes is simple: a **finite** hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ is PAC-learnable, using the ERM rule, with sample complexity at most $\log(|\mathcal{H}|/\delta)/\epsilon$.

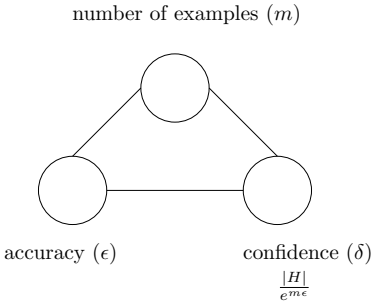
Theorem 5 *Fix $0 < \epsilon, \delta < 1$. If $m \geqslant \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$ then for every $\mathcal{D}, f$, with probability of at least $1 - \delta$ (over the choice of S of size m), $L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) \leqslant \epsilon$.*

The figures below explain schematically the relation between the accuracy, confidence and the sample size for finite classes, as implied by the above theorem.









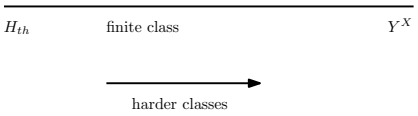
To summarize this section, we have the following: **A finite hypothesis class $\mathcal{H}$ is PAC learnable using the ERM learning algorithm, and has a sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leqslant \log(|\mathcal{H}|/\delta)/\epsilon$ samples.**

Several natural questions may come to mind.

- Is the bound $m_{\mathcal{H}}(\epsilon, \delta) \leqslant \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$ tight? Can the ERM learner, or an other learner, be Probably Approximately correct (with accuracy ϵ and confidence δ) using fewer than $\frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$ samples?
- What happens when noise is present so the y's are not deterministically determined by x?
- What happens when our hypothesis class is infinite?

Consider the last question. As we have seen, the class of threshold functions over $\mathbb{R}$, $\mathcal{H}_{th}$, in spite of being infinite, *is* PAC learnable, with sample complexity $m_{\mathcal{H}_{th}}(\epsilon, \delta) \leqslant \frac{\log(1/\delta)}{\epsilon}$, which is obtained by using the $ERM_{\mathcal{H}_{th}}$ learning rule. So $\mathcal{H}_{th}$ appears to be simple to learn. Can we explain why? Somehow, the hypothesis class $\mathcal{H}_{tr}$ is **small** or **simple** and the hypothesis class $\mathcal{Y}^{\mathcal{X}}$ is **large** or **complicated**.

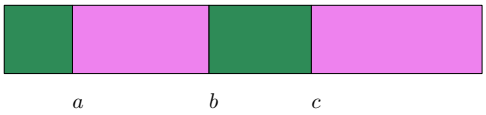
What we need in order to answer the above questions systematically is some sort of a *complexity measure* with which we can order classes by their difficulty along the complexity axis shown in the figure below.



For example, consider the **Two-Intervals** hypothesis class, defined by

$$\mathcal{X} = \mathbb{R}, \quad \mathcal{H} = \{h_{a,b,c} : a < b < c \in \mathbb{R}\},$$

where $h_{a,b,c}(x) = 1$ if $x \in [a, b]$ or $x \geqslant c$ and $h_{a,b,c}(x) = 0$ elsewhere.

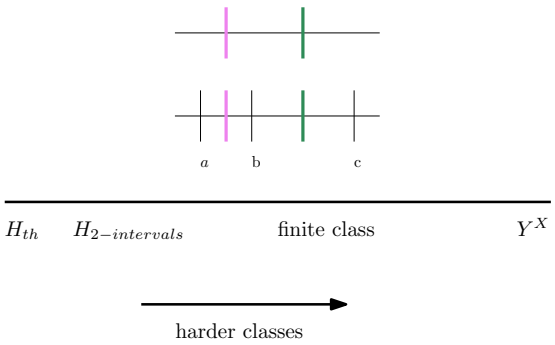


The following figures demonstrate this point. Suppose we would like to learn with threshold functions the following sample (pink = 1, green = 0)

A possible answer would be:

However, if $x_1 < x_2$ and $y_1 = 1, y_2 = 0$, can not be learned by $\mathcal{H}_{th}$, although it can be learned with 2-intervals:

Indeed, we somehow feel that the **2-Interval** class has larger complexity than that of $\mathcal{H}_{th}$ but smaller than that of finite classes.



17 VC Dimension

VC-Dimension is the definition of complexity of an hypothesis class we are looking for. This is a **combinatorial measure of complexity** of a function class. What does "combinatic measure" mean here? It means that VC-dimension is just based on counting stuff, so that $\mathcal{X}$ can be any set, with no additional structure. In particular, the VC-dimeniosn of $\mathcal{H} \subset \{\pm 1\}^{\mathcal{X}}$ is defined even if $\mathcal{X}$ does not need any geometric or algebraic structure. VC-dimension is interesting as it provides a decisive characterization of hypothesis classes that are "simple enough" to learn (in the sense of PAC learnability), versus hypothesis classes that are "too complicated" to learn. It also provides a decisive characterization of $m_{\mathcal{H}}$, the sample complexity of a "simple" hypothesis class $\mathcal{H}$.

17.1 Motivation

Suppose we got a training set $S = (x_1, y_1), \ldots, (x_m, y_m)$ and were able to fully explain the labels using a hypothesis from a class $\mathcal{H}$, namely, to find a function $h \in \mathcal{H}$ with empirical risk $L_S(h) = 0$. Suppose that, only to see what will happen, we deliberately corrupt our sample S by changing the labels $\{y_i\}$ - call the corrupt sample S'. Suppose that we **also** succeed in explaining S' using a different hypothesis from the same class $\mathcal{H}$, namely, find another function $h' \in \mathcal{H}$ with $L_{S'}(h') = 0$. If we can do that, no matter what corrupt labels we choose, it means that something isn't right: how can we hope to generalize based on a training sample S if, regardless of the labels in S, we can find $h \in \mathcal{H}$ with $L_S(h) = 0$?

Definition: Let $C \subset \mathcal{X}$ be a subset of the sample space and let $h : \mathcal{X} \to \mathcal{Y}$ be some hypothesis. We define the **restriction** of h to C, denoted $h_C : C \to \mathcal{Y}$, by $h_C(x) = h(x)$, for $x \in C$.

Here is a crucial observation: Suppose that $\mathcal{H}$ contains **all** functions over some set $C \subset \mathcal{X}$ of size m, in the sense that $\{h_C \,|\, h \in \mathcal{H}\} = (C \to \mathcal{Y})$. Then we cannot find a Probably Approximately correct learner that uses $m/2$ or fewer training samples.
Why? Go back to the "proof" of the No Free Lunch theorem we saw above. Suppose that indeed there exists a set $C \subset \mathcal{X}$ of size m such that $\{h_C \,|\, h \in \mathcal{H}\} = (C \to \mathcal{Y})$. We play our game against Nature, choose a learner $\mathcal{A}$, and choose a training sample size of $m/2$ or less. Our learner specifies that if it hasn't seen a point $x \in \mathcal{X}$ in the training set, the output rule h_S will predict $g(x)$ for some $g : \mathcal{X} \to \mathcal{Y}$ we specify - we just make sure that the resulting h_S will belong to $\mathcal{H}$. Now, as in the "proof" of the No Free Lunch theorem, Nature plays the distribution $\mathcal{D}$ that is uniform over C, and makes an evil choice to play $f(x) = -g(x)$ for $x \in C$ (Since $\mathcal{D}$ is

supported over C, Nature doesn't really care how her f is defined outside C). **This is a legal move for Nature since for every function $\tilde{f} : C \to \mathcal{X}$ there is a hypothesis $f \in \mathcal{H}$ with $f_C(x) = \tilde{f}(x)$, $x \in C$.** Again as in the "proof", our learner fails - regardless of the training sample of size $m/2$, the loss will be $1/2$.

We see that, as long as $\mathcal{H}$ contains **any** set C of size $2m$ with the property that $\{h_C \,|\, h \in \mathcal{H}\} = (C \to \mathcal{Y})$, then we cannot learn with a training sample of size m. It follows that the **maximal size** of such a set C in $\mathcal{H}$ is a **critical quantity**: (i) it gives us a lower bound on $m_{\mathcal{H}}$, the minimal sample size needed, and (ii) if the maximal size is ∞, namely, if for any $m \in \mathbb{N}\mathcal{X}$ contains such as set C with $|C| > m$, the $\mathcal{H}$ is not PAC-learnable!

17.2 Formal Definition

Let $C = \{x_1, \ldots, x_{|C|}\} \subset \mathcal{X}$ and let $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to C, namely, $\mathcal{H}_C = \{h_C : h \in \mathcal{H}\}$. We are working with $\mathcal{Y} = \{\pm 1\}$, so we can represent each h_C as the vector $(h(x_1), \ldots, h(x_{|C|})) \in \{\pm 1\}^{|C|}$. Therefore the total number of possible such vectors is $2^{|C|}$, so that

$$|\mathcal{H}_C| \leqslant 2^{|C|}\,.$$

We will say that $\mathcal{H}$ **shatters** C if $|\mathcal{H}_C| = 2^{|C|}$.

Definition 18 *The VC dimension of the hypothesis class Hc is defined as*

$$VCdim(\mathcal{H}) = \max\{|C| \;:\; \mathcal{H} \;\; \textit{shatters} \;\; C\}\,,$$

that is, the VC dimension is the maximal size of a set $C \subset \mathcal{X}$ such that $\{h_C \,|\, h \in \mathcal{H}\} = (C \to \mathcal{Y})$.

We interpret this as the maximal size of a set $C \subset X$ such that $\mathcal{H}$ gives no prior knowledge on label functions restricted to C.

According to the above definition, in order to show that $VCdim(\mathcal{H}) = d$ we need to show that:

1. There exists a set C of size d which is shattered by $\mathcal{H}$.
2. Every set C of size k with $k \geqslant d + 1$ is not shattered by $\mathcal{H}$.

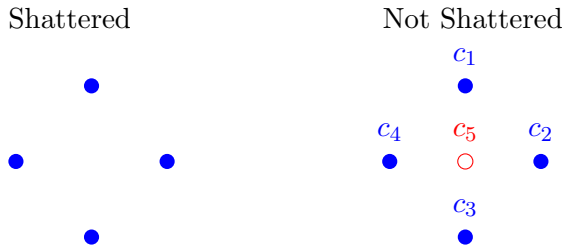
17.3 Exercises to help you understand the definition of VC-dimension

Make sure to solve all these exercises. They will help you understand the definition of VC-dimension.

17.3.1 Axis aligned rectangles

Consider the **Axis aligned rectangles** hypothesis class over the sample space $\mathcal{X} = \mathbb{R}^2$. We define $\mathcal{H} = \{h_{(a_1,a_2,b_1,b_2)} : a_1 < a_2 \text{ and } b_1 < b_2\}$, where $h_{(a_1,a_2,b_1,b_2)}(x_1, x_2) = 1$ if $x_1 \in [a_1, a_2]$, and $x_2 \in [b_1, b_2]$, and $h_{(a_1,a_2,b_1,b_2)}(x_1, x_2) = 0$ otherwise. (Convince yourself that a function in this hypothesis class is an indicator of a finite open rectangle aligned with the canonical basis of $\mathbb{R}^2$.)

Verify that:



Exercise: show that no set of 5 points can be shattered by the Axis aligned rectangles class. Hint: note that the 3 points (x_k, y_k), (x_i, y_i), and $(x_{k'}, y_{k'})$ can not be shattered if $x_k \leqslant x_i \leqslant x_{k'}$ and $y_k \leqslant y_i \leqslant y_{k'}$.

17.3.2 Finite classes

Exercise:

- Show that the VC dimension of a finite $\mathcal{H}$ is at most $\log_2(|\mathcal{H}|)$.
- Assume $\mathcal{H}$ is finite. Show that there can be arbitrary gap between $VCdim(\mathcal{H})$ and $\log_2(|\mathcal{H}|)$, namely, construct a finite hypothesis class $\mathcal{H}$ over some sample space $\mathcal{X}$ with $VCdim(\mathcal{H}) = \log_2(|\mathcal{H}|)$ and another finite hypothesis class with $VCdim(\mathcal{H}) = 1$.

17.3.3 Half-spaces through the origin

Consider the sample space $\mathcal{X} = \mathbb{R}^d$ and the hypothesis class of half-spaces through the origin $\mathcal{H} = \{\mathbf{x} \mapsto sign(\langle \mathbf{w}, \mathbf{x} \rangle) : \mathbf{w} \in \mathbb{R}^d\}$.

Exercise:

- Show that $\{\mathbf{e}_1, \ldots, \mathbf{e}_d\}$ is shattered
- Show that any $d + 1$ points cannot be shattered (hint: consider the standard basis vectors...)
- What is $VCdim(\mathcal{H})$?

Lecture 5: PAC Theory of Statistical Learning, Part II

18 Recap of PAC Theory so far

Make sure you've read the previous handout (PAC Part I). Recall that we are developing a theory of "learning from previous examples" or "generalizing from previous examples", which suggests a mathematical model for what it means to learn, what is learnable and what isn't learnable, how to learn when learning is possible, and what's the minimal number of training samples we need to learn. All for **supervised batch learning**.

Our framework and The Learning Game

Framework. Our task is to design a learning algorithm (a "learner"), which we denote by $\mathcal{A}$. For a given sample size m, $\mathcal{A}$ is a map from the training sample

$$S = ((x_1, y_1), .., (x_m, y_m)) \in (\mathcal{X} \times \mathcal{Y})^m$$

where each $\mathbf{x}_i$ to an **hypothesis class** $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$. We write $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$ so that $\mathcal{A} : S \mapsto h_S \in \mathcal{H}$.

We're thinking of classification problems now, so in this lecture $\mathcal{Y} = \{\pm 1\}$, but everything here can be generalized. The **data generation model** we assume is probabilistic. We assume that there is an unknown distribution $\mathcal{D}$ over $\mathcal{X}$. We assume that each sample - both in the training set and in the test set - is sampled according to $\mathcal{D}$ independently from any other sample before of after it. For the labels y, we assume a function $f : \mathcal{X} \to \mathcal{Y}$ such that for each sample $x \in \mathcal{X}$, the corresponding label is $y = f(x)$. We further assume (the **"realizability assumption"**) that f is $\mathcal{D}$-almost-everywhere equal to some $h^* \in \mathcal{H}$, namely that $\mathcal{D}(\{x \in \mathcal{X} \mid f(x) = h^*(x)\}) = 1$. In particular, for our training data, we have $y_i = f(x_i)$ for $i = 1, \ldots, m$. Finally, for a candidate prediction rule $h \in \mathcal{H}$ that our learner may produce, we will measure the generalization performance of h_S - how well it will perform on future unseen samples - by simply using the expected misclassification rate

$$L_{\mathcal{D},f}(h) \stackrel{\text{def}}{=} \mathop{\mathbb{P}}_{x \sim \mathcal{D}}[h(x) \neq f(x)] .$$

Note that using this notation, the Realizability Assumtion can be written like this: we assume that the labeling function $f : \mathcal{X} \to \mathcal{Y}$ is such that $\exists h^* \in \mathcal{H}$ with $L_{\mathcal{D},f}(h^*) = 0$.

The Learning Game. The definition of PAC-learnability is more easily understood by thinking about supervised batch learning as a "game between us and Nature". Fix desired accuracy $\epsilon > 0$ and confidence $\delta > 0$. Fix an hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$. We play a game against Nature, with random payoff.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{H}$. Both m and $\mathcal{A}$ can depend on (ϵ, δ).
- Nature knows our strategy, and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X}$, and a label function **from the specified hypothesis class** $f \in \mathcal{H}$. That is, Nature's strategy can depend on $\epsilon, \delta, \mathcal{H}$ specified, and also on the $m, \mathcal{A}$ we chose.

- A sample S of size m is drawn according to $\mathcal{D}$ and is labeled according to f
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$. Note that $h_S \in \mathcal{H}$.
- The payoff is $L_{\mathcal{D},f}(h_S)$. It is random since S is random and therefore h_S is random.
- We are going to assume Nature is "cruel" and does her best to win. So we'll look for learners $\mathcal{A}$ that have a **guaranteed maximal loss** $L_{\mathcal{D},f}(h)$ **for any** strategy $\mathcal{D}, f$ that Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,|\, L_{\mathcal{D},f}(h_S) \leqslant \epsilon\}$. If this probability is found to be larger than $1-\delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ϵ and confidence δ - against Nature's best strategy - **we say that we've been successful (with regards to the parameters ϵ, δ) and hypothesis class $\mathcal{H}$.**

We saw that the definition of PAC learnability of an hypothesis class $\mathcal{H}$ (coming next) is equivalent to the following statement: We are able to "win" against Nature, for any $\delta, \epsilon > 0$, regardless of how Nature plays.

PAC Learnability and Sample Complexity of an hypothesis class $\mathcal{H}$

Slowly and gradually, in the last lecture we unpacked the following definitions:

Definition 19
1. *A hypothesis class $\mathcal{H}$ is* **PAC Learnable** *if there exists a function $\tilde{m}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a learning algorithm $\mathcal{A}$ with the following property: For every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$ over $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{\pm 1\}$ that satisfies $L_{\mathcal{D},f}(h^*) = 0$ for some $h^* \in \mathcal{H}$, when running the learning algorithm $\mathcal{A}$ on $m \geqslant \tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ i.i.d. examples generated by $\mathcal{D}$ and labeled by f, the algorithm returns an hypothesis $h_S = \mathcal{A}(S)$ such that, with probability of at least $1-\delta$ (over the choice of the training samples), we have $L_{\mathcal{D},f}(h_S) \leqslant \epsilon$.*
2. *For a PAC learnable hypothesis class, we define the* **Sample Complexity** *of $\mathcal{H}$ for specified ϵ, δ as the minimal number of samples $\tilde{m}_{\mathcal{H}}(\epsilon, \delta)$ required for the definition to hold with respect to ϵ, δ. The Sample Complexity function of $\mathcal{H}$ is denoted $m_{\mathcal{H}}(0,1)^2 \to \mathbb{N}$.*

We defined a Probably Approximately Correct learner and saw that if an hypothesis class $\mathcal{H}$ is PAC-learnable, this just means that for any $0 < \epsilon, \delta < 1$ we are able to find a Probably Approximately correct learner $\mathcal{A}$ with accuracy ϵ and confidence δ, regardless of $\mathcal{D}, f$ chosen by Nature.

VC Dimension

Definition: Let $C \subset \mathcal{X}$ be a subset of the sample space and let $h : \mathcal{X} \to \mathcal{Y}$ be some hypothesis. We define the **restriction** of h to C, denoted $h_C : C \to \mathcal{Y}$, by $h_C(x) = h(x)$, for $x \in C$.

Crucial observation: As long as $\mathcal{H}$ contains **any** set C of size $2m$ with the property that $\{h_C \,|\, h \in \mathcal{H}\} = \mathcal{Y}^C$, then we cannot learn with a training sample of size m. It follows that

the **maximal size** of such a set C in $\mathcal{H}$ is a **critical quantity**: (i) it gives us a lower bound on $m_{\mathcal{H}}$, the minimal sample size needed, and (ii) if the maximal size is ∞, namely, if for any $m \in \mathbb{N}\mathcal{X}$ contains such as set C with $|C| > m$, the $\mathcal{H}$ is not PAC-learnable!

This is the intuition behind the **lower bound** in the Fundamental Theorem of Statistical Learning: When trying to learn a function from an hypothesis class $\mathcal{H}$, the **minimal** number of samples needed to learn - using **any** learner (not just ERM) - is related to the maximal size of a set $C \subset \mathcal{X}$ with the property that $\{h_C \,|\, h \in \mathcal{H}\} = \mathcal{Y}^C$.

Formal Definition

Let $C = \{x_1, \ldots, x_{|C|}\} \subset \mathcal{X}$ and let $\mathcal{H}_C$ be the restriction of $\mathcal{H}$ to C, namely, $\mathcal{H}_C = \{h_C : h \in \mathcal{H}\}$. Observe that, always, $|\mathcal{H}_C| \leqslant 2^{|C|}$. The case where the upper bound is obtained is the one most interesting to us: We will say that $\mathcal{H}$ **shatters** C if $|\mathcal{H}_C| = 2^{|C|}$.

Definition 20 *The VC dimension of the hypothesis class Hc is defined as*

$$VCdim(\mathcal{H}) = \max\{|C| \;:\; \mathcal{H} \;\; shatters \;\; C\}\,,$$

that is, the VC dimension is the maximal size of a set $C \subset \mathcal{X}$ such that $\{h_C \,|\, h \in \mathcal{H}\} = \mathcal{Y}^C$.

We interpret this as the maximal size of a set $C \subset X$ such that $\mathcal{H}$ gives no prior knowledge on label functions restricted to C.

The **lower bound** part of the Fundamental Theorem of Statistical Learning gives an explicit connection between $VCdim(\mathcal{H})$ (maximal size of a subset of X that is shuttered by $\mathcal{H}$) and the minimal number of samples required to produce a PAC learner for $\mathcal{H}$ with confidence δ and accuracy ϵ ($m_{\mathcal{H}}(\epsilon, \delta)$ - the sample complexity of $\mathcal{H}$ at ϵ, δ): there is a universal constant C_1 such that

$$C_1 \frac{d + \log(1/\delta)}{\epsilon} \leqslant m_{\mathcal{H}}(\epsilon, \delta)$$

Examples

According to the above definition, in order to show that $VCdim(\mathcal{H}) = d$ we need to show that:

1. There exists a set C of size d which is shattered by $\mathcal{H}$.
2. Every set C of size $d + 1$ is not shattered by $\mathcal{H}$.

Threshold class

Consider first $\mathcal{H}_{th}$. The set $\{0\}$ is shattered by $\mathcal{H}_{th}$ since $x = 0$ can receive both the label 0 and 1, depending on the location of the threshold θ.

In contrast to the above, no two points, x_1 and x_2 can be shattered because if $x_1 < x_2$ then $y_1 \leqslant y_2$ and therefore the pair of labels $y_1 = 1, y_2 = 0$ is forbidden.

One-Interval

Consider the **One-Interval** hypothesis class over the sample space $\mathcal{X} = \mathbb{R}$. We define $\mathcal{H} = \{h_{a,b} : a < b \in \mathbb{R}\}$, where $h_{a,b}(x) = 1$ if $x \in [a, b]$, and $h_{a,b}(x) = 0$ otherwise. Now, take for example the two points $\{0, 1\}$. We can place the interval over both, over any one of them, or outside $[0, 1]$. Therefore, $\{0, 1\}$ is shattered. However, any three points cannot be shattered: Let $x_1 < x_2 < x_3$, then no single interval can cover x_1 and x_3 without containing also x_2 and therefore the labeling $y_1 = 1, y_2 = 0, y_3 = 1$ is forbidden.

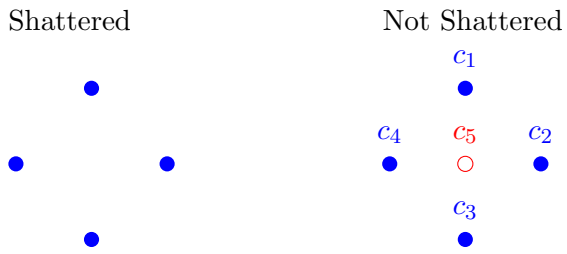
Exercises to help you understand the definition of VC-dimension

Make sure to solve all these exercises. They will help you understand the definition of VC-dimension.

Axis aligned rectangles

Consider the **Axis aligned rectangles** hypothesis class over the sample space $\mathcal{X} = \mathbb{R}^2$. We define $\mathcal{H} = \{h_{(a_1,a_2,b_1,b_2)} : a_1 < a_2 \text{ and } b_1 < b_2\}$, where $h_{(a_1,a_2,b_1,b_2)}(x_1, x_2) = 1$ if $x_1 \in [a_1, a_2]$, and $x_2 \in [b_1, b_2]$, and $h_{(a_1,a_2,b_1,b_2)}(x_1, x_2) = 0$ otherwise. (Convince yourself that a function in this hypothesis class is an indicator of a finite open rectangle aligned with the canonical basis of $\mathbb{R}^2$.)

Verify that:



Exercise: show that no set of 5 points can be shattered by the Axis aligned rectangles class. Hint: note that the 3 points (x_k, y_k), (x_i, y_i), and $(x_{k'}, y_{k'})$ can not be shattered if $x_k \leqslant x_i \leqslant x_{k'}$ and $y_k \leqslant y_i \leqslant y_{k'}$.

Finite classes

Exercise:

- Show that the VC dimension of a finite $\mathcal{H}$ is at most $\log_2(|\mathcal{H}|)$.
- Assume $\mathcal{H}$ is finite. Show that there can be arbitrary gap between $VCdim(\mathcal{H})$ and $\log_2(|\mathcal{H}|)$, namely, construct a finite hypothesis class $\mathcal{H}$ over some sample space $\mathcal{X}$ with $VCdim(\mathcal{H}) = \log_2(|\mathcal{H}|)$ and another finite hypothesis class with $VCdim(\mathcal{H}) = 1$.

Half-spaces through the origin

Consider the sample space $\mathcal{X} = \mathbb{R}^d$ and the hypothesis class of half-spaces through the origin $\mathcal{H} = \{\mathbf{x} \mapsto sign(\langle \mathbf{w}, \mathbf{x} \rangle) : \mathbf{w} \in \mathbb{R}^d\}$.

Exercise:

- Show that $\{\mathbf{e}_1, \ldots, \mathbf{e}_d\}$ is shattered
- Show that any $d + 1$ points cannot be shattered (hint: consider the standard basis vectors...)
- What is $VCdim(\mathcal{H})$?

19 Finite Hypothesis Classes are PAC learnable

We talked about finite hypothesis classes in the previous lecture, and stated a theorem whereby finite hypothesis classes are PAC learnable, but did not prove it. Let's recall the details and prove the theorem.

We recall that a finite hypothesis classes can be huge: for example, take $\mathcal{H}$ is all the functions from $\mathcal{X}$ to $\mathcal{Y}$ that can be implemented using a Python program of length at most b, for b fixed and large. Or, take $\mathcal{H}$ to be all the functions from $\mathcal{X}$ to $\mathcal{Y}$ where $|\mathcal{X}|$ and $|\mathcal{Y}|$ are finite.

19.0.1 Empirical Risk Minimization

It turns out that **there is a simple learner that is always successful on finite hypothesis classes** (and on many other hypothesis classes as we will see later).

The idea behind this amazing learning is very simple and natural: try to be as correct as possible on the training data!

Formally, given a training set $S = (x_1, y_1), \ldots, (x_m, y_m)$ we define the **empirical risk** of a candidate prediction rule $h \in \mathcal{H}$ by

$$L_S(h) = \frac{1}{m} |\{i : h(x_i) \neq y_i\}| .$$

Our amazing learning algorithm is simple: on a training sample S, it returns $h \in \mathcal{H}$ that **minimizes the empirical risk $L_S(h)$.** In other words,

$$\mathcal{A}_{ERM} : S \mapsto \text{argmin}_{h \in \mathcal{H}} L_S(h) = \frac{1}{m} .$$

The minimum may not be unique, in which case the algorithm returns one of the minimizers. Our amazing learner is therefore called **Empirical Risk Minimization (ERM)** learner. We give this important learner its own special notation and denote it by $ERM_{\mathcal{H}}$ instead of Ac_{ERM}.

But wait, how do we know that there is a minimum? Note that $L_S(h) \geqslant 0$, and we are minimizing over a finite class, so there is a minimum. In fact, under the assumption that Nature plays $f \in \mathcal{H}$, we know that for any training sample S, $L_S(f) = 0$ for the particular labeling function that Nature chose. So that the lower bound 0 is achievable. In other words, under our assumption that $f \in \mathcal{H}$, the ERM learner will always return a rule h with $y_i = h(x_i)$ for $i = 1, \ldots, m$. Such a rule is called **Consistent** - it is consistent with the training sample.

19.0.2 Learning Finite Classes

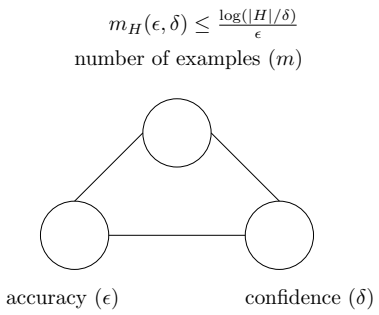
Our main observation concerning finite classes is simple: a **finite** hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ is PAC-learnable, using the ERM rule, with sample complexity at most $\log(|\mathcal{H}|/\delta)/\epsilon$.

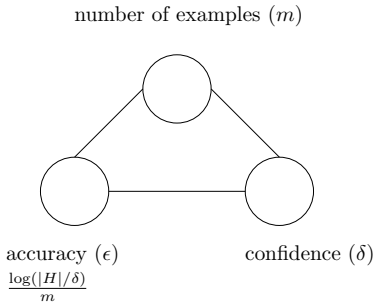
Theorem 6 *Fix $0 < \epsilon, \delta < 1$. If*

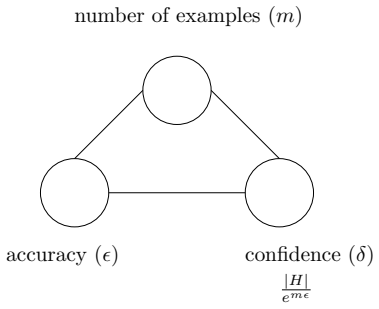
$$m \geqslant \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$$

then for every $\mathcal{D}$ over $\mathcal{X}$ and every f such that $\exists h^* \in \mathcal{H}$ with $L_{D,f}(H^*) = 0$, with probability of at least $1 - \delta$ (over the choice of S of size m), $L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) \leqslant \epsilon$.

In words, the theorem states that a finite hypothesis class $\mathcal{H}$ is PAC learnable with sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leqslant \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$. The figures below explain schematically the relation between the accuracy, confidence and the sample size for finite classes, as implied by the above theorem.







Let us prove the theorem in full (more details can be found in *Understanding Machine Learning* Ch. 2.3.1). For each S our algorithm chooses a hypothesis $ERM_{\mathcal{H}}(S)$ Let $\mathcal{H}_B$ be the set of "bad" hypotheses,

$$\mathcal{H}_B = \{h \in \mathcal{H} : L_{\mathcal{D},f}(h) > \epsilon\}$$

and let $S|_x = (x_1, \ldots, x_m)$ be the instances of the training set. We would like to prove:

$$\mathcal{D}^m(\{S|_x : ERM_{\mathcal{H}}(S) \in \mathcal{H}_B\}) \leqslant \delta$$

Let us denote by M the set of "misleading" samples,

$$M = \{S|_x : \exists h \in \mathcal{H}_B, L_S(h) = 0\}$$

that is, for every input sample S in M there exists an h that in spite of being perfectly correct on that sample, and therefore *could be* chosen by the ERM algorithm as a possible output for that input, its global error is larger than ϵ. In particular, all the S's in the set $\{S|_x : ERM_{\mathcal{H}}(S) \in \mathcal{H}_B\}$ belong to M because these are the samples for which our *specific* algorithm chooses a "bad" h (that is, one with $L_{\mathcal{D},f}(h) > \epsilon$, which means it is in $\mathcal{H}$) and being an ERM algorithm, the h it outputs must satisfy $L_S(h) = 0$. Therefore:

$$\{S|_x : L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) > \epsilon\} \subseteq M\,.$$

We can rearrange the samples in M according to the h's they share by writing M as:

$$M = \bigcup_{h \in \mathcal{H}_B} \{S|_x : L_S(h) = 0\}$$

In this representation, samples in M which share several h's will be counted more than once but since M is a set, this has no effect. Combining the last two relations we have

$$\{S|_x : L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) > \epsilon\} \subseteq M$$

and using the **Union Bound**, that is, the fact that for any two sets A, B and a distribution $\mathcal{D}$ we have $\mathcal{D}(A \cup B) \leqslant \mathcal{D}(A) + \mathcal{D}(B)$, we get

$$\mathcal{D}^m(\{S|_x : L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) > \epsilon\}) \leqslant \sum_{h \in \mathcal{H}_B} \mathcal{D}^m(\{S|_x : L_S(h) = 0\})\,.$$

Given an h, $\mathcal{D}^m(\{S|_x : L_S(h) = 0\}$ is the probability to pull a sample over which h is perfectly correct (has the right labels for all x_i's). Since the x_i's are drawn independently, this probability equals the probability that h will be correct for each of the x_i's separately. But the probability that h will be *incorrect* for a random x is exactly $L_{\mathcal{D},f}(h)$. So the probability that h will be correct for m such x's is $(1 - L_{\mathcal{D},f}(h))^m$. We therefore have, for any h,

$$\mathcal{D}^m(\{S|_x : L_S(h) = 0\}) = (1 - L_{\mathcal{D},f}(h))^m \quad \forall h\,.$$

In particular, if $h \in \mathcal{H}_B$ then $L_{\mathcal{D},f}(h) > \epsilon$ and therefore

$$\mathcal{D}^m(\{S|_x : L_S(h) = 0\}) < (1 - \epsilon)^m \quad \forall h \in \mathcal{H}_B\,.$$

Substituting this bound above we obtain

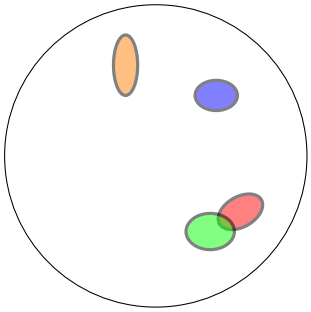
$$\mathcal{D}^m(\{S|_x : L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) > \epsilon\}) \leqslant \sum_{h \in \mathcal{H}_B} (1 - \epsilon)^m < |\mathcal{H}_B|\,(1 - \epsilon)^m \leqslant |\mathcal{H}|\,(1 - \epsilon)^m\,.$$

Finally, using $1 - \epsilon \leqslant e^{-\epsilon}$ we conclude:

$$\mathcal{D}^m(\{S|_x : L_{\mathcal{D},f}(ERM_{\mathcal{H}}(S)) > \epsilon\}) \;<\; |\mathcal{H}|\,e^{-\epsilon\,m}\,.$$

The right-hand side would be $\leqslant \delta$ if $m \geqslant \frac{\log(|\mathcal{H}|/\delta)}{\epsilon}$ and therefore we are done. ■

The figure below illustrates the use of the union bound. Each point is a possible sample $S|_x$. Each colored oval represents misleading samples for some $h \in \mathcal{H}_B$. The probability mass of each such oval is at most $(1-\epsilon)^m$. But, the algorithm might err if it samples $S|_x$ from any of these ovals.



To summarize this section, we have shown the following: **A finite hypothesis class $\mathcal{H}$ is PAC learnable using the ERM learning algorithm, and has a sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leqslant \log(|\mathcal{H}|/\delta)/\epsilon$ samples.**

20 The Fundamental Theorem of Statistical Learning

We worked hard to understand the two fundamental definitions above: VC-Learnability (and sample complexity) of an hypothesis class, and VC-dimension of an hypothesis class.

Along the way, we saw some connections between these two definitions:

- We saw that a finite hypothesis class is PAC-learnable (using the ERM learner) with sample complexity
$$m_{\mathcal{H}}(\epsilon, \delta) \leqslant \frac{\log(|\mathcal{H}|) + \log(1/\delta)}{\epsilon}$$
and also that in this case $VCdim(\mathcal{H}) \leqslant \log(|\mathcal{H}|)$. Somehow it seems that $VCdim(\mathcal{H})$ is related to an upper bound on $\mathcal{H}$ for finite hypothesis classes.

- We saw, using a loose argument (the "proof" of No Free Lunch), that $VCdim(\mathcal{H})$ also gives a **lower bound** on the sample complexity $m_{\mathcal{H}}$ of an hypothesis class $\mathcal{H}$, and that if $VCdim(\mathcal{H})$ is infinite, that class is not PAC-Learnable.

The surprising, shocking, wonderful truth is that VC-dimension gives a complete and full characterization of PAC-learnabilty and sample complexity of an hypothesis class, and gives a decisive answer to all the questions we posed at various stages along the way (such as which classes are PAC-learnable, with what sample complexity, and with what algorithm, and is there an algorithm that uses the minimal possible sample size.)

The Fundamental Theorem of Statistical Learning states as follows:

- The PAC-lernability of an hypothesis class is characterized by the **VC dimension**, a combinatorial property of the class that denotes the maximal size of a sample that can be shuttered by the class. The characterization is as follows: an hypothesis class is PAC-learnable **if and only if** its VC-dimension is finite.

- When $VCdim(\mathcal{H})$ is finite (so that $\mathcal{H}$ is PAC-learnable and we can ask about its sample complexity), its sample complexity is basically

$$m_H(\epsilon, \delta) \sim \frac{VCdim(\mathcal{H}) + \log(1/\delta)}{\epsilon}$$

  up to come constants.

- The ERM learning rule is a generic (near) optimal learner, in the sense that when a hypothesis class is PAC-learnable, ERM using

$$m(\epsilon, \delta) \sim \frac{VCdim(\mathcal{H}) \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

  is a Probably Approximately correct learner with accuracy ϵ and confidence δ.

The formal statement of the theorem is as follows: (see Understanding Machine Learning book, ch. 6.6)

Theorem 7 (The Fundamental Theorem of Statistical Learning) *Let $\mathcal{H}$ be a hypothesis class of binary classifiers with VC-dimension $d \leqslant \infty$. Then, $\mathcal{H}$ is PAC-learnable if and only if $d < \infty$. In this case: (1) there there are absolute constants C_1, C_2 such that the sample complexity of $\mathcal{H}$ satisfies*

$$C_1 \frac{d + \log(1/\delta)}{\epsilon} \leqslant m_{\mathcal{H}}(\epsilon, \delta) \leqslant C_2 \frac{d \log(1/\epsilon) + \log(1/\delta)}{\epsilon}$$

(2) Furthermore, the upper bound on sample complexity is achieved by the ERM learner.

We already saw the intuition behind the lower bound - how the VC-dimension $VCdim(\mathcal{H})$ is related to the **minimal** number of training samples needed to PAC-learn the hypothesis class $\mathcal{H}$ - using **any** learning algorithm.

To understand the upper bound, we need to understand how is it that the ERM learner is a **generic learning algorithm** that is able to PAC-learn $\mathcal{H}$ with a training sample size again related to the VC-dimension $VCdim(\mathcal{H})$.

Before we discuss the upper bound, let us extend our theoretic framework and make it much more flexible and realistic.

21 Agnostic PAC: Extending our theoretical framework

The theoretical framework we developed is not so satisfying, and is not such a great theoretical model for "real" learning problems, for a few reasons:

- **It doesn't model noisy labels.** We have no model for measurement errors. In practice, sometimes even though the label for some $\mathbf{x} \in \mathcal{X}$ "should" have been 1 (say), it can be measured as -1 due to measurement mistake, noise, etc. We want our framework to allow for the fact that we may, with low probability, observe the point $x \in \mathcal{X}$ twice, and get two different labels - namely, observe $(x, +1)$ and later sample again and observe $(x, -1)$ - due to noise.

- **The realizability assumption is unrealistic.** We define the hypothesis class. It is unrealistic to restrict Nature to choose from the hypothesis class that we defined. Nature will do as she likes.
- **Limited to misclassification loss.** In our framework, we could only measure the classifier performance using the misclassification loss (otherwise known as the $0-1$ loss). We would like to be able to measure performance using any loss function we like.

We now introduce an improved theoretical framework, sometimes know as **Agnostic PAC**. It improves on the PAC learning framework and solves the problems above: (i) it allows measurement noise, (ii) it removes the realizability assumption, and (iii) it allow us to specify any loss function. The good thing is, as we mention below, the fundamental theorem of statistical learning holds in the Agnostic PAC framework as well.

21.1 Moving from a probability distribution over $\mathcal{X}$ to a joint probability distribution over $\mathcal{X} \times \mathcal{Y}$

Our first step will be to change the probability distribution $\mathcal{D}$. For far, $\mathcal{D}$ was a probability distribution over the sample space $\mathcal{X}$, and the labels were determined - deterministically - using the labeling function f. In our upgraded framework, $\mathcal{D}$ will be a probability distribution over $\mathcal{X} \times \mathcal{Y}$. This means that when we draw a new random example (x, y) - whether for the training sample S or as a test sample - there is randomness in **both** x and y, and, crucially, it's a **joint** randomness.

We can factor $\mathcal{D}$ in two ways, conditioning on x or on y. Both are useful for our understanding. Let (X, Y) be a random variable taking valued in $\mathcal{X} \times \mathcal{Y}$ whose distribution is $\mathcal{D}$.

- $\mathbb{P}(X = x, Y = y) = \mathbb{P}(X = x)\,\mathbb{P}(Y = y|X = x)$. From this perspective, this is a direct generalization of our previous framework, where x was random and $y = f(x)$. Indeed, we draw x from the marginal distribution with probability $\mathbb{P}(X = x)$ - as we did in the previous framework. We then choose a corresponding label according to the conditional probability $\mathbb{P}(Y = y|X = x)$. Since the marginal random variable Y is a Bernoulli random variable, this means there's a function $p : \mathcal{X} \to [0, 1]$ such that $\mathbb{P}(Y = +1|X = x) = p(x)$, namely, choosing the label for x is a coin flip with probability $p(x)$. If $p(x) = 0$ or $p(x) = 1$ for some x, the label is deterministic and we are back to the "label function" f. But for other values of $p(x)$, whereas before the label depended deterministically on x, now it is random. This models **measurement noise** - the fact that there may be noise in the labels, and that the distribution of the noise may change from x to x. (This is the perspective of the Logistic Regression classifier you saw in the classification lecture, for example. There, we didn't pay attention a distribution on x - just assumed the samples are given - and tried to estimate the conditional probability $\mathbb{P}(Y = +1|X = x) = p(x)$.)
- $\mathbb{P}(X = x, Y = y) = \mathbb{P}(Y = y)\,\mathbb{P}(X = x|Y = y)$. For this perspective, we first draw the label according to a "coin flip" - a Bernoulli random variable. Then, each class has its own distribution for the samples x. We draw the label x from $\mathbb{P}(X = x|Y = +1)$ or from $\mathbb{P}(X = x|Y = -1)$, according to the label y chosen. (This is the perspective of the LDA classifier you saw in homework, for example.)

Exercise. Let $\tilde{\mathcal{D}}$ be a distribution on X alone, and let $f : \mathcal{X} \to \mathcal{Y}$ be a labeling function. Construct an equivalent joint distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$ such that the random variable $(X, f(X))$,

where $X \sim \tilde{\mathcal{D}}$, has the same distribution over $\mathcal{X} \times \mathcal{Y}$ as $(X, Y) \sim \mathcal{D}$.

How shall we define the loss, when $\mathcal{D}$ is a distribution over $\mathcal{X} \times \mathcal{Y}$? It's simple. Staying (for now) with the misclassification loss, we just define, for a hypothesis $h : \mathcal{X} \to \mathcal{Y}$,

$$L_{\mathcal{D}}(h) := \mathbb{P}_{(x,y)\sim\mathcal{D}} \{h(x) \neq y\} \equiv \mathcal{D} \{(x, y) \,|\, h(x) \neq y\} \tag{2}$$

Notice we no longer have a "ground truth" labeling function f. The closest thing we have to f is the conditional probability $\mathbb{P}(Y = y|X = x)$.

21.2 Removing the realizability assumption

Recall that in the deterministic labeling case, under the realizability assumption, we could reach zero generalization error:

$$\min_{h\in\mathcal{H}} L_{\mathcal{D},f}(h) = L_{\mathcal{D},f}(f) = 0\,.$$

However, in the random labeling case, there is no "ground truth" labeling function $f : \mathcal{X} \to \mathcal{Y}$, so that the realizability assumption no longer makes sense. Due to the measurement noise, we may no longer be able to reach 0 generalization loss. This means we have to change the definition of **accuracy**: we expect the learning algorithm to output a rule which has generalization loss **at most ϵ larger than the minimal possible loss** $\min_{h\in\mathcal{H}} L_{\mathcal{D}}(h)$. But wait, does a minimum exist? What can we say about this minimum?

Definition: Bayes optimal predictor In the recitation you saw the following definition. For a given distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$ define the *Bayes optimal predictor* for $\mathcal{D}$ by

$$f_{\mathcal{D}}(x) = \begin{cases} 1 & \mathbb{P}(y = 1|x) \geqslant 1/2 \\ 0 & otherwise \end{cases}$$

Note that $f_{\mathcal{D}} : \mathcal{X} \to \mathcal{Y}$ is a rule (an hypothesis). However it depends on $\mathcal{D}$, which, according to the rules of the game, we don't know. So $f_{\mathcal{D}}$ is what is known as an **Oracle Quantity** - if we had an oracle telling us $\mathcal{D}$, then we could classify with $f_{\mathcal{D}}$. Oracle quantities, like this one, are used to compare the loss of any other rule to the loss of the **best possible** rule.

In homework you solved the following **Exercise.** The Bayes optimal predictor has the best possible generalization error: $\forall g : \mathcal{X} \to \mathcal{Y}\ ,\ L_{\mathcal{D}}(f_{\mathcal{D}}) \leqslant L_{\mathcal{D}}(g)$.

It follows that for any hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$,

$$L_{\mathcal{D}}(f_{\mathcal{D}}) = \min_{h\in\mathcal{Y}^{\mathcal{X}}} L_{\mathcal{D}}(h) \leqslant \min_{h\in\mathcal{H}} L_{\mathcal{D}}(h)$$

and so $\{L_{\mathcal{D}}(h) \,|\, h \in \mathcal{H}\}$ is bounded from below and, glossing over the difference between minimum and infimum, we can write $\min_{h\in\mathcal{H}} L_{\mathcal{D}}(h)$.

Definition: Let $\epsilon > 0$. We say that a rule $h \in \mathcal{H}$ is Approximately correct with accuracy ϵ with respect to the distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$ if

$$L_{\mathcal{D}}(h) \leqslant \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon \,,$$

namely, if $L_{\mathcal{D}}(h)$ is at most ϵ away from the best possible loss achievable by **any** hypothesis in $\mathcal{H}$. We note again that, in our previous framework, under the realizability assumption, the minimal loss is simply 0 since we **assumed** the existence of some $h' \in \mathcal{H}$ with $L_{\mathcal{D}}(h') = 0$.

So, we got rid of the realizability assumption: we no longer assume that Nature plays a labeling function in the chosen hypothesis class $\mathcal{H}$. In fact, we no longer have a labeling function at all! Nature's strategy just consists of the joint distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$.

21.3 Introducing a general loss function

So we achieved two of the three improvements we wanted: we allow measurement noise in the labels, and we got rid of the realizability assumption. Our last improvement is to allow a general loss function.

Definition: general loss function. A **loss function** is a function $\ell : \mathcal{H} \times Z \to [0, \infty)$, where $Z = \mathcal{X} \times \mathcal{Y}$. For $z = (x, y)$, we simplify the notation and, instead of the cumbersome notation $\ell(h, z)$, we simply write $\ell(h(x), y)$.

We already know well the most common example for classification loss - the misclassification loss, also known as the **0-1 loss**.

$$\ell_{0,1}(h, (x, y)) := \begin{cases} 1 & h(x) \neq y \\ 0 & h(x) = y \end{cases} .$$

Indeed the loss we work with $L_{\mathcal{D}}(h)$ from Equation (2) above can we written as

$$L_{\mathcal{D}}(h) \equiv \mathbb{E}_{\mathcal{D}}[\ell_{0-1}(h, (x, y))]$$

where here and below $\mathbb{E}_{\mathcal{D}}[\cdot]$ denotes the expected value according to $(x, y) \sim \mathcal{D}$.

From here on we will assume a general loss function ℓ, but will often have in mind $\ell_{0,1}$.

Definition: generalization loss induced by a general loss function. For a distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$ and a loss $\ell : \mathcal{H} \times Z \to [0, \infty)$ (where again $Z = \mathcal{X} \times \mathcal{Y}$) we extend Definition **??** and define the loss of a rule (hypothesis) $h : \mathcal{X} \to \mathcal{Y}$ with respect to $\mathcal{D}$ and ℓ as

$$L_{\mathcal{D}}(h) := \mathbb{E}_{z \sim \mathcal{D}}[\ell(h, z)]$$

where $Z = \mathcal{X} \times \mathcal{Y}$ and $z = (x, y) \in Z$.

How shall we define **accuracy** with respect to a general loss induced by a general loss function? The definition above (for $0 - 1$ loss) generalizes naturally and easily:

Definition: Let $\epsilon > 0$. We say that a rule $h \in \mathcal{H}$ is Approximately correct with accuracy ϵ with respect to the distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$, the loss function $\ell : \mathcal{H} \times (\mathcal{X} \times \mathcal{Y}) \to [0, \infty)$ and the hypothesis class $\mathcal{H}$ if

$$L_{\mathcal{D}}(h) \leqslant \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon \,.$$

Note that now accuracy must be defined with respect to an hypothesis class $\mathcal{H}$.

22 Agnostic-PAC learnability

Our new framework is called Agnostic-PAC. We are now ready to define Agnostic-PAC Learnability of a hypothesis class.

22.1 Probably Approximately correct learner - in the new framework.

Exercise. Define - rigorously - what it means for $h : \mathcal{X} \to \mathcal{Y}$ to be an Agnostic Probably Approximately correct learner with accuracy ϵ and confidence δ, with respect to a loss function ℓ, hypothesis class $\mathcal{H}$ and a distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$.

Exercise. Let X be a sample space, $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ an hypothesis class, and let ℓ_{0-1} be the $0-1$ loss function. Let $0 < \epsilon, \delta < 1$. Assume that $h \in \mathcal{H}$ is an Agnostic Probably Approximately correct learner (with accuracy ϵ and confidence δ), with respect to ℓ_{0-1}, $\mathcal{H}$. Show that it follows that h is a Probably Approximately Correct learner (with accuracy ϵ and confidence δ)

22.2 Agnostic-PAC learnability

Definition 21 *A hypothesis class $\mathcal{H}$ is Agnostic-PAC learnable with respect to loss $\ell : \mathcal{H} \times (\mathcal{X} \times \mathcal{Y}) \to [0, \infty)$ if there exists a function $\tilde{m}_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ and a a learning algorithm $\mathcal{A} : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$ with the following property: For every $(\epsilon, \delta) \in (0,1)$ for every distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$, for any $m \geqslant \tilde{m}_{\mathcal{H}}(\varepsilon, \delta)$*

$$\mathcal{D}^m\{S_m \,|\, L_{\mathcal{D}}(h_S) \leqslant \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h) + \varepsilon\} \geqslant 1 - \delta$$

where $S_m = (x_1, y_1), \ldots (x_m, y_m)$ is sampled i.i.d according to $\mathcal{D}$, and $h_S = \mathcal{A}(S)$.

Note that, as before, we abused notation and wrote $\mathcal{A}$ for the entire sequence of learners $A_m : (\mathcal{X} \times \mathcal{Y})^m \to \mathcal{H}$, one for each possible sample size m.

Exercise. To make sure you understand the definition, prove the following rigorously: if an hypothesis class $\mathcal{H}$ is Agnostic-PAC learnable with respect to ℓ_{0-1}, then it is PAC-learnable.

The Learning Game. To help us understand the definition of Agnostic-PAC learnability, let us write the "learning game" in the Agnostic PAC framework.

Fix desired accuracy $\epsilon > 0$ and confidence $\delta > 0$. Fix an hypothesis class $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ and a loss function ℓ. We play a game against Nature, with random payoff.

- We choose a sample size m and a learner $\mathcal{A} : (\mathcal{X}, \mathcal{Y})^m \to \mathcal{H}$. Both m and $\mathcal{A}$ can depend on (ϵ, δ).
- Nature knows our strategy, and, after us, chooses strategy that consists of a probability distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. (Nature's strategy can depend on $\epsilon, \delta, \mathcal{H}$ specified, and also on the $m, \mathcal{A}$ we chose.)
- A sample $S \in (\mathcal{X} \times \mathcal{Y})^m$ of size m is drawn according to $\mathcal{D}$.
- The sample S is fed into $\mathcal{A}$ to produce a prediction rule $h_S = \mathcal{A}(S)$. Note that $h_S \in \mathcal{H}$.
- The payoff is $L_{\mathcal{D}}(h_S) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\ell(h, (x, y))$. It is random since S is random and therefore h_S is random.
- We are going to assume Nature is "cruel" and does her best to win. So we'll look for learners $\mathcal{A}$ that have a **guaranteed maximal loss $L_{\mathcal{D}}(h)$ for any** strategy $\mathcal{D}, f$ that Nature might play.
- To determine if we were successful in the game, we play the game many many times (both us and Nature play the same strategies, just the training samples drawn are different). We count and calculate the probability, over the random draws of training samples S, of the event $\{S \sim \mathcal{D}^m \,\big|\, L_{\mathcal{D}}(h_S) \leqslant \min_{h'\in\mathcal{H}} L_{\mathcal{D}}(h') + \epsilon\}$. (To do this we assume knowledge of the "Oracle quantity" $\min_{h'\in\mathcal{H}} L_{\mathcal{D}}(h')$ - the best possible loss of any rule in $\mathcal{H}$.) If this probability is found to be larger than $1-\delta$, that is, if the learner $\mathcal{A}$ we chose was Probably Approximately correct with accuracy ϵ and confidence δ - against Nature's best strategy - **we say that we've been successful (with regards to the parameters ϵ, δ) and hypothesis class $\mathcal{H}$.**

22.3 PAC learnability is equivalent to Agnostic-PAC learnability

While we will not prove this, it turns out that moving to the more general framework of Agnostic-PAC didn't change anything:

Theorem. Let X be a sample space and $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ an hypothesis class. Then $\mathcal{H}$ is PAC-Learnable if and only if it is Agnostic-PAC learnable.

23 Back to the Fundamental Theorem of Statistical Learning

23.1 Empirical Risk Minimization strikes again

Recall that in our previous framework, in the realizable case, the Empirical Risk Minimization (ERM) learner was defined as any $h \in \mathcal{H}$ consistent with the training set $S = (x_1, y_1), \ldots, (x_m, y_m)$. In our more general case we redefine ERM as **any** minimizer of the empirical risk.

Definition: Empirical Risk with respect to a general loss function. Let $h : \mathcal{X} \to \mathcal{Y}$ be a rule (hypothesis). We define the empirical risk of h with respect to the loss function ℓ and sample $S = (x_1, y_1), \ldots, (x_m, y_m)$ by

$$L_S(h) := \frac{1}{m}\sum_{i=1}^{m} \ell(h, z_i) \tag{3}$$

where $z_i = (x_i, y_i)$.

Definition: ERM learner in the Agnostic-PAC framework. The ERM learning algorithm, in our upgraded framework, is define as

$$\mathcal{A}_{ERM} : S \mapsto \text{argmin}_{h \in \mathcal{H}} L_S(h).$$

Note - as before - that the ERM rule may not be unique. There may be many hypotheses in $\mathcal{H}$ that achieve the minimum $\min_{h \in \mathcal{H}} L_S(h)$.

23.2 ERM makes sense due to WLLN

Now, recall the **Weak Law of Large Numbers** (WLLN):

- If X_i are i.i.d random variables and $\mu = \mathbb{E}(X_i)$, then

$$\lim_{m \to \infty} \frac{1}{m} \sum_{i=1}^{m} X_i = \mu$$

  where the convergence is **in probability**

- Namely, for any $\delta > 0$

$$\lim_{m \to \infty} \mathbb{P}\left\{ \left| \frac{1}{m} \sum_{i=1}^{m} X_i - \mu \right| > \delta \right\} = 0$$

- Namely, for any $\delta > 0$ there is $m_0 \in \mathbb{N}$ such that for $m > m_0$,

$$\mathbb{P}\left\{ \left| \frac{1}{m} \sum_{i=1}^{m} X_i - \mu \right| > \delta \right\} < \epsilon \,.$$

Observe now that for any h we have $\mathbb{E}_{\mathcal{D}} L_S(h) = L_{\mathcal{D}}(h)$.

- By WLLN, when S is i.i.d sample of size m, $\lim_{m \to \infty} L_S(h) = L_{\mathcal{D}}(h)$, in probability
- Therefore for any $\delta > 0$ there is $m_0 \in \mathbb{N}$ such that for $m > m_0$,

$$\mathbb{P}\left\{ \left| L_S(h) - L_{\mathcal{D}}(h) \right| > \delta \right\} < \epsilon \,.$$

- But does this mean that $\text{argmin}_{h \in \mathcal{H}} L_S(h)$ is close to $\text{argmin}_{h \in \mathcal{H}} L_D(h)$??

23.3 The Fundamental Theorem - now with Agnostic-PAC

Let us reformulate the Fundamental Theorem using Agnostic-PAC learbnability, and the generalized notion of ERM we just defined.

Theorem 8 (The Fundamental Theorem of Statistical Learning - for Agnostic PAC) *Let $\mathcal{H}$ be a hypothesis class of binary classifiers with VC-dimension $d \leqslant \infty$. Then, $\mathcal{H}$ is* **Agnostic-PAC learnable** *if and only if $d < \infty$. In this case:*

1. *There there are absolute constants C_1, C_2 such that the sample complexity of $\mathcal{H}$ satisfies*

$$C_1 \frac{d + \log(1/\delta)}{\epsilon^2} \leqslant m_{\mathcal{H}}(\epsilon, \delta) \leqslant C_2 \frac{d + \log(1/\delta)}{\epsilon^2}$$

2. *Furthermore, the upper bound on sample complexity is achieved by the ERM learner.*

(Note that the price we pay for Agnostic PAC learning is that the sample complexity is proportional to $1/\epsilon^2$, not to $1/\epsilon$ as in the PAC Fundamental theorem)

24 A Taste of the Proof

In the last part of this lecture, and to conclude this two-lecture series on the PAC Theory of learning, let us go a little into the proof of this spectacular theorem. This is an opportunity for you to feel a "heavy-weight" argument in machine learning theory, a complicated argument which consists of several stages. This is just a taste - you are encouraged to look at the full proof in the "Understanding Machine Learning" book. We also won't try to understand the quantitative part of the theorem - the actual bounds on Sample Complexity of $\mathcal{H}$ (the minimal number of samples required to create an Agnostic-PAC learner for $\mathcal{H}$).

What we will do is gain intuition as to how is it that the VC-dimension characterizes learnability, namely, why is that $\mathcal{H}$ **is Agnostic-PAC learnable if and only if** $VCdim(\mathcal{H}) < \infty$**.**

Part One of the Fundamental Theorem: Learning $\mathcal{H}$ with infinite VC-dimension is impossible: One part of the "if and only if" is about impossibility of learning: If $VCdim(\mathcal{H}) = \infty$, it's impossible to create an Agnostic-PAC learner $\mathcal{A}$ for $\mathcal{H}$ (with accuracy ϵ and confidence δ) - by **any** learning algorithm - and using **any** number of training samples.

We already gained intuition for this direction in previous lecture, where we looked at a "proof" of the No Free Lunch theorem, and introduced the VC-dimension. We saw an informal argument that, if there exists $C \subset \mathcal{X}$ that is **shuttered** by $\mathcal{H}$, then no learning algorithm can be a Probably Approximately correct learner if it's based on less than $|C|/2$ samples. Now, the statement $VCdim(\mathcal{H}) = \infty$ just means that there are subsets of $\mathcal{X}$ of arbitrary size that are shuttered by $\mathcal{H}$ - and therefore, no finite sample size will do.

Let us now turn to the Part Two of the Theorem.

Part Two of the Fundamental Theorem: The ERM learner is a universal learner for any $\mathcal{H}$ with finite VC-dimension.
The second part of the Fundamental Theorem states that if $\mathcal{H}$ is a hypothesis class with $VCdim(\mathcal{H}) = d < \infty$ then $\mathcal{H}$ is Agnostic-PAC learnable as defined in Definition 21, using any ERM learner. Let's see how this is proved.

24.1 The uniform convergence property

An ERM learner chooses a rule $ERM_{\mathcal{H}}(S)$ which minimizes $L_S(h)$ for the sample S at hand; We hope that the rule $h_S \in ERM_{\mathcal{H}}(S)$, which has minimal empirical risk, will generalize well. Formally, we have to prove that

$$\mathcal{D}^m \left\{ S \in (\mathcal{X} \times \mathcal{Y})^m \,\middle|\, |L_{\mathcal{D}}(h_S) - L_S(h_S)| < \epsilon \right\} > 1 - \delta$$

Obviously, this can only happen if S is a "special" sample - one for which for any $h \in \mathcal{H}$ **the empirical risk $L_S(h)$ is pretty close to the generalization loss $L_{\mathcal{D}}(h)$.**

Why is that hard to prove? Note that for any $h \in \mathcal{H}$ we have $\mathbb{E}[L_S(h)] = L_{\mathcal{D}}(h)$, so, as we have seen above, by the weak law of large numbers, $L_S(h)$ converges to $L_{\mathcal{D}}(h)$ in probability as the sample size $m \to \infty$. This means that for each $\mathcal{D}$, for each h, and for each ϵ, δ there is m_0 such that for $m > m_0$ we have $\mathbb{P}\{|L_S(h) - L_{\mathcal{D}}(h)| < \epsilon\} > 1 - \delta$. However **this m_0 depends on both $\mathcal{D}$ and h.** We want m_0 that is **uniform in the distributions $\mathcal{D}$ and the hypotheses $h \in \mathcal{H}$.**

Recall the definition of **uniform convergence** of function sequence: a sequence of functions $f_n : X \to \mathbb{R}$ converges uniformly to $f : X \to \mathbb{R}$ if $\forall \epsilon > 0 \, \exists m_0 \, s.t. \forall x \in X \, |f_n(x) - f(x)| < \epsilon$.

Indeed, what we want is to ensure that $L_S(h)$ converges to $L_D(h)$ **uniformly in $\mathcal{D}$ and in $h \in \mathcal{H}$.**

This leads to the following definition.

Definition. A training sample S is called ϵ-representative for $\mathcal{D}, \mathcal{H}, \ell$ if

$$\forall h \in \mathcal{H} \ \left| L_S(h) - L_{\mathcal{D}}(h) \right| < \epsilon \, .$$

This condition will ensure that minimizing $L_S(h)$ over $h \in \mathcal{H}$ (which is what ERM does) will be close to minimizing $L_D(h)$ over $h \in \mathcal{H}$ (which is what we would like to achieve, Approximately.)
Specifically, we see immediately that if we have an ϵ-representative training set S at hand, then $ERM_{\mathcal{H}}(S)$ will "almost" achieve $\min_{h \in \mathcal{H}} L_{\mathcal{D}}(h)$. Formally:

Lemma 2 *Let S be an $\epsilon/2$-representative sample for $\mathcal{D}, \mathcal{H}, \ell$. Let h_S be any output of $ERM_{\mathcal{H}}(S)$, namely, $h_S \in argmin_{h \in \mathcal{H}} L_S(h)$. Then*

$$L_{\mathcal{D}}(h_S) \leqslant \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \epsilon \, .$$

Exercise: Prove Lemma 2 yourself - it's one line.

Definition. We say that an hypothesis class $\mathcal{H}$ has the **uniform convergence property** if there exists $m_{\mathcal{H}}^{UC} : (0,1)^2 \to \mathbb{N}$ such that for every $0 < \epsilon, \delta < 1$ and every distribution $\mathcal{D}$ on $\mathcal{X} \times \mathcal{Y}$,

$$\mathcal{D}^m \left(\{ S \in (\mathcal{X} \times \mathcal{Y})^m \mid S \text{ is } \epsilon\text{-representative} \} \right) \geqslant 1 - \delta \, .$$

Exercise. Prove that if $\mathcal{H}$ has the uniform convergence property with function $m_{\mathcal{H}}^{UC} : (0,1)^2 \to \mathbb{N}$ then $\mathcal{H}$ is Agnostic-PAC learnable with sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leqslant m^{UC}(\epsilon/2, \delta)$.

25 Proving that if $VCdim(\mathcal{H}) < \infty$ then $\mathcal{H}$ has the uniform convergence property

So to show Part Two of the fundamental theorem (that ERM is a universal learner), it's enough to show that if $VCdim(\mathcal{H}) < \infty$ then $\mathcal{H}$ has the uniform convergence property. This means showing that for large enough m – that does not depend on $\mathcal{D}$ – an i.i.d sample is ϵ-representative with probability at least $1 - \delta$, and that this hold for any possible $\mathcal{D}$.

25.1 Achieving uniformity in both $\mathcal{H}$ and $\mathcal{D}$

But how are we going to achieve uniformity across both $\mathcal{D}$ and $h \in \mathcal{H}$? Here is what we are going to do: we are going to define a function $F_m^{\mathcal{D}} : (\mathcal{X} \times \mathcal{Y})^m \to \mathbb{R}$ by

$$F_m^{\mathcal{D}}(S) := \sup_{h \in \mathcal{H}} \Big| L_{\mathcal{D}}(h) - L_S(h) \Big|. \tag{4}$$

$F_m^{\mathcal{D}}$ maps a training sample of size m, to a real number measuring its "worse possible confusion" - the maximal difference, over $\mathcal{H}$, between an empirical risk of a hypothesis h and the generalization error of that h. Observe that $F_m^{\mathcal{D}}$ is a function of the random sample S, so it is a random variable, whose distribution depends on the distributions $\mathcal{D}^m$ of training sets of length m.

In essence, we would like to show that with high probability, $F_m^{\mathcal{D}}$ is small. Formally, if we're able to show that for every $0 < \epsilon, \delta < 1$ there exists $m_{\mathcal{H}}^{UC}(\epsilon, \delta) \in \mathbf{N}$ such that for every distribution $\mathcal{D}$

$$\mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \epsilon \right\} < \delta$$

then we are done.

25.2 The case of finite $\mathcal{H}$

To understand the key argument, let us first consider the much much easier case of finite $\mathcal{H}$ and see how Agnostic-PAC learnability is proved in this case.

Claim: Fix ϵ, δ. There exists m_0 such that for all $m > m_0$, the following holds: for any $\mathcal{D}$,

$$\mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \epsilon \right\} =< \delta \,.$$

Proof: Now by definition

$$\mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \epsilon \right\} = \mathcal{D}^m \left\{ S \,\middle|\, \exists h \in \mathcal{H} \,,\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\}$$

By union bound

$$\mathcal{D}^m \left\{ S \,\middle|\, \exists h \in \mathcal{H} \,,\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\} \leqslant \sum_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\} .$$

To achieve uniformity over $\mathcal{H}$ we simply use

$$\sum_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\} \leqslant |\mathcal{H}| \cdot \max_{h \in \mathcal{H}} \mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\} .$$

We thus need to bound $\mathcal{D}^m \left\{ S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon \right\}$ uniformly in $\mathcal{D}$ and h. By the weak law of large numbers (WLLN), since $L_S(h)$ is an empirical mean of i.i.d random variables with expected value $L_{\mathcal{D}}(h)$, we know that for any $\epsilon > 0$, as $m \to \infty$, $\mathcal{D}^m \{ \big| L_S(h) - L_{\mathcal{D}}(h) \big| > \epsilon \} \to 0$. But we need more than that - we want a bound on this probability that does not depend on $\mathcal{D}, h$. This we don't get from WLLN. What we need is a known as a **concentration of measure** inequality: something that bounds the distance between the empirical mean and the expected value.

Indeed, recall the famous **Hoeffding Inequality**: Let $\theta_1, \dots \theta_m$ be a sequence of i.i.d random variables and assume that for all i, we have both $\mathbb{E}[\theta_i] = \mu$ and $\mathbb{P}\{a \leqslant \theta_i \leqslant b\} = 1$. Then, for any $\epsilon > 0$,

$$\mathbb{P}\left\{\left|\frac{1}{m}\sum_{i=1}^{m}\theta_i - \mu\right| > \epsilon\right\} \leqslant 2e^{-2\frac{m\epsilon^2}{(b-a)^2}}.$$

To use Hoeffding, we define $\theta_i := \ell(h, (x_i, y_i))$. Observe that $L_{\mathcal{D}}(h) = \mathbb{E}[\theta_i]$ (where the expectation is with respect to $\mathcal{D}$) and that $L_S(h) = \frac{1}{m}\sum_{i=1}^{m}\theta_i$. This is exactly what we wanted - to bound the difference between $L_D(h) - L_S(h)$, **uniformly** in h and $\mathcal{D}$!

So, using the Hoeffding Inequality, we get $\mathcal{D}^m\left\{S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon\right\} \leqslant 2exp(-2m\epsilon^2)$. So we have shown

$$|\mathcal{H}| \cdot \max_{h\in\mathcal{H}} \mathcal{D}^m\left\{S \,\middle|\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon\right\} \leqslant 2|\mathcal{H}|exp(-2m\epsilon^2)\,.$$

From our union bound, we get

$$\mathcal{D}^m\left\{S \,\middle|\, \exists h \in \mathcal{H}\,,\, |L_S(h) - L_{\mathcal{D}}(h)| > \epsilon\right\} \leqslant 2|\mathcal{H}|exp(-2m\epsilon^2)$$

So just take $m \geqslant \frac{\log(2|\mathcal{H}|/\delta)}{2\epsilon^2}$, and we get $\mathcal{D}^m\left\{F_m^{\mathcal{D}}(S) > \epsilon\right\} < \delta$ as required. ■

25.3 The general case - infinite $\mathcal{H}$

Unfortunately, we can't use this argument in the infinite $\mathcal{H}$ case - we can't just do a union bound over an infinite number of hypothesis. So what are we going to do in the general case?

Recall that, for every finite $C \subset \mathcal{X}$, we write $\mathcal{H}_C$ for the hypotheses in $\mathcal{H}$, all restricted to C. It turns out the key to the proof is to understand **how fast can the restriction $\mathcal{H}_C$ grow with $|C|$**.

What do mean by that? If $|C| \leqslant VCdim(\mathcal{H})$ it could be that $\mathcal{H}$ shutters $|C|$. So it could be that $|\mathcal{H}_C| = 2^{|C|}$. But if $|C| > VCdim(\mathcal{H})$ it can't be - by definition - that $|\mathcal{H}_C| = 2^{|C|}$. So how large can $|\mathcal{H}_C|$ be (at most)?

Definition. For an hypothesis class $\mathcal{H}$ Define $\tau\mathcal{H}(m)$ by

$$\tau_{\mathcal{H}}(m) := \max\left\{|\mathcal{H}_C| \,\middle|\, C \subset \mathcal{X}\,,\, |C| = m\right\}.$$

This is a purely combinatorial property of $\mathcal{H}$: the maximal number of functions that can be obtained by restricting $\mathcal{H}$ to any subset of size m. The larger and more complicated $\mathcal{H}$, the larger we can expect $\tau_{\mathcal{H}}(m)$ to be. In other words, $\tau_{\mathcal{H}}(m)$ measures how fast - at most - $\mathcal{H}_C$ can grow with $|C|$.

For example, we saw that if $VCdim(\mathcal{H}) = \infty$, then $\tau_{\mathcal{H}}(m) = 2^m$, namely, $\mathcal{H}_C$ can grow exponentially in $|C|$.

Definition. Let $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$. Suppose that there exist $m_0 \in \mathbb{N}$, $b > 0$ and $\beta > 0$ such that for all $m > m_0$,

$$\tau_{\mathcal{H}}(m) \leqslant b \cdot m^{\beta} .$$

Then we'll say that $\mathcal{H}_C$ grows **polynomially** in $|C|$.

The proof is really wonderful, and is based on two parts:

1. If $|\mathcal{H}_C|$ grows polynomially in $|C|$, then $\mathcal{H}$ has the uniform convergence property - and hence is Agnostic-PAC learnable using the ERM rule.
2. If $VCdim(\mathcal{H}) < \infty$, then $|\mathcal{H}_C|$ grows polynomially in $|C|$.

25.3.1 First part of the proof: if $|\mathcal{H}_C|$ grows polynomially in $|C|$ then $\mathcal{H}$ has the uniform convergence property

Recall that we would like to show

$$\mathcal{D}^m \left\{ F_m^{\mathcal{D}}(S) > \epsilon \right\} < \delta ,$$

uniformly in $\mathcal{D}$. Since $F_m^{\mathcal{D}}$ is a non-negative random variable, there is a useful inequality that does just that. Recall Markov's inequality: If X is a non-negative random variable then

$$\mathbb{P}\{X > \alpha\} \leqslant \mathbb{E}[X]/\alpha$$

So, we are going to find a sequence of numbers α_m, such that

$$\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)] \leqslant \alpha_m . \tag{5}$$

The magic here is that the sequence α_m will depend on $\mathcal{H}$ but **not** on the distribution $\mathcal{D}$. In other words, we will bound $\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)]$ uniformly across $\mathcal{D}$.

If we succeed in doing this, then we're done. Why? In our case, $F_m^{\mathcal{D}}(S)$ is a non-negative random variable. If we find a sequence α_m for which Equation (5) holds, then

$$\mathbb{P}_{\mathcal{D}^m}\left\{ \sup_{h \in \mathcal{H}} \left| L_{\mathcal{D}}(h) - L_S(h) \right| > \epsilon \right\} = \mathbb{P}_{\mathcal{D}^m}\left\{ F_m^{\mathcal{D}}(S) > \epsilon \right\} \leqslant \frac{\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)]}{\epsilon} \leqslant \frac{\alpha_m}{\epsilon} .$$

In other words, with probability at least $1 - \alpha_m/\epsilon$, a training set of length m is ϵ-representative! We managed to achieve uniformity across both $h \in \mathcal{H}$ (by using $F_m^{\mathcal{D}}$, a supremum over h) and over $\mathcal{D}$ (by bounding the expected value of $F_m^{\mathcal{D}}$ independently of $\mathcal{D}$.

Now for the punch line: if we're able to find such as sequence α_m that decreases to 0, then for any ϵ, δ we can set m_0 to be such that for all $m > m_0$, $\alpha_m/\epsilon < \delta$. This would imply that $\mathcal{H}$ has the uniform convergence property.

So let's find such a sequence α_m.

Most of the heavy lifting that remains is in the following lemma, which we will not prove:

Lemma 3 *Let $F_m^{\mathcal{D}}$ be as in Equation (4). Then*

$$\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)] \leqslant O\left(\frac{\sqrt{\log(\tau_{\mathcal{H}}(2m))}}{\sqrt{2m}} \right) + o(m)$$

independently of $\mathcal{D}$.

Now, since we assumed that $|\mathcal{H}_C|$ grows polynomially in $|C|$, we have for all $m > m_0$ (for some m_0) that $\tau_{\mathcal{H}}(m) \leqslant b \cdot m^{\beta}$ for some $b, \beta > 0$. Hence,

$$\mathbb{E}_{\mathcal{D}^m}[F_m^{\mathcal{D}}(S)] \leqslant O\left(\frac{\sqrt{\beta \cdot \log(2m)}}{\sqrt{2m}}\right) + o(m) \searrow 0$$

So we see that, indeed, if $|\mathcal{H}_C|$ grows polynomially in $|C|$ then $\mathcal{H}$ has the uniform convergence property.

25.3.2 If $VCdim(\mathcal{H}) < \infty$, $|\mathcal{H}_C|$ only grows polynomially in $|C|$

Finally, here the VC-dimension appears on stage. By definition, if $m \leqslant VCdim(\mathcal{H})$ then there exists a set $C \subset \mathcal{X}$, of size m, which is shuttered by $\mathcal{H}$. This means that if $m \leqslant VCdim(\mathcal{H})$ then $\tau_{\mathcal{H}}(m) = 2^m$.

The next lemma, which you will prove in homework, is surprising. It says that while $\tau_{\mathcal{H}}(m)$ grows exponentially in m for $m \leqslant VCdim(\mathcal{H})$, it only grows **polynomially** in m for $m > VCdim(\mathcal{H})$:

Lemma 4 *If $m > VCdim(\mathcal{H})$ then*

$$\tau_{\mathcal{H}}(m) \leqslant \left(\frac{em}{d}\right)^d.$$

25.3.3 Summary

So, that was a taste of the proof of the second part of the fundamental theorem. We proved everything formally, except Lemma 2. This lemma is indeed deep and meaningful: it bounds the expected value of the "worse possible deviation" between empirical risk and generalization error, $\sup_{h \in \mathcal{H}} \left|L_{\mathcal{D}}(h) - L_S(h)\right|$, over a random choice of training sample, uniformly in $\mathcal{D}$. The bound uses $\tau_{\mathcal{H}}(m)$, which bounds how fast the size of a restriction $\mathcal{H}_C$ can grow with $|C|$.

Lecture 6: Ensemble Methods - Bagging and Boosting

Suggested Reading:

- Bootstrap: ESL 7.11
- Bagging: ESL 8.7
- Random forests: ESL 15
- Boosting: UML 10
- Adaboost: UML 10.2, ESL 10.1

More advanced reading (beyond the material covered):

- Statistical learning view of boosting: ESL 10.2, 10.4, 10.5
- Gradient boosting: ESL 10.10
- Interpreting Boosted Trees and Random Forests: ESL 10.13

26 Introduction

After two lectures on theory of machine learning, we are back to new machine learning algorithms - and specifically, for supervised batch learning.

In the classification lecture we saw several learning algorithms for classification. We saw that they are very different from each other - each of them implements a different **principle** for choosing the learned rule $h_S \in \mathcal{H}$ based on the training sample S, and each uses a different algorithm to implement the chosen principle computationally - first as a formally stated algorithm, and then actual implementation in software.

The classifiers we saw (eg. SVM, trees) are not "cutting edge" - none of them would be considered the best choice for classification today. But with the methods of this lecture we finally get to create state-of-the-art classifiers. Some of the best-known classification methods, which often that win competitions, are based on the methods we'll see this in this lecture[19].

This lecture will be different from our classification lecture. We won't be seeing any new learning algorithms. Instead, we will see **"meta-algorithms"** - general methods that can be applied to any **existing** learning algorithm and improve its performance. The improvement in performance can be quite radical.

We will address classification problems for simplicity, but everything you see here generalizes (sometimes trivially) to regression problems as well.

[19] For example, a list someone compiled of competitions won using gradient boosting: `https://github.com/Microsoft/LightGBM/blob/master/examples/README.md#machine-learning-challenge-winning-solutions`

We will learn about the three B's: **Bootstrap**, **Bagging** and **Boosting**. You should package these ideas together (call them B^3) and take them with you wherever you go.

- **Bootstrap** is one of the most useful, important and influential ideas in statistics since computers stared being used to analyze data. It is a truly magical idea that has many, many uses and applications.
- One of the uses of Bootstrap is for improving prediction of a supervised learning algorithm. (We'll see another use in future lectures.) **Bagging** is a nickname for Bootstrap as it is used to improve prediction of a supervised learning algorithm.
- **Boosting** is another truly magical idea, which is completely different. And is one of the most useful, important and influential ideas in machine learning.

So, we're learning powerful and broadly applicable ideas today. You'll be able to use these ideas when you encounter difficult problems, even outside the context of the meta-algorithms we will see today.

26.1 Bias/Variance

The meta-algorithms we will see in this lecture succeed because they change the bias and the variance of the learning algorithms on top of which they are applied. So, before we actually start talking about them, let us recall the **bias-variance tradeoff** (also known as the bias-complexity tradeoff).

Several times in the course so far, we stated - informally - that the "larger" or "more complicated" our chosen hypothesis class, typically our learner will have lower **bias** and higher **variance.**

We said informally that **bias** is part of the generalization error that is incurred by the "best" hypothesis in $\mathcal{H}$. If we think of an unknown labeling function f chosen by nature, then bias measures how well the unknown labeling function f can be decried by the "closest" hypothesis in $\mathcal{H}$. Obviously, the larger $\mathcal{H}$, the more expressive power it has to describe more complicated functions f - hence a lower bias.

We said informally that **variance** is the part of the generalization error that is incurred by the fact that the training sample is random, hence our chosen rule h_S is also random. The larger $\mathcal{H}$ will be, the more freedom our learning algorithm has to "chase" random fluctuations in the training sample, which do not represent the underlying labeling we are trying to learn. (Variance can be further broken down into two parts - one part comes from randomness in the choice of training samples, and another part comes from the measurement noise or noise in the labels. Let's keep in simple and not go into that now.)

We mentioned the bias-variance **tradeoff** - the more complicated the model, the smaller the bias and the larger the variance. Informally, the generalization error is the sum (or somehow the combination) of these two. So when we can tune the model complexity (another name for the size / complexity of our hypothesis class) we'll look for the "sweet spot" of a model that not has just the right amount of complexity, not too much and not too little.

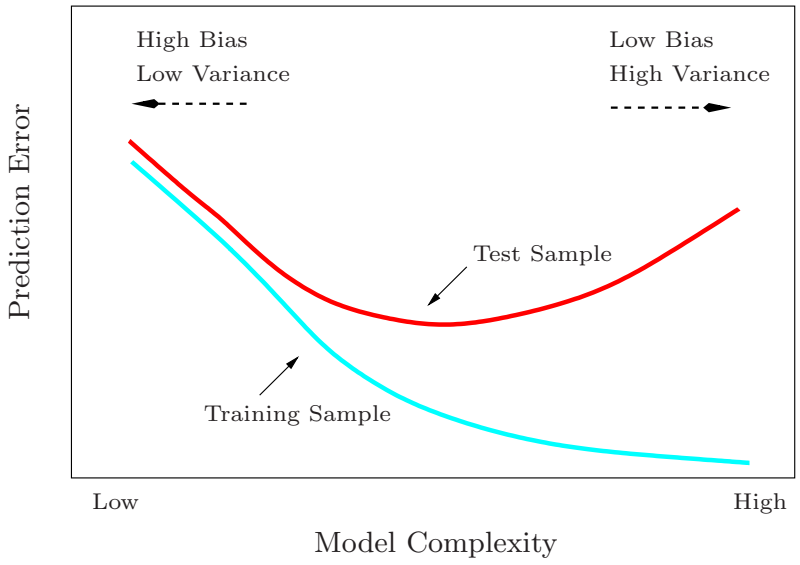


Figure 29: the bias-variance tradeoff

We will revisit the bias-variance more formally in one of the next lectures.

The magic in the methods we'll see in this lecture is that they allow us to escape the tradeoff in a certain sense. One meta-algorithm we'll see will reduce the bias of the learning algorithm it's applied to - without substantially increasing its variance. The other meta-algorithm we'll see will reduce variance without substantially increasing the bias. So, magic.

26.2 Ensemble / Committee methods

> "A collective wisdom of many is likely more accurate than any one." — Aristotle, in *Politics*, circa 300BC

It's been known for a long time that committees typically make better decisions than individuals. (The original Greek democracy basically comes down to this idea.)

Consider a committee of T members, which has to make a yes/no decision. In hindsight we'll know whether the decision has been right or wrong. Each member casts her vote. Each one has probability p of being correct and probability $1-p$ of being wrong. Let's assume for simplicity that all members are "equally wise", so that p is the same for all members. After all members vote, the committee's decision is simply the majority vote.

Exercise. Let $X_1, \ldots, X_T$ be i.i.d Bernoulli (p) random variables taking values in $\{\pm 1\}$. Show that the above committee's random decision is given by (when each member has equal probability p to be right) is simply $\overline{X} := sign(\sum_{t=1}^{T} X_t)$. Conclude that the probability that the above committee will make the right decision is $\mathbb{P}\left\{\overline{X} > 0\right\}$.

26.3 The uncorrelated case.

Accuracy of the committee's decision. It turns out that if each member is typically right $(p > 0.5)$, then the probability that the committee is right is much higher than any individual

member, and growing with the number of committee members T: **Exercise.** Assume that all members vote **independently** of each other. What is the probability that the committee's decision is right (namely, what is $\mathbb{P}\left\{\overline{X}=+1\right\}$ as function of p and T? What is the limit of this probability as $T\to\infty$? Plot the probability that the committee's decision is right over T, for $p=0.4,0.5,0.7,0.9$. (Note: A committee of fools is a terrible decision maker: Observe that if $p<0.5$, so that each member is usually wrong, then the majority vote will be worse than a single vote.)

Variance of the committee's decision. Now we wonder if the committee makes decisions **consistently**, namely, if it votes several times - each time the entire voting process is independent of all other times - how likely is the committee to make the same decision? So let's consider the **variability** in the committee's decision

Exercise. Given i.i.d real-valued random variables $X_1,\ldots,X_T$, each with variance σ^2, show that the variance of $\overline{X}:=T^{-1}\sum_{t=1}^{T}X_t$ is $(\sigma^2)/T$.

Exercise. What is the variance of a single Bernoulli random variable $X\sim Ber(p)$? What is the variance of the committee's vote $\overline{X}$ as function of T and p? plot this variance over T for $p=0.4,0.5,0.7,0.9$.

26.4 The correlated case.

In practice, however, committee members rarely vote independently. So let's assume that each two members are correlated with equal correlation $0\leqslant\rho\leqslant 1$. So that each member is right with (equal) probability p and each pair of members have (equal) correlation ρ.

Accuracy of the committee's decision.
Exercise. Assume every two committee members have equal correlation ρ as above, and that their individual decision-making process is the same. **Use a simulation** to plot the probability that the committee's decision is right (namely, what is $\mathbb{P}\left\{\overline{X}=+1\right\}$ as function of T, for $p=0.7$ and $\rho=0.2,0.5,0.9$. What do you think is the limit of this probability as $T\to\infty$?

Variance of the committee's decision.
Exercise. Given identically-distributed real-valued random variables $X_1,\ldots,X_T$, each with variance σ^2, and such that $corr(X_i,X_j)=\rho$ for all $1\leqslant i\neq j\leqslant T$, show that the variance of $\overline{X}:=T^{-1}\sum_{t=1}^{T}X_t$ is

$$\rho\cdot(\sigma^2)+(1-\rho)\cdot\frac{\sigma^2}{T}$$

Exercise. Use a simulation to plot the variance of the committee's vote $\overline{X}$ over T T, for $p=0.7$ and $\rho=0.2,0.5,0.9$. What do you think is the limit of this variance as $T\to\infty$?

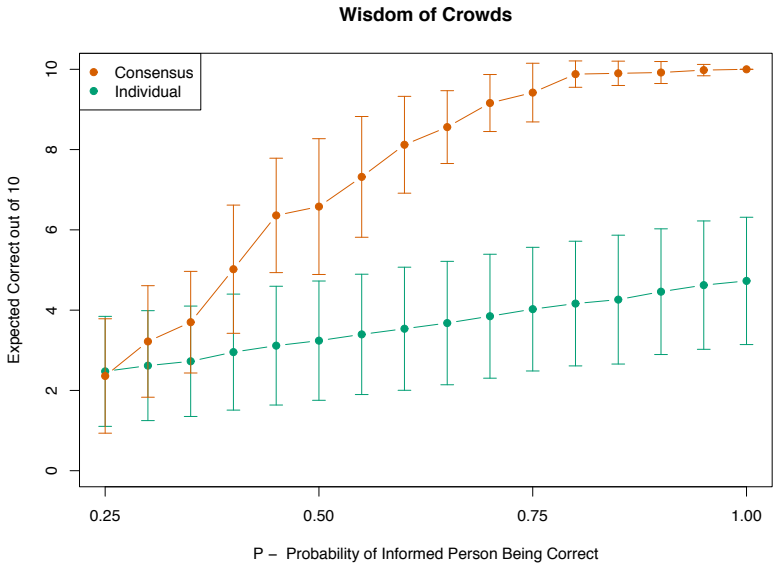


Figure 30: Simulation: wisdom of the crowd. The probability of the committee being right (and its standard deviation in errorbars) overlay with the probability (and standard deviation) of each member individually being right (ESL figure 8.11)

26.5 Summary

We have seen quantitatively the following general statement: A committee of members decide by majority vote. Decisions are correlated with correlation ρ and each member is right with probability p. If $p > 0.5$, the committee's decision will improve with the number of members T in two ways: it has been probability of being right, and will be more consistent (less variable). If $\rho > 0$, increasing T will only help up to a certain point, so ρ gives a bound on the improvement possible by moving from a single member to a whole committee.

27 Committee methods in machine learning

Back to machine learning. Suppose we had T training samples of size m chosen independently from $\mathcal{X}$ according to some distribution $\mathcal{D}$. Denote them by $S_1, \ldots, S_T$. Suppose have a learning algorithm $\mathcal{A}$ and train it on each of the training samples, to obtain $h_{S_1}, \ldots, h_{S_T}$. The prediction $h_{S_t}(x)$ on some sample $x \in \mathcal{X}$ is independent of all other predictions of the other trained rules, and has the same distribution. So if we used $h_{S_1}, \ldots, h_{S_T}$ in a committee - using majority vote - we have the situation from above. The generalization loss will improve with T, tending to 0 as $T \to \infty$ (if any rule h_{S_t} separately has generalization loss < 0.5). As we saw above, the variance of the prediction will decrease as $1/T$.

However, in batch learning, we don't have T training samples. We just have one. So why not train the same algorithm $\mathcal{A}$ again and again on the same training sample S? well, that won't do much good - the predictions will be identical - perfectly correlated.

So the first magic we would like to do is how to create T training samples from the one training sample S we have, in a way that will mimic fresh independent draws of new training samples of size m according to $\mathcal{D}$.

Committee methods - Definition. Committee methods are **"meta-algorithms"**. In a committee method, we take an existing learner $\mathcal{A}$ (which we call the "base" learner, or sometimes the "weak" learner, for reasons we will see below) and apply it to a sequence of T "artificial" training samples.

In this lecture we work with classification. The label set is $\mathcal{Y} = \pm 1$ and the committee decides by $h(x) = \text{sign}\left(\sum_{t=1}^{T} h_t(x)\right)$. Everything here applies to regression as well: in a regression problem, the committee decides by $h(x) = T^{-1}\left(\sum_{t=1}^{T} h_t(x)\right)$.

We are going to see two very different ideas for building the committee member rules. But After the committee is built, it is averaged the same way in all cases. Sometimes the committee's decision will be a weighted average, to allow some members tohave more weight than others: $h(x) = \text{sign}\left(\sum_{t=1}^{T} w_t h_t(x)\right)$, for some $w_t \geqslant 0$, $\sum w_t = 1$.

28 The Bootstrap

Here comes the first magic. Creating new "artificial" training samples from the one training sample S we have seems impossible. But yet it actually is, and the fact that it is is one of the most groundbreaking ideas of statistics in the 20th century.

Given a training sample $S = \{(x_i, y_i)\}_{i=1}^m$ we are going to construct a new training sample S^{*1} as follows. We are going to sample m times **with replacements** from the set S. The first sample we draw from S will be denoted (x_1^{*1}, y_1^{*1}). The second sample we draw will be denoted (x_2^{*1}, y_2^{*1}), and so on. So we now have a sample

$$S^{*1} = \{(x_i^{*1}, y_i^{*1})\}_{i=1}^m$$

Of course, since we sampled from S with replacements, there might be repeated samples in S^{*1}, even if S itself had no repeated samples. Now we can repeat this process B times, obtaining B training samples, each of length m: $S^{*1}, \ldots, S^{*B}$. The samples in the b-th training sample will be denoted

$$S^{*b} = \{(x_i^{*b}, y_i^{*b})\}_{i=1}^m .$$

This method of new training samples is called The Bootstrap, and the sample S^{*b} is called a bootstrap sample created from S.

28.1 Why does the Bootstrap work?

Assume for a moment that samples in our learning problem are i.i.d samples from an unknown distribution $\mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$. We are hoping that each Bootstrap from S somehow behaves like a fresh i.i.d sample from $\mathcal{D}$ itself.

It may seem at a first glance crazy that bootstrap samples can serve us instead of fresh, i.i.d samples from $\mathcal{D}$. But in fact it is often the case. Why?

Given a training sample S (assume for simplicity that all the points of S are distinct) let's define the **empirical distribution** $\widehat{\mathcal{D}}_S$ induced by S on $\mathcal{X} \times \mathcal{Y}$ as the following probability distribution on $\mathcal{X} \times \mathcal{Y}$: for a subset $C \subset \mathcal{X} \times \mathcal{Y}$, define

$$\widehat{\mathcal{D}}_S((X,Y) = (x,y)) = \begin{cases} \frac{1}{m} & (x,y) \in S \\ 0 & (x,y) \notin S \end{cases}$$

or equivalently, for any $C \subset \mathcal{X} \times \mathcal{Y}$,

$$\widehat{\mathcal{D}}_S(C) := \frac{|C \cap S|}{m}.$$

Observe that this is equivalent to putting a probability mass of $1/m$ on each of the points of S, and zero mass on all other points in $\mathcal{X} \times \mathcal{Y}$.

Now observe that a bootstrap sample S^{*b} is just an i.i.d draw of m points from the empirical distribution $\widehat{\mathcal{D}}_S$ induced by the one training sample we have, S.

As m grows, namely as S becomes larger, the empirical distribution $\widehat{\mathcal{D}}_S$ converges in distribution to $\mathcal{D}$. The idea behind the bootstrap is that, if $\widehat{\mathcal{D}}_S$ is not so different from $\mathcal{D}$, then m i.i.d draws from $\widehat{\mathcal{D}}_S$ is a good approximation to m i.i.d draws from $\mathcal{D}$.

One way to see the convergence of the empirical distribution to the underlying distribution is on the real line:

Exercise. Let $X_1, \ldots, X_m$ be i.i.d random variables with some distribution F on the real line $\mathbf{R}$. Let $x_i \in \mathbb{R}$ be a sampled value of X_i. Show that the CDF (cumulative distribution function) of this sample is a step function, increasing from 0 to 1 (as all CDFs should), with jump of size $1/m$ at each value x_i (see Figure 31). In simulation, take F to be, say, $\mathcal{N}(0,1)$. For each value $m = 10, 100, 1000$, draw a sample and plot its empirical CDF - and overlay the CDF of $\mathcal{N}(0,1)$. Recall that on the real line, convergence in distribution is equivalent to convergence of the CDFs to a limiting CDF at the continuity points of the limiting CDF.

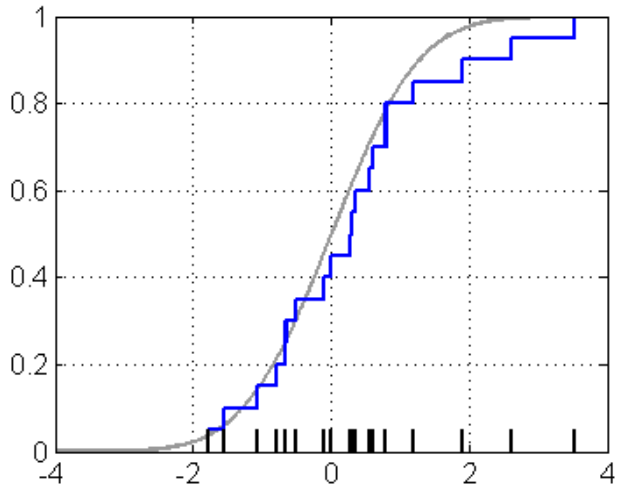


Figure 31: CDF of a probability distribution on the real line, and empirical CDF of an i.i.d sample from that distribution. Black lines on the horizontal axis show the random sample.

Why the name "The Bootstrap"? We're seemingly creating new datasets out of nothing, as if we were pulling ourselves up by the straps of our own boots. As you may know, there was only one man strong enough to do that - the Baron Munchausen.

Figure 32: The Great Baron Munchausen pulling himself up

Here's a question you might ask: How many point of S are left **out** of each bootstrap sample, typically? Answer: About a third.

Exercise. Show that, for m large, about 37% of data points are left out of a bootstrap sample created from S.

29 Bagging

The idea of Bootstrap samples can be used whenever we would like to create new artificial samples from our only training sample S. It has many uses throughout machine learning, statistics and data science. **Bagging** is a nickname for a straightforward use of the Boostrap in machine learning, to improve accuracy of an existing supervised machine learning algorithm.

We start with a "base" learning algorithm $\mathcal{A}$ and a training sample S. We choose T (later we'll discuss how to choose it) and form T bootstrap training samples, $S^{*1}, \ldots, S^{*T}$, each of size m. We then train our learner **separately** on each of the T bootstrap training samples. We

form the committee $h_{S^*1}, \ldots, h_{S^*T}$. We store all T trained models. When we need to classify a new test sample $x \in \mathcal{X}$, we run x through all the rules and classify using the majority vote of the committee,

$$h_{bag}(x) := sign\left(\sum_{t=1}^{T} h_{S^*t}(x)\right)$$

For example, if we run Bagging on top of the Decision Tree classifier, we'll obtain a committee of decision trees:

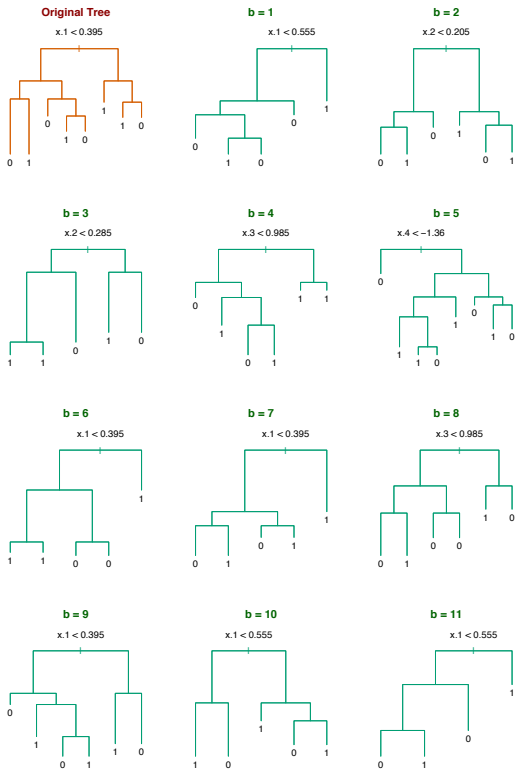


Figure 33: Collection of Bagged Decision Trees. (Source: ESL)

Handling repeated samples. Note that our learner $\mathcal{A}$ must know how to handle repeated samples. We may have them anyway in S, but running on a bootstrap sample we are sure to have them. Some learning algorithms don't like repeated samples - as they cause numerical problems (for example, linear and logistic regression.) Some really don't care (for example, decision trees and k-NN).

29.1 This is shockingly effective

Does Bagging a learning algorithm reduce its generalization error? This is what the original paper observed:

Data Set	$\bar{e}_S$	$\bar{e}_B$	Decrease
waveform	29.1	19.3	34%
heart	4.9	2.8	43%
breast cancer	5.9	3.7	37%
ionosphere	11.2	7.9	29%
diabetes	25.3	23.9	6%
glass	30.4	23.6	22%
soybean	8.6	6.8	21%

Figure 34: Improvement of Bagging Decision Trees over a single tree. From the original paper *Shang and Breiman, Distribution Based Trees Are More Accurate*

So a simple and straightforward trick can hugely reduce generalization risk.

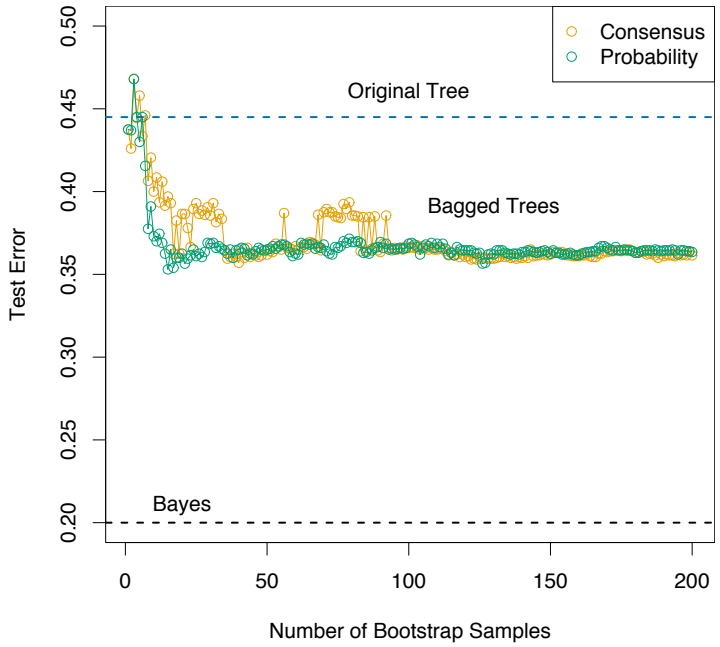


Figure 35: Test error of simple Bagging of decision trees, over T the number of bagged trees (Source: ESL)

29.2 Bagging reduces variance

We saw that a committee majority vote reduces variance - but only to a certain degree, which is determined by the correlation between committee members. So, we can expect bagging to reduce variance as T increases - and therefore to reduce the generalization error - but only to a certain degree, determined by the correlation between the bagged prediction rules. So, Bagging can be improved by somehow de-correlating the bagged prediction rules.

29.3 Decorrelation

How do we de-correlate the committee members - namely, cause their predictions somehow to be less correlated? One way to do this is by handicapping (restricting) each learner a little, in a random way, and hope that the performance gain (in bagging them) due to de-correlation is more than the performance loss to each learner by handicapping. The most well know example of this principle is **Random Forests.**

29.4 Random Forest: Bagging of Decision Trees + De-correlation

Recall the Decision Tree classification algorithm over $\mathcal{X} = \mathbb{R}^d$. We have a training sample S with m points. The Random Forest classifier is obtained by using Bagging on top of the Decision Tree algorithm, **with an important twist** for de-correlation: the algorithm has a tuning parameter $k \leqslant d$. When growing each decision tree, in each split, we choose k out of the d coordinate uniformly at random, and only choose the split among these k coordinates. Formally:

- The tuning parameters are:
  - $R \in \mathbb{N}$, the maximum depth of each tree
  - m_{min}, the minimal number of training samples in any leaf of any of the trees
  - T, the number of Bagging samples (number of trees in the forest)
  - k, the number of coordinates allows in choosing each split.
  - (There is an additional tuning parameter for **pruning** a decision tree which we'll discuss in a future lecture.)
- For each $t = 1 \dots T$:
  - Draw a Bootstrap sample S^{*t} from S
  - Train a decision tree $h_{S^{*t}}$ on the sample S^{*t}. While growing the tree, in each split do the following:
    - Select k coordinates from $\{1, \dots d\}$ uniformly at random
    - Pick the best (coordinate,split-point) combination using only the k coordinates chosen
    - Split on the best combination
  - Do not split a box if the maximal depth R or the minimal number of training samples m_{min} are reached.
- Output the grown trees $h_{S^{*1}}, \dots, h_{S^{*T}}$.

This de-correlation trick works: pretty much on every classification problem you'll work on, you'll observe something like the next plot: Bagging trees is much better than a single tree, and Random Forest (Bagging with the de-correlation trick) is better than just Bagging trees. (And, sometimes, Boosting trees is better than both - but we're coming up to that.)

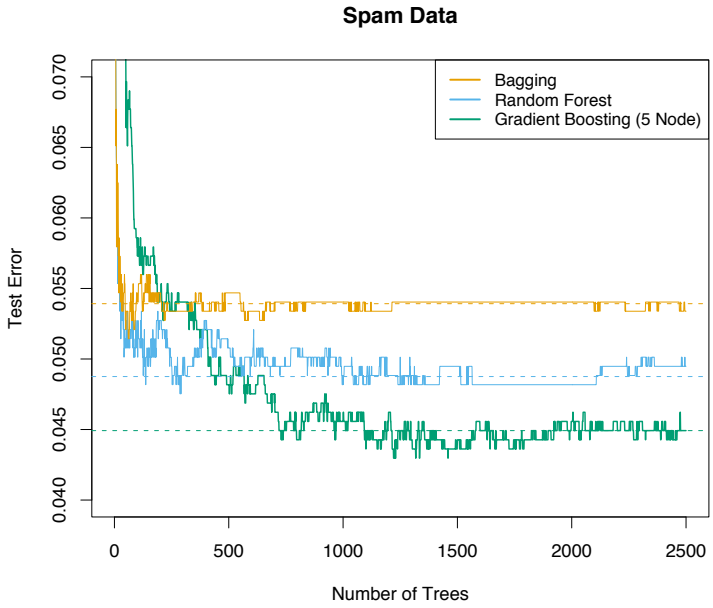


Figure 36: Test error of simple Bagging of decision trees (no de-correlation), Random Forests, and Gradient Boosting of Trees. (Source: ESL)

29.5 Some discussion points about Bagging

Can Bagging hurt us?

Always remember that a committee of fools (a committee where each member has probability $p < 0.5$ to make the right decision) makes worse decisions than a single member. So, when our base learner is so poor that its generalization loss is less than 0.5 we shouldn't use Bagging.

What are the disadvantages of Bagging?

- We need to train T models, not just one
- For prediction on new samples, we need to store T models, not just one
- Loss of interpretability: it's harder to understand why the committee made a decision - we need to understand the decision of each of the T members

Bagging and predicted class probabilities

Question: Can we use the **proportion** of the committee members who voted +1 as a predicted class probability?

Answer: It's not a good idea. Estimated class probabilities are estimates of $\mathbb{P}\{Y = +1, | X = x\}$. The proportion of members who voted +1 estimates $\mathbb{P}\{h_S(x) = +1\}$, which is a different quantity.

Parallel implementation of Bagging and Random Forests

From the computational perspective, it is important to note that Bagging in general (and Random Forests in particular) is **embarrassingly parallelizable**. When training a Bagging model with T committee members, we can use T machines in parallel, each using its own random seed to select Bootstrap samples (and random splits, in Random Forest). The machines do not need to interact; when each machine is done, it returns the committee member h_t to the master node.

Decision boundary in bagging

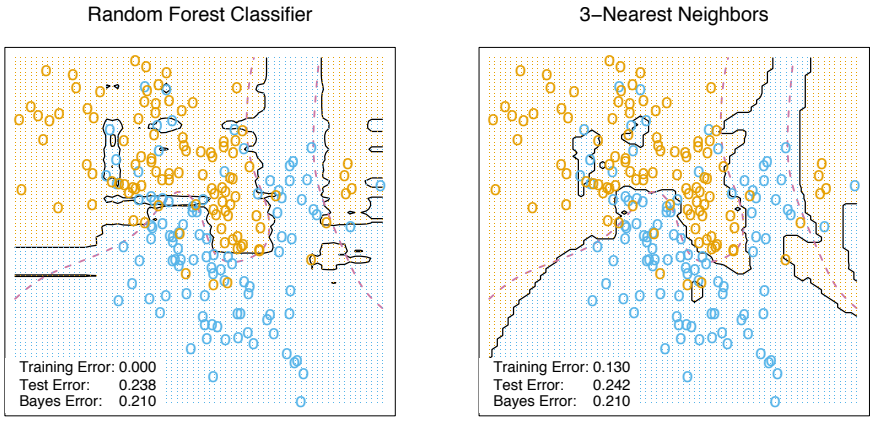


Figure 37: Left: Decision boundary of a Random Forest classifier. Right: Decision boundary of a $3-NN$ classifier. Observe that the Random Forest tends to have axis-parallel boundaries. (Source: ESL)

29.6 Random Forest classifier summary

Random Forest is a very popular classifier. As it doesn't overfit for T (number of trees) too large, we just try to avoid making T too large for efficiency reasons. For choosing k (the number of random coordinates allowed for each split), the rule of thumb is $k := \sqrt{d}$ where d the ambient dimension, $\mathcal{X} = \mathbb{R}^d$.

Exercise: Complete the following summary of the Random Forest Classifier (note that we still didn't learn how to **prune** each decision tree in the forest.)

- Hypothesis class
- Learning principle for training model (choosing $f \in \mathcal{H}$):
- Computational implementation of learning principle:
- How to make predictions on new samples:
- Interpretable
- Estimates class probabilities

- Family of models
- Time complexity for training, and for predicting on a new sample
- How to store trained model

30 Boosting

Bootstrap was magic of the following kind: we take a single training sample S and turn it into many training samples. Bagging uses this magic by training a model over these "new" training samples, and averaging the result to reduce the variance and hence the generalization error.

Boosting is magic of a different kind. In Boosting we take a "weak" learning algorithm - an algorithm with better-than-random but possibly not so good accuracy (accuracy = generalization error) and **boost** it - using a clever committee method - to obtain a learning algorithm with good accuracy.

The core magical idea of Boosting is a completely different idea for creating a committee of prediction rule from a base learning algorithm $\mathcal{A}$ and a single training sample S. In Bagging, we "pretended" to have fresh training samples $S_1, \ldots, S_T$, and each committee member trained on a different sample. In Boosting, we go even further and "pretend" to have **different underlying distributions $\mathcal{D}$ from which the training sample is drawn**.

More specifically, in Boosting each committee member h_t is the result of running $\mathcal{A}$ against a training sample S_t that mimics an i.i.d sample of size m from a **different distribution** D^t. Whereas in Bagging each committee member is trained independently of all other members, in Boosting the committee members are trained sequentially - one after the other - and each is an improvement, in some sense, on the previous one.

The clever idea behind Boosting is that after we finish training h_t, based on the distribution D^t, we update the distribution in a way that **increases the distribution at training samples where h_t was wrong**. This way, h_{t+1} will try very hard not to be wrong on those particular samples, and so on.

Here is a cartoon of how Boosting iterations progress:

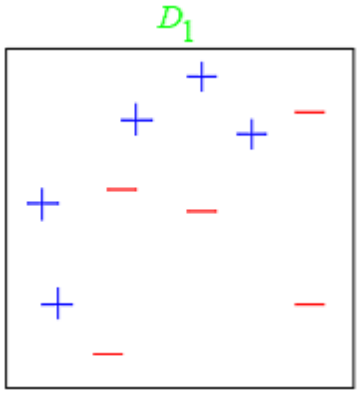


Figure 38: Original problem. Uniform distribution D^1.

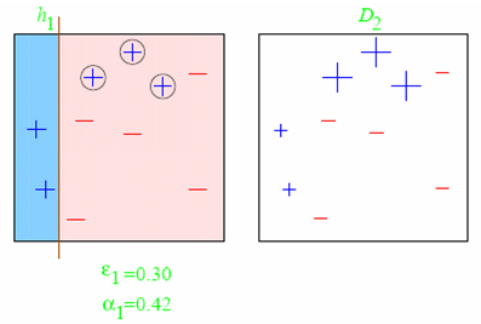


Figure 39: Iteration 1. Left: h_1 with D^1. Right: D^2

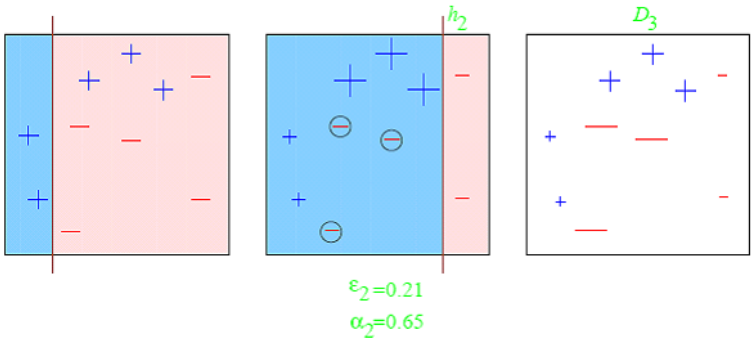


Figure 40: Iteration 2. Left: h_1 with D^1. Center: h_2 with D^2. Right: D^3

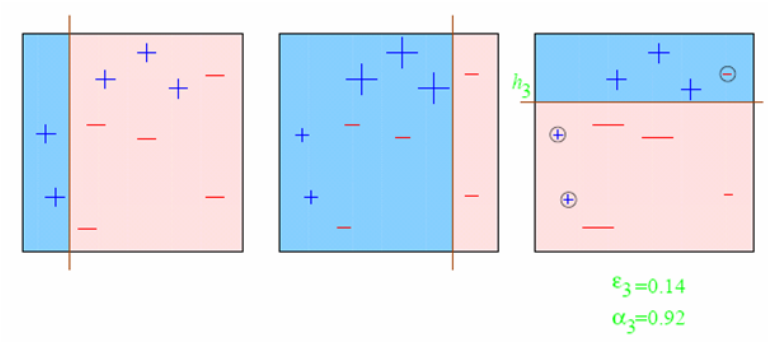


Figure 41: Iterations 3. Left: h_1 with D^1. Center: h_2 with D^2. Right: h^3 with D^3

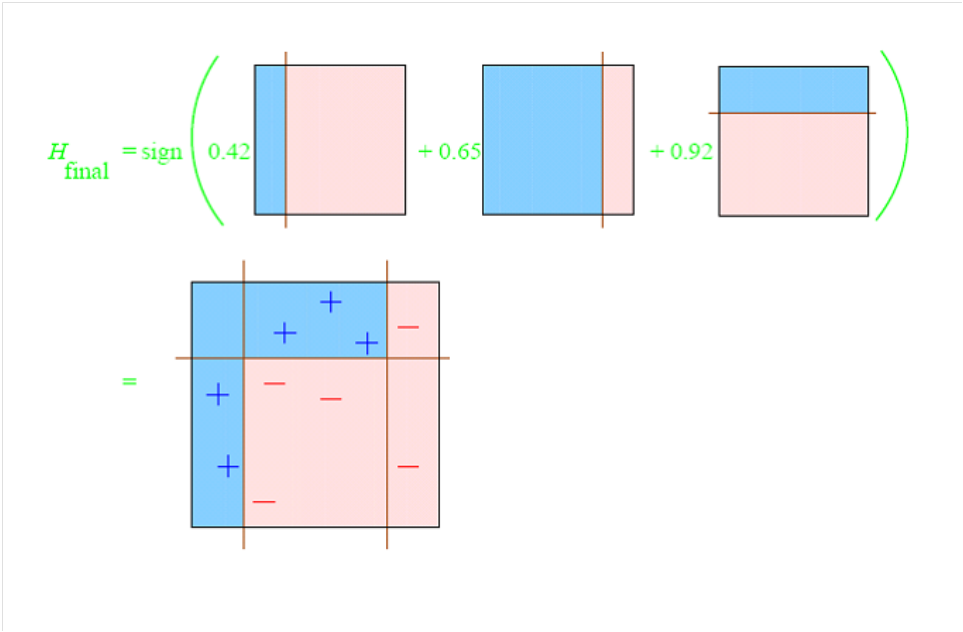


Figure 42: The Boosting committee

30.1 Classification problem with a weighted sample

But first, we have to understand what is meant, exactly, by "running $\mathcal{A}$ against the training sample S with distribution D^t".

One way to interpret this is to take a **weighted Bootstrap** sample from S, where the probability of selecting $(x, y) \in S$ is proportional to $D^t(x, y)$.

A simpler way to interpret this is as follows. If $\mathcal{A}$ uses the ERM principle, say for standard misclassification ($0-1$ loss), namely, looking to minimize the empirical risk,

$$L_S(h) = \sum_{i=1}^{m} \mathbf{1}_{[y_i \neq h(x_i)]}$$

then we can use S itself (not any bootstrap sample) and have the base learner minimize the **weighted** empirical risk

$$L_{S,D^t}(h) = \sum_{i=1}^{m} D_i^t \mathbf{1}_{[y_i \neq h(x_i)]}$$

where for each $(x_i, y_i) \in S$ we write $D_i^t := D^t(x_i, y_i)$, so that $\sum_{i=1}^m D_i^t = 1$.

Observe that these two interpretations are equivalent in expectation. Indeed, the expected number of times for a sample (x_i, y_i) to appear in the weighted Bootstrap sample is D_i^t, and so it would (in expectation) appear D_i^t) times in the empirical risk sum.

Note that we usually prefer second option (using weighted empirical risk) to the first option (using weighted bootstrap). It's more computationally efficient, and does not require worrying about repeated samples. However, the first option (using weighted bootstrap) is always available. The second option (using weighted empirical risk) is not always possible, and is implemented ad-hoc for the particular base learner we are boosting.

Exercise: To help you understand this point, describe how you would implement a Decision Tree with each of the two methods:

- Using weighted empirical risk: How would you change the Decision Tree algorithm we've seen (CART) to work with a given weight vector D^t over the training sample S? (Hint: what is the best splitting now that we have weights?)
- Using weighted Bootstrap: How would you How would you change the Decision Tree algorithm we've seen (CART) to work with a given weight vector without changing the splitting algorithm, namely, by giving the algorithm a different training sample selected by weighted Bootstrap? Will the algorithm work with repeated samples?

It is much less trivial to adapt learning algorithms that do not use the ERM principle. For example, Soft SVM can be adapted to work with weights, by penalizing the slack variables - but this is not trivial[20].

30.2 Adaboost

There are many Boosting meta-algorithms. The one we will learn here was the original one, known as **Adaboost** (for "**Ada**ptive **Boost**ing")

Adaboost, as a form of boosting, is characterized as by the following statements:

- Set the initial distribution to be uniform, $D_i^1 = 1/m$, $i = 1, \ldots, m$
- Use exponential updates for the distribution

  $$D_i^{t+1} \leftarrow \frac{D_i^t \cdot e^{-w_t y_i h_t(x_i)}}{\sum_{j=1}^m D_j^t \cdot e^{-w_t y_j h_t(x_j)}}$$

  for some **exponent** $w_t > 0$. (Notice how if point i is classified correctly then $y_i h_t(x_i) = 1$, so its weight goes down in the next iteration. Conversely, if point i is classified incorrectly then $y_i h_t(x_i) = -1$, so its weight goes up.)

[20] Look for "boosting support vector machines."

- Choose the exponent w_t exactly such that

$$\sum_{i=1}^{m} D_i^{t+1} \mathbf{1}_{[y_i \neq h_t(x_i)]} = \frac{1}{2}\,.$$

  (We'll soon discuss why.)

- Each committee member h_t votes with weight w_t, so that the label predicted by the committee is

$$h_{boost}(x) := sign\left(\sum_{t=1}^{T} w_t h_t(x)\right)$$

The idea is simple: from iteration t to iteration $t+1$, we want to **increase** the weights of samples misclassified by h_t (where $y_i h_t(x_i) = -1$) and **decrease** the weights of samples correctly classified by h_t. We want to make the classification problem "maximally hard" in the sense that weighted empirical risk of h_t, with respect to the updated weights D^{t+1}, is the worse possible, namely 1/2. Finally, the prediction rules vote in the committee with weights w_t.

Claim: The exponent we are looking for is

$$w_t := \frac{1}{2}\log\left(\frac{1}{\epsilon_t} - 1\right)$$

where $\epsilon_t = \sum_{i=1}^{m} D_i^t \mathbf{1}_{[y_i \neq h(x_i)]}$ is the weighted empirical risk of h_t.

Proof:

$$\sum_{i=1}^{m} D_i^{t+1}\, \mathbf{1}_{[y_i \neq h_t(\mathbf{x}_i)]} = \frac{\sum_{i=1}^{m} D_i^t\, e^{-\mathbf{w}_t y_i h_t(\mathbf{x}_i)} \mathbf{1}_{[y_i \neq h_t(\mathbf{x}_i)]}}{\sum_{j=1}^{m} D_j^t e^{-\mathbf{w}_t y_j h_t(\mathbf{x}_j)}}$$

$$= \frac{e^{\mathbf{w}_t}\epsilon_t}{e^{\mathbf{w}_t}\epsilon_t + e^{-\mathbf{w}_t}(1-\epsilon_t)} = \frac{\epsilon_t}{\epsilon_t + e^{-2\mathbf{w}_t}(1-\epsilon_t)}$$

$$= \frac{\epsilon_t}{\epsilon_t + \frac{\epsilon_t}{1-\epsilon_t}(1-\epsilon_t)} = \frac{1}{2}\,.$$

You may be wondering why the right weights for the committee votes are given by the same exponents w_t we used to update the distribution. This will become clear next.

To recap, here is the Adaboost meta-algorithm:

- **input:** training set $S = (\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)$; base ("weak") learner $\mathcal{A}$ that takes both a training sample of size m and a distribution on m points; number of rounds T.
- **initialize** $D^1 = (\frac{1}{m}, \ldots, \frac{1}{m})$
- **for** $t = 1, \ldots, T$
  - invoke base learner $h_t = \mathcal{A}(D^t, S)$
  - compute $\epsilon_t = \sum_{i=1}^{m} D_i^t\, \mathbf{1}_{[y_i \neq h_t(\mathbf{x}_i)]}$

- let $w_t := \frac{1}{2}\log\left(\frac{1}{\epsilon_t} - 1\right)$
- update $D_i^{t+1} = \frac{D_i^t \exp(-w_t y_i h_t(\mathbf{x}_i))}{\sum_{j=1}^m D_j^t \exp(-w_t y_j h_t(\mathbf{x}_j))}$ for all $i = 1, \ldots, m$

- **output** the hypothesis $h_{boost}(\mathbf{x}) = sign\left(\sum_{t=1}^T w_t h_t(\mathbf{x})\right)$.

30.3 PAC view of boosting

Historically, Boosting appeared as an answer to a fascinating question. Let us formulate this question in PAC framework.

Definition 22 (γ-weak-learner) *A learning algorithm $\mathcal{A}$ is a γ-weak-learner for an hypothesis class $\mathcal{H}$ if there exists a function $m_{\mathcal{H}} : (0,1) \to \mathbb{N}$ such that for every $0 < \delta < 1$, for every distribution $\mathcal{D}$ over the sample space $\mathcal{X}$, and for every labeling function $f : \mathcal{X} \to \{\pm\}$, if the realizabiity assumption holds with respect to $\mathcal{H}, \mathcal{D}, f$, then when running $\mathcal{A}$ on a training sample of $m \geqslant m_{\mathcal{H}}(0,1)$ i.i.d samples drawn according to $\mathcal{D}$ and labeled by f, the algorithm returns an hypothesis $h_S = \mathcal{A}(S)$ such that wwith probability at least $1 - \delta$ (with respect to choice of the training sample S), we have $L_{\mathcal{D},f}(h_S) \leqslant 1/2 - \gamma$.*

An hypothesis class $\mathcal{H}$ is γ-weak-learnable if there exists a γ-weak-learner for $\mathcal{H}$.

How is this different than PAC-learnability? If an hypothesis class $\mathcal{H}$ is PAC-learnable, then for **every** (ϵ, δ) there exists a learner $\mathcal{A}$. This means that we can learn and generalize a labeling function from $\mathcal{H}$ to any accuracy ϵ we want. But if $\mathcal{H}$ is γ-weak-learnable, for any δ **and just for** $\epsilon = 1/2 - \gamma$ there is a learner $\mathcal{A}$. We may not be able to find a learner that has better accuracy (lower ϵ).

The question that motivated Boosting was the following:

- Suppose that $\mathcal{H}$ is PAC-learnable. Then we know that the rule $ERM_{\mathcal{H}}$ will learn (namely, will be probably approximately correct etc) with a near-minimal number of samples.
- But what if $ERM_{\mathcal{H}}$ is computationally hard? (we've seen examples)
- Assume we can find a **simple** hypothesis class (a "base hypothesis class") $\mathcal{H}_{base}$, such that $ERM_{\mathcal{H}_{base}}$ (choosing the hypothesis in $\mathcal{H}_{base}$ with lowest empirical risk) is computationally efficient, and is γ-weak-learner for $\mathcal{H}$ for some γ.
- This means that we have a computationally efficient way to learn with accuracy $1/2 - \gamma$, for some γ. Maybe we can't find an efficient learner with better γ.
- Is there a way to **boost** $ERM_{\mathcal{H}_{base}}$ in a computationally efficient way, and create a computationally efficient learner $\mathcal{A}$ which is close to minimizing ERM over $\mathcal{H}$?

For example, think about Decision trees. We saw that the ERM learner is not computationally feasible on this hypothesis class. But a small tree may be able to achieve accuracy (over a sample labeled by a larger tree) which is not great, but better than random.

Well, as the following theorem shows, Adaboost does just that. (We won't prove this theorem - you can see the proof in UML 10.2).

Theorem. Let S be a training set. Assume that at each iteration of Adaboost, the base learner returns a prediction rule (hypothesis h_t) for which the weighted empirical risk satisfies

$$\sum_{i=1}^{m} D_i^t \mathbf{1}_{[y_i \neq h(x_i)]} \leqslant \frac{1}{2} - \gamma \,.$$

Then the (standard, non-weighted) empirical risk of the output prediction rule of Adaboost, h_{boost}, (the weighted committee vote) satisfies

$$L_S(h_{boost}) \equiv \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}_{[y_i \neq h_{boost}(x_i)]} \leqslant e^{-2\gamma^2 T} \,.$$

30.4 Bias and variance in boosting

The hope is, of course, that we're not overfitting, so that low empirical risk will imply low generalization loss.

Suppose we run T iterations of Adaboost over a learner $\mathcal{A}_{base}$ that returns hypothesis from $\mathcal{H}_{base}$. What is the effective hypothesis class we have now, and how large is it?

Well, Adaboost with T iterations will return a function from the hypothesis class

$$\mathcal{H}_T = \left\{ x \mapsto \sum_{t=1}^{T} w_t h_t(x) \;\middle|\; w_1 \ldots w_T \in [0,\infty),\ \sum_t w_t = 1,\ h_1 \ldots, h_T \in \mathcal{H}_{base} \right\}$$

namely convex combinations of hypotheses from $\mathcal{H}_{base}$. So $\mathcal{H}_t$ becomes larger (contains more functions) as T grows. But it doesn't grow too fast with T.

For example, we have a canonical way to measure the "size" of $\mathcal{H}_T$. While we won't go into the details, under certain conditions, $VCdim(\mathcal{H}_T)$ is roughly $T \cdot VCdim(\mathcal{H}_{base})$. So we can expect Boosting to increase the variance (compared with the base learner) as T increases, but "not too fast".

On the other hand, it's clear that Boosting decreases bias - that is obvious from the fact that the empirical risk decreases as T grows - indeed $\mathcal{H}_T$ is able to come closer and closer to the labeling function on the training set. And the fact that empirical risk decreases **exponentially** with T tells us that bias decreases quite quickly.

Overall, Boosting typically decreases bias much faster than it increases variance, which is why it typically improves generalization loss quite dramatically.

Question for you: If we use T too large, will boosting overfit?

30.5 It's often better to Boost very simple learners

Very often we see that boosting ERM over a very simple base hypothesis class is better than boosting ERM over a more complicated class. For example, here is the test error (number

of misclassification errors on a test set the algorithm has never seen) of boosting **Decision stumps** - Decision trees with a single split - over number of iterations T, compared with the test error of a single stump, and with a single large Decision tree:

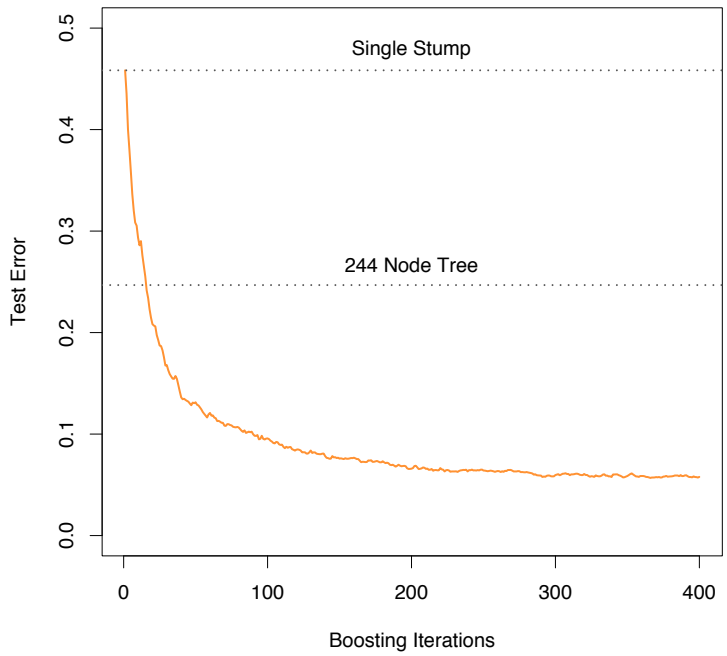


Figure 43: Test error of boosting decision stumps (single level decision trees), with Adaboost over the number of boosting iterations T.

31 Bagging vs Boosting - Comparison

Bagging and Boosting are both committee methods, but they are very different. Let's compare them:

	Bagging	**Boosting**
Learns committee members:	in parallel	sequentially
Dataset for each committee member:	bootstrap training samples	weighted bootstrap **or** original S with weighted ERM
De-correlation:	recommended	not necessary
When T is too large	Does not overfit	may overfit
Cause of improvement in generalization error:	reduces variance	reduces bias
With decision trees, use:	deep trees	shallow trees
Parallel compute implementation:	yes - easy	no
Committee vote:	unweighted	weighted

Table 1: Comparison of Bagging and Boosting

32 Summary

We saw that a committee of learners using majority vote will have better accuracy than a single member if each member is better than a random guess; the improvement in accuracy due to the majority vote improves as the size of the committee grows, but is bounded from below by the correlation between the members.

We saw three general methods:

- **Bootstrap** is a method for generating "new" training sample from the one training sample we have.
- **Bagging** is a committee method where we run the learner against bootstrap samples. Learners are unrelated to each other. All have the same voting weight.
- **Boosting** is a committee method where we run the learner sequentially on weighted bootstrap samples. The weights are larger for samples where we made a mistake in the last iteration. Learners vote with weights related to their empirical loss.

These methods implement three general principles:

- We can create "artificial" training sets from our one training set S by sampling from S with replacements. This method is known as **The Bootstrap.** In a typical Bootstrap sample, about a third of the points a left out and others appear more than once.
- We can create a learner with **improved accuracy** and **reduced variance** by averaging base learners. The base learners **must** be better than a random guess. When each base learner gets a Bootstrap training sample, this is called **Bagging.** Bagging can be done in parallel as each prediction rule is created independently of the others. The prediction accuracy of the Bagging learner improves if the different prediction rules used are as de-correlated as possible. (Example: Random Forest is a classifier than achieves de-correlation by restricting each split in each tree to a random subset of coordinates.)
- We can create a learner with with improved accuracy by **boosting** a base learner. The key idea behind Boosting is working with a probability distribution over S. Boosting means creating a **weighted** committee of prediction rules. Rules are created sequentially (not in parallel). Each rule is created the previous rule by modifying the distribution in such a way that misclassified training samples get an increased weight. (Example: Adaboost is a Boosting method that uses exponential updates to the probability distribution on S, such that the weighted empirical risk of the previous rule according to the updated distribution is exactly 1/2 - the worse it can be.)

Lecture 7: Regression, Regularization, Model Selection and Model Evaluation

Suggested Reading:

- CART Regression Trees ESL 9.2.2
- Pruning CART regression and regularization trees ESL 9.2
- ℓ_2-regularized linear regression (Ridge regression): ESL 3.4.1, 3.4.3, UML 13.1.1.
- ℓ_1-regularized linear regression (Lasso): ESL 3.4.2, 3.4.3, UML 25.1.3
- ℓ_1-regularized logistic regression ESL 4.4.4

33 Introduction

Recall that we divided supervised learning into **regression** problems (label set $\mathcal{Y} = \mathbb{R}$) and **classification** problems (binary, where $\mathcal{Y} = \{\pm 1\}$ or multiclass, where $\mathcal{Y} = \{1, \ldots, k\}$).

So far in this course we've paid little attention to **regression** problems. We only saw linear regression.

So after talking a lot about classification, in this lecture we come back to regression problems. As in the classification lecture, we will work on the sample space $\mathcal{X} = \mathbb{R}^d$ - the d-dimensional Euclidean space, so that each ssample is a feature vector with d real entries. Many modern regression methods on $\mathcal{X} = \mathbb{R}^d$ use a principle called **Regularization**.

So in this lecture we will introduce the concept of **regularization**, which is a fundamental concept in machine learning. We will then see four different modern regression methods, which are all different examples of regularization. To keep classification in the picture, we will finish with a classification method using regularization.

34 Regularization

34.1 The setup: Choosing $h \in \mathcal{H}$ by minimizing fidelity

Regularization is a principle that allows us to build a continuous family of learners $\mathcal{A}_\lambda$, all producing hypotheses from a single hypothesis class $\mathcal{H}$.

Let's think about general $\mathcal{X}$ and $\mathcal{Y}$ - so this section works for both regression and classification. Assume that we have a learner $\mathcal{A}_0$ that chooses an hypothesis $h_S \in \mathcal{H}$ from a given hypothesis class $\mathcal{H}$, based on minimizing some cost function $\mathcal{F}_S(h)$ over $h \in \mathcal{H}$. (Obviously, this cost function depends on the training sample S.) This means that $h_S = \mathcal{A}_0(S)$ is given by

$$h_S := \text{argmin}_{h \in \mathcal{H}} \mathcal{F}_S(h) \, .$$

The function $\mathcal{F}$ measures how well h fits the training sample S. We can call $\mathcal{F}(h)$ the **fidelity**

term.

We have seen a few examples for such learners already:

- In linear regression, $\mathcal{F}_S(h)$ measures the **sum of squares**
- In logistic regression, $-\mathcal{F}_S(h)$ is the **likelihood** for the logistic regression model (which we want to maximize)
- In SVM, $-\mathcal{F}_S(h)$ is the **margin** (which we want to maximize)

The best example for learners that minimize a cost function is of course **ERM**: In any learner based on **Empirical Risk Minimization**, we define some loss function $\ell(\cdot,\cdot)$ and define the empirical risk induced by ℓ, with respect to the training sample $S = \{(x_i, y_i)\}_{i=1}^m \subset (X \times \mathcal{Y})^m$, to be

$$L_S(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(x_i), y_i)\,.$$

So an ERM learner fits in this framework if we take the fidelity term $\mathcal{F}_S(h)$ to be the empirical risk $L_S(h)$, and choose h_S using

$$h_S = \text{argmin}_{h\in\mathcal{H}} L_S(h)\,.$$

34.2 Adding a regularization term

If the hypothesis class $\mathcal{H}$ is "too large", we are concerned that minimizing $\mathcal{F}_S$ over $\mathcal{H}$ may lead to over-fitting - namely, we are concerned our learner will output an hypothesis h_S which will not generalize well, because it's too well adapted to the particular training sample S.

One way to solve this problem would be to restrict $\mathcal{H}$. But a more elegant and flexible way is to leave $\mathcal{H}$ large, but change the learner $\mathcal{A}_0$. We do this by changing the optimization problem that $\mathcal{A}_0$ uses to choose h_S. Instead of

$$h_S := \text{argmin}_{h\in\mathcal{H}} \mathcal{F}_S(h)\,.$$

we choose $\lambda \geqslant 0$ and define the learner $\mathcal{A}_\lambda : S \mapsto h_S$ by

$$h_S := \text{argmin}_{h\in\mathcal{H}} \left[\mathcal{F}_S(h) + \lambda\mathcal{R}(h)\right]\,.$$

The term $\mathcal{R}$ is called the **regularization term**. The value $\mathcal{R}(h)$ will measure the "complexity" of the hypothesis h. The more complicated the hypothesis h, the larger $\mathcal{R}(h)$ will be.

So we see that in minimizing $\mathcal{F}_S(h) + \lambda\mathcal{R}(h)$ we now have a **tradeoff**:

- On one hand, more complicated h, the better it can describe the training sample S, so the fidelity term $\mathcal{F}_S(h)$ will be smaller.
- On the other hand, the more complicated h, the larger $\mathcal{R}(h)$ will be.

And conversely, the simpler h, the larger the fidelity term $\mathcal{F}_S(h)$ (as it won't be able to describe the training sample very well) but the smaller $\mathcal{R}(h)$ will be.

So λ is a tradeoff parameter:

- For $\lambda = 0$, we have no regularization and are back to finding a minimizer of $\mathcal{F}_S(h)$.
- For $\lambda \to \infty$, the minimization problem pays no attention to the fidelity term and just wants to find the simplest possible $h \in \mathcal{H}$.
- Any finite value $\lambda \in (0, \infty)$ defines a specific tradeoff between the need for fidelity (small $\mathcal{F}_S(h)$) and the need for simplicity of h (small $\mathcal{R}(h)$).

So, when we add regularization to the learner $\mathcal{A}_0 : S \mapsto \text{argmin}_{h \in \mathcal{H}} \mathcal{F}_S(h)$, we won't find a minimizer of $\mathcal{F}_S(h)$ over $\mathcal{H}$. We are hoping that a minimizer of the regularized objective function $\mathcal{F}_S(h) + \lambda \mathcal{R}(h)$ (which does not minimize $\mathcal{F}_S(h)$) will generalize better. This is because, the larger the value of λ, the simpler this minimizer will be.
We therefore get a **family** of learners $\{\mathcal{A}_\lambda\}_{\lambda \in [0,\infty)}$. The regularization parameter λ controls the bias-variance tradeoff: for $\lambda = 0$ we get the most variance and least bias (most complicated h); for $\lambda \to \infty$ we get the least variance and most bias (simplest h).

Example: Soft SVM. We already saw one example for regularization - Soft SVM. Recall that the Soft-SVM classifier classifies using the half-space $\mathbf{w}^\perp$ where $\mathbf{w}$ is a solution to the optimization problem

$$\begin{aligned} \text{minimize} \quad & \lambda \|\mathbf{w}\|^2 + \frac{1}{m} \sum_{i=1}^{m} \xi_i \\ \text{subject to} \quad & y_i \cdot (\langle \mathbf{x}_i \, , \, \mathbf{w} \rangle + b) \geqslant 1 - \xi_i \text{ and } \xi_i \geqslant 0 \end{aligned}$$

Here, the fidelity term is $\|\mathbf{w}\|^2$. The smaller $\|\mathbf{w}\|^2$, the better we fit the training data (in the sense of larger margin). Indeed, we saw in the recitation that the total margin is proportional to $1/\|\mathbf{w}\|$, so that minimizing $\|\mathbf{w}\|^2$ means maximizing the margin. The regularization term is $\frac{1}{m}\sum_{i=1}^{m} \xi_i$. The smaller this term is, the "simpler" the hypothesis since we allow less violations of the margin[21].

34.3 Let's focus on Euclidean sample space, regression problems and ERM fidelity for the square loss

For most of this lecture, we'll focus on **regression** problems on Euclidean sample space, so from now on, $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}$. We'll focus on the most common fidelity term - the empirical risk $\mathcal{F}_S(h) = L_S(h)$. Recall that the empirical risk L_S is induced by a loss function $\ell(\cdot, \cdot)$. When we talked about classification problems, most of the time we worked with the $0-1$ loss (the misclassification loss). In this lecture, when we talk about regression problems, we will work

[21] You'll note that here λ is placed on the fidelity term, not on the regularization term. That happens sometimes. But in this lecture we'll use the structure $\mathcal{F}_S(h) + \lambda \mathcal{R}(h)$ and put λ in front of the regularization term.

with the **squared loss** $\ell(h(x), y) = (h(x) - y)^2$. So the empirical risk will be the sum of squares that we have seen when we talked about linear regression:

$$L_S(h) = \sum_{i=1}^{m} \left(h(x_i) - y_i\right)^2 .$$

We will cover four modern regression methods - these will be the most advanced regression methods we will learn in this course. They are all based on adding a regularization term to ERM. The methods are:

- CART regression trees with pruning
- Linear regression with ℓ_0 regularization, related to **best subset selection**
- **Ridge regression**: Linear regression with ℓ_2 **regularization**
- **The Lasso**: Linear regression with ℓ_1 **regularization**

Near the end of the lecture we'll see a completely different example: ℓ_1-regularized logistic regression. In that example, we will add a regularization term to fidelity based on maximum likelihood - not on ERM - in a classification problem. That will allow us to remember that the regularization principle is very general and not at all specific to regression, or to ERM fidelity.

35 CART Regression Trees

Our first example for adding regularization to ERM is the CART algorithm for **Regression Trees**. We have only seen CART classification trees, so before we get to adding regularization, let's start by defining a regression tree and see that we already understand the CART algorithm for fitting a regression tree.

This section assumes you and understand classification (decision) trees, and the CART algorithm for growing a classification tree. If you don't, it is recommended that you review the classification lecture handout.

35.1 Regression Trees

For a Tree Partition $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$ of $\mathbb{R}^d$ into N axis-parallel boxes, and label assignments $c_j \in \mathbb{R}$ $(j = 1, \ldots, N)$ assigning label $c_j \in \mathbb{R}$ to box B_j, the regression tree $h \in \mathcal{H}_{RT}$ is a function $h : \mathbb{R}^d \to \mathbb{R}$ defined by

$$h(\mathbf{x}) = \sum_{j=1}^{N} c_j \mathbf{1}_{B_j}(\mathbf{x})$$

where $\mathbf{1}_{B_j}$ is the indicator function of B_j.

The only difference from a classification tree is that now the labels c_j are real numbers, not classes in $\{\pm 1\}$. Any function in this hypothesis class corresponds to a decision tree leading to a **numerical** decision.

35.2 Growing a CART regression tree

As for classification trees, given a training sample S, we would ideally like to search $\mathcal{H}_{RT}$ for a tree minimizing the empirical risk - namely, to learn using the ERM principle. But as for classification trees, this is computationally hard, so must use a heuristic to choose the tree based on the training sample.

Suppose that we have already somehow decided to use a certain Tree Partition of $\mathbb{R}^d$ that consists of N disjoint boxes, $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$. Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^{m}$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ be our training sample.

Exercise. Let $y_1, \ldots, y_k \in \mathbb{R}$ be real numbers. Then

$$\operatorname*{argmin}_{c \in \mathbb{R}} \sum_{i=1}^{k} (y_i - c)^2$$

is simply the sample average

$$ave(y_1, \ldots, y_k) = \overline{y} = \frac{1}{k} \sum_{i=1}^{m} y_i \,.$$

This means that given an existing tree partition, the empirical risk would be minimized by choosing the label $c_j := ave(y_i \,|\, y_i \in B_j)$, namely, the average of the labels of the training samples in B_j.

The CART heuristic for growing a regression tree starts with $B_0 = \mathbb{R}^d$ and recursively splits each existing box B as follows:

- for each coordinate $i \in \{1, \ldots d\}$, and each potential chopping value $t \in \mathbb{R}$, let $g_i(t)$ be the the lowest empirical misclassification risk (with respect to the training sample S) incurred by chopping the box B at value t along coordinate i. Calculate the value $g_i(t)$ for each i and t as follows:
  - For each $t \in \mathbb{R}$ (such that t is a valid chopping point for B along coordinate i) let $B_{+,t} := \{\mathbf{x} \in \mathbb{R} \,|\, x_i > t\}$ and $B_{-,t} := \{\mathbf{x} \in \mathbb{R} \,|\, x_i \leqslant t\}$ by the two boxes obtained from B by chopping the box B along coordinate i at the value t.
  - Now let $\hat{y}_S(B_{\pm,t})$ be the class assignment for box $B_{\pm,t}$ that minimizes the empirical risk for square loss (with respect to training sample S). Namely, $\hat{y}_S(B_{+,t}) = ave(y_i \,|\, y_i \in B_{+,t})$ and similarly $\hat{y}_S(B_{-,t}) = ave(y_i \,|\, y_i \in B_{-,t})$. Now let $L^S_{\hat{y}_S(B_{\pm,t})}$ be that optimal empirical risk for square loss incurred by these class assignments.
  - Now, let $g_i(t) = L^S_{\hat{y}_S(B_{+,t})} + L^S_{\hat{y}_S(B_{-,t})}$. Convince yourself that this is the best empirical risk that can be incurred when chopping the box B at value t along coordinate i. A simple formula for $g_i(t)$ is

$$g_i(t) = \sum_{i:\mathcal{V}x_i \in B_{+,t}} (y_i - \hat{y}_S(B_{+,t}))^2 + \sum_{i:\mathcal{V}x_i \in B_{-,t}} (y_i - \hat{y}_S(B_{-,t}))^2 \,.$$

- Let $t_i := argmin_{t \in \mathbb{R}}\ g_i(t)$ be the **best** chopping point along coordinate i. Calculate t_i for each coordinate $i = 1, \ldots d$.

- Let $i_* := \text{argmin}_{1 \leqslant i \leqslant d} g_i(t_i)$ be the **best** coordinate along which to chop B.
- Now chop B along coordinate i_* at the value t_{i_*}.

We continue chopping each box into two boxes unless:

- The box is at the maximal depth specified, or
- The number of training samples in the box is smaller than the minimal number specified.

So the CART algorithm for growing a regression tree (when using the square loss) is exactly the same as the algorithm for growing a classification tree, just using averages instead of majority votes for the labels.

35.3 Pruning a CART regression tree - using regularization

We finally get to our first regularization example.

The CART algorithm for growing a regression tree results in a mostly balanced tree. As we mentioned in the classification lecture, we may have done too many splits. Each split we do decreases the bias (as we have a more complicated hypothesis, with more expressive power) but also increases the variance (as the labels for prediction in the leaf are averaged over fewer training samples).

The last stage of CART (for both regression and classification trees, but let's talk about regression trees now) is **pruning** the grown tree. (To Prune in English means to cut off some branches off a tree.) In the context of regression and classification trees, this means decreasing the tree size by merging together some boxes we've chopped in the growing stage - to reach a better bias-variance tradeoff that will lead to better generalization.

Consider a training sample S and a regression tree T induced by a partition $\mathbb{R}^d = \biguplus_{j=1}^{N} B_j$. The empirical risk (for square loss) of T on S is

$$L_S(T) = \sum_{j=1}^{N} \sum_{i \,:\, \mathcal{V} x_i \in B_j} (y_i - \hat{y}_S(B_j))^2 \,.$$

This will be our fidelity term $\mathcal{F}_S(T)$.

Now let T_0 be the fully grown tree obtained from the growing stage of CART. Let's write $T \subset T_0$ if the tree T is a sub-tree of T_0, namely, if T is obtained from T_0 by merging some boxes of T_0.

For our regularization term we simply use $\mathcal{R}(T) = |T|$, where $|T|$ denotes the number of leaves (boxes) of T - the same quantity we denoted above by N. This is a good measure for complexity of an hypothesis in $\mathcal{H}_{RT}$: more leaves is a more complicated tree, since it uses a finer partition of $\mathbb{R}^d$ into boxes.

Our regularization problem will be

$$\min_{T \subset T_0} \left[L_S(T) + \lambda \cdot |T| \right] . \tag{6}$$

Namely among all the subtrees of our fully grown T_0, we look for the one optimally balancing empirical risk and hypothesis complexity.

Note that we do **not** optimize this objective over the entire hypothesis class $\mathcal{H}_{RT}$ (that would be infeasible) - just over subtrees of the large tree we grew by the CART greedy splitting.

Is there an efficient implementation? Yes. It turns out that there is a simple, efficient algorithm for solving the minimization problem Equation (6) - see ESL 9.9.2 for details.

35.4 The complete Random Forest algorithms for regression and for classification

Finally we know all the details of random forests - for both regression and classification. A single tree is grown using a heuristic (such as CART) that consists of two stages - growing a full tree, then pruning that tree. A single tree needs three tuning parameters - the maximum number of levels, the minimum number of training samples in a leaf (a box), and the regularization parameter λ. When we run bagging on top of the CART algorithm, with the de-correlation trick (restricting each split to a random subset of coordinates), we get a random forest - either for regression or classification. In a typical random forest implementation there is no pruning - it is too computationally expensive to solve the pruning problem Equation (6) for each tree if we're training hundreds of trees or more.

We've seen trees and forests over three lectures - and finally we know everything we need to know about them.

36 Modern Regression methods on $\mathbb{R}^d$

36.1 Linear Regression with high-dimensional data

We now turn to modern regression methods based on the linear hypothesis class - with regularization. Recall that the first hypothesis class we saw in this course was the hypothesis class of linear functions for regression:

$$\mathcal{H}_{lin} = \left\{ h : \quad h(x_1, \ldots, x_d) = w_0 + \sum_{i=1}^{d} x_i w_i, \quad w_0, w_1, \ldots, w_d \in \mathbb{R} \right\}$$

When we introduced linear regression we assumed that the training sample size m was not smaller than the number of features d: we assumed $m \geqslant d$. We saw that we prefer $m \gg d$, since if $m \sim d$ the variance of the linear hypothesis we find by least squares (the learning algorithm we derived both as ERM for the square loss and using the maximum likelihood principle for Gaussian noisy label) can be large.

But in modern learning problems based on d features (namely, on the sample space $\mathcal{X} = \mathbb{R}^d$) very often d can be quite large. Linear regression was invented when features were measured and recorded manually. In the last few decades it became very easy to collect features and record them automatically, and a typical regression problem can easily have $d \sim m$ or even $d \gg m$.

When $d \sim m$ or, worse, $d \gg m$, we will have correlated features. The coefficients fitted by linear regression will be poorly determined - for example, a large positive coefficient for some feature can be cancelled out by a large negative coefficient for an almost-parallel feature. So the linear regression learner we saw should not be used for $m \sim d$ and cannot be used for $d > m$.

36.2 Best subset selection

Recall that each hypothesis in $\mathcal{H}_{lin}$ corresponds to a unique weight vector $\mathcal{V}w \in \mathbb{R}^d$, where $\mathcal{V}w = (w_1, \ldots, w_d)$, with an intercept $w_0 \in \mathbb{R}$. **Unlike the linear regression class, we won't include w_0 inside the vector $\mathcal{V}w$**, so that $\mathcal{V}w \in \mathbb{R}^d$.

Recall our d-by-m regression matrix X (now we don't have "1" entries in the first row), and vector $\mathcal{V}y$, and recall that we can write

$$L_S(\mathcal{V}w) = \left\| w_0\mathbf{1} + X^\top \mathcal{V}w - \mathcal{V}y \right\|^2$$

for the empirical risk (for square loss), where $\mathbf{1}$ is a vector whose entries are all 1.

The ultimate solution to the issues of linear regression on large d is known as **best subset selection**:

$$\begin{array}{rl} \text{minimize}_{w_0 \in \mathbb{R}, \mathcal{V}w \in \mathbb{R}^d} & \left\| w_0\mathbf{1} + X^\top \mathcal{V}w - \mathcal{V}y \right\|^2 \\ \text{subject to} & \left\| \mathcal{V}w \right\|_0 \leqslant t \end{array}$$

Where $\left\| \mathcal{V}v \right\|_0 = \#\left\{ i \,|\, v_i \neq 0 \right\}$. (Note that the "norm" ℓ_0 is not really a norm (why?).)

This is perfect! We specify t and among all subsets t features out of the d features in our training sample, we find the one with lowest training loss (in the sense of least squares). Now we just play with t to walk on the bias-variance tradeoff - on one hand we would like t to be large enough to have low bias (descriptive power) and one the other hand we would like t to be small enough (e.g. much smaller than m) so we will avoid all the nasty effects of large d mentioned above.
The bad news? You won't be surprised to hear that this problem is known to be NP-hard. Indeed the quantity $\left\| \mathcal{V}w \right\|_0$ is not convex, and solving this smells like a combinatorial search over all possible subsets of size t.

What can we do? Instead of

$$\begin{array}{rl} \text{minimize}_{w_0 \in \mathbb{R}, \mathcal{V}w \in \mathbb{R}^d} & \left\| w_0\mathbf{1} + X^\top \mathcal{V}w - \mathcal{V}y \right\|^2 \\ \text{subject to} & \left\| \mathcal{V}w \right\|_0 \leqslant t \end{array}$$

we will want to consider some other constrain on the coefficient vector $\mathcal{V}w$ that will measure its complexity. For a norm $\|\cdot\|$ on $\mathbb{R}^d$, we'll consider instead

$$\begin{array}{cc} \text{minimize}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d} & \left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 \\ \text{subject to} & \|\mathcal{V}w\| \leqslant t \end{array}$$

It is simpler to consider an **unconstrained** form of these optimization problems - instead of specifying the constrain t, we'll specify a regularization parameter λ and consider

$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d}\left[L_S(w_0,\mathcal{V}w) + \lambda\|\mathcal{V}w\|\right]$$

So this is where regularization comes in.

The following three regression methods all choose a linear hypothesis in $\mathcal{H}_{lin}$ using

$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d}\left[L_S(w_0,\mathcal{V}w) + \lambda\mathcal{R}(\mathcal{V}w)\right]$$

where the fidelity term is just the empirical risk (sum of squares)

$$L_S(w_0,\mathcal{V}w) = \sum_{i=1}^m \left(w_0 + \langle\mathcal{V}w\,,\,\mathcal{V}x_i\rangle - y_i\right)^2 .$$

The regularization term measures the complexity of the linear hypothesis $\mathcal{V}w$.

We will explore three different regularization terms, based on three different norms. For a vector $\mathcal{V}v \in \mathbb{R}^d$, with $\mathcal{V}v = (v_1,\ldots,v_d)$ define the following norms:

- The ℓ_2 norm: $\|\mathcal{V}v\|_2 = \sqrt{\sum_{j=1}^d |v_i|^2}$ (our usual Euclidean norm)
- The ℓ_1 norm: $\|\mathcal{V}v\|_1 = \sum_{j=1}^d |v_i|$
- The ℓ_0 "norm": $\|\mathcal{V}v\|_0 = \#\{i \mid v_i \neq 0\}$. (Again, this is not actually a norm.)

We'll define three corresponding regression learning algorithms that learn $h \in \mathcal{H}_{lin}$:

- The learner
$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d}\left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda\|\mathcal{V}w\|_2^2$$
is called **Ridge Regression** or **ℓ_2-regularized linear regression**.
- The learner
$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d}\left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda\|\mathcal{V}w\|_1$$
is called **Lasso** or **ℓ_1-regularized linear regression**.
- The learner
$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d}\left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda\|\mathcal{V}w\|_0$$
is related to **best subset regression**, and is sometimes called by that name.

(There is an important detail regarding intercept: see below.)

For all three, if $\lambda = 0$ we get our usual linear regression. For all three, as $\lambda \to \infty$ we get $\hat{\mathcal{V}w} \to 0$ so the regression fits an intercept only (a constant function). But they are very very different from one another.

36.3 ℓ_0 regularization: Best subset selection

The first regularization we will explore uses $\mathcal{R}(\mathcal{V}w) = \|\mathcal{V}w\|_0$.

The problem

$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d} \left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda\|\mathcal{V}w\|_0$$

is related (but not equivalent) to the subset selection problem we mentioned above.

Here we have a tradeoff between the sum of squares $L_S(\mathcal{V}w)$ and the number of features used $\|\mathcal{V}w\|_0$. At certain critical values of λ, as λ grows, less and less features will be allowed, until at some large value of λ we are left with $\mathcal{V}w = 0$ so we only have the intercept left. (Note that we are not including the intercept in the regularization term.)

Note that while this problem is NP hard, there are still software packages that solve it for small d. So if you have $d \approx 40$ you may still use best subset selection.

36.4 Ridge

Ridge regression is a nickname for linear regression with the regularization term $\mathcal{R}(\mathcal{V}w) = \|\mathcal{V}w\|_2^2$:

$$\text{argmin}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d} \left\|w_0\mathbf{1} + X^\top\mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda\|\mathcal{V}w\|_2^2$$

Observe that this is a convex optimization problem, indeed a quadratic program.

Ridge typically shrinks shrinks the regression coefficients:

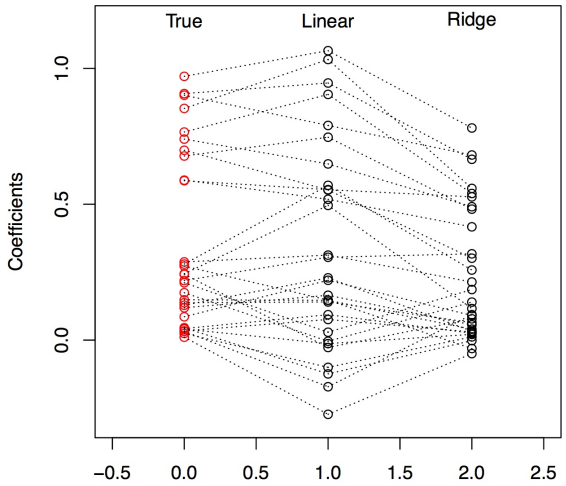


Figure 44: Left to right: Original weights $\mathcal{V}w$ in a linear model $\mathcal{V}y = X^\top\mathcal{V}w + noise$, weights fitted by linear regression ($\lambda = 0$), weights fitted by Ridge ($\lambda > 0$). Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662

Denote by $\hat{\mathcal{V}w}_\lambda^{ridge}$ the minimizer of the ridge problem. For $\lambda = 0$ we get the least squares (standard linear regression solution) and for $\lambda \to \infty$ we get $\hat{\mathcal{V}w}_\lambda^{ridge} \to 0$. As λ increases, bias increases and variance decreases.

How do we do find the Ridge solution computationally? There is actually no need to use a QP solver: ridge is the only case in this course (other than standard linear regression) where we have a **closed-form** solution for the minimizer. Applying the same method we used in linear regression (calculating the gradient with respect to $\mathcal{V}w$ and looking for an extremal point) we find that

$$\frac{\partial}{\partial w_i}\left[\left\|w_0\mathbf{1} + X^\top \mathcal{V}w - \mathcal{V}y\right\|^2 + \lambda \left\|\mathcal{V}w\right\|_2^2\right] = 0 \qquad i = 1, \ldots, d$$

leads to the linear system

$$X\mathcal{V}y = (XX^\top + \lambda I)\mathcal{V}w\,.$$

Observe that for $\lambda = 0$ we recover the normal equations for linear regression. The value of regularization is now clear - even if $XX^\top$ is not invertible, even for $d > m$, for any $\lambda > 0$ we always have that $(XX^\top + \lambda I)$ is invertible.

The SVD method for calculating the linear regression weights generalizes to Ridge regression. Let $X = U\Sigma V^\top$ be an SVD of X, with singular values $\{\sigma_i\}$. In the recitation you will prove that the Ridge solution is given by

$$\mathcal{V}\hat{w}_\lambda^{ridge} = U\Sigma^\lambda V^\top \mathcal{V}y$$

where Σ^λ is diagonal whose (i,i) entry is

$$\frac{\sigma_i}{\sigma_i^2 + \lambda}\,.$$

The regularization parameter λ controls the bias-variance tradeoff. Here's an example:

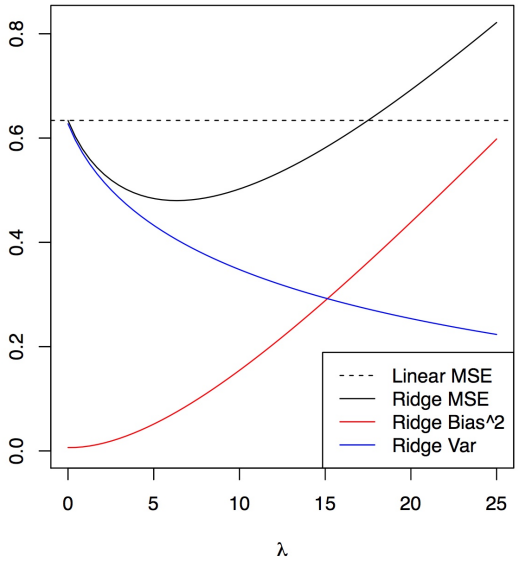


Figure 45: Bias, variance and MSE (mean square error) of a regression problem using Ridge regression, over λ. Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662

Regularization Path. It is interesting to trace each individual weight $(\hat{w})_i$ for $1 \leqslant i \leqslant d$ as λ changes. Such a plot is called the **Regularization Path** of the particular problem at hand. Observe how the weights start from the linear regression weights (for $\lambda = 0$) and follow a decay that looks roughly like $1/\lambda$ decay as λ grows.

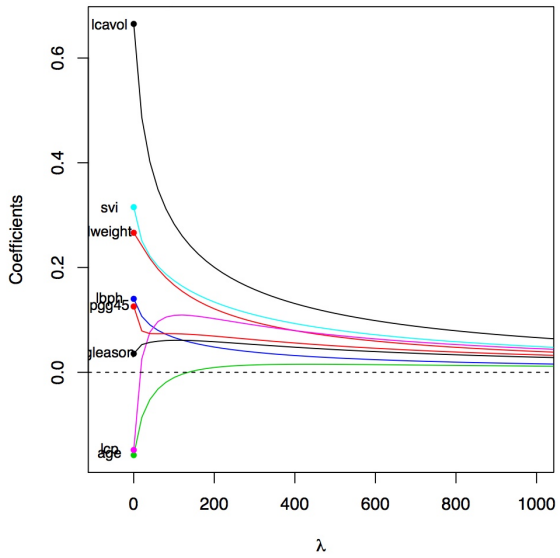


Figure 46: Regularization Path of a Ridge regression, over λ. Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662

36.5 Lasso

So Ridge regression is a good method for using linear regression when d is large or, when $d > m$, or when $XX^\top$ is not invertible. It's also a good way to get a simple handle on the bias-variance tradeoff of linear regression. But is does not do what best-subset can do, namely, **select** a specific subset of the features / variables to use for regression.

The Lasso[22] is a nickname for ℓ_1 regularized linear regression:

$$\text{argmin}_{w_0 \in \mathbb{R}, \mathcal{V}w \in \mathbb{R}^d} \left\| w_0 \mathbf{1} + X^\top \mathcal{V} w - \mathcal{V} y \right\|^2 + \lambda \left\| \mathcal{V} w \right\|_1 .$$

Observe that this is a convex optimization problem, indeed a quadratic program.

This is one of the most effective and well known methods for modern regression.

[22] The name is an acronym for **Least Absolute Shrinkage and Selection Operator**, but you might as well think about the Lasso that cowboys (and Wonder Woman) use.

Figure 47: A Lasso

It has the same behavior as Ridge for $\lambda = 0$ and $\lambda \to \infty$, but in between, it is completely different. Lasso solutions - the weight vectors $\hat{\mathcal{V}w}_\lambda^{lasso}$ produced by Lasso - are **sparse**, meaning that they have few nonzero entries. The larger λ, typically the more zeros $\hat{\mathcal{V}w}_\lambda^{lasso}$ will have - so it gets sparser and sparser until, in the limit $\lambda \to \infty$, it is the zero vector. Let's compare Ridge and Lasso coefficients:

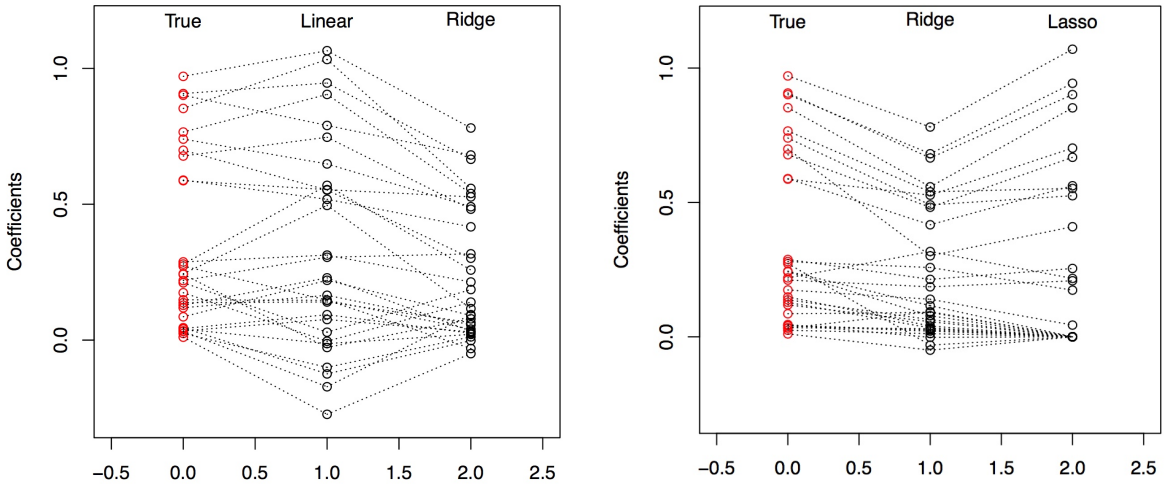


Figure 48: Right: Ridge weights. Left: Lasso weights on the same problem. Note how in Lasso, many coefficients are sent to 0, so that the weight vector is **sparse.** Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662

Let's also compare the Regularization Path or Ridge and Lasso:

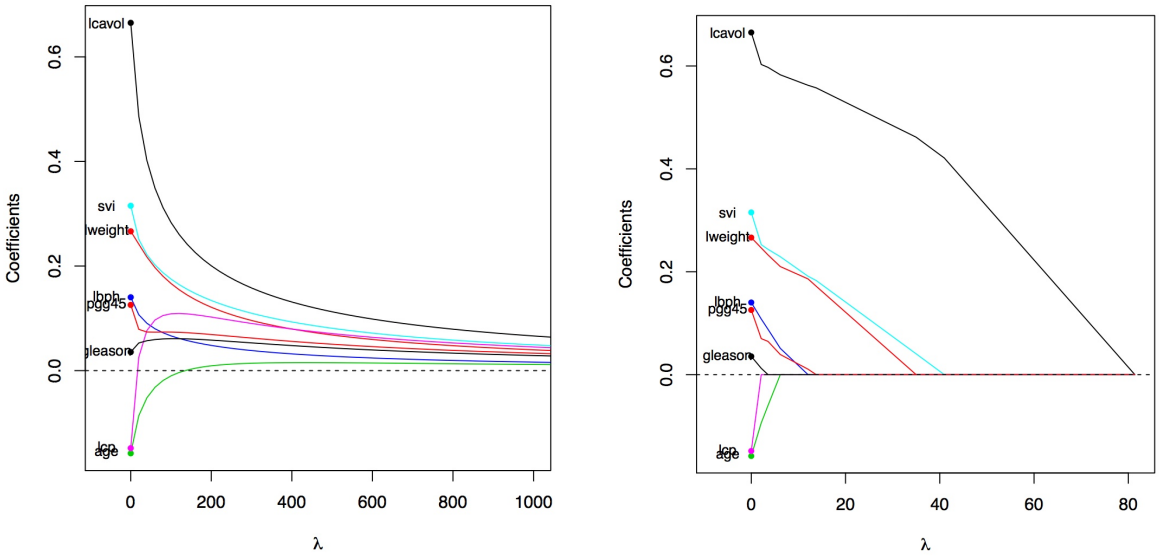


Figure 49: Right: Ridge regularization path. Left: Lasso regularization path on the same problem. Note how in Lasso, many coefficients are sent to 0, so that the weight vector is **sparse.** Source: Ryan Tibshirani Data Mining slides, CMU 36-462/36-662

Computational considerations. The Lasso is a convex optimization problem, indeed a quadratic program. There are specialized solvers for the Lasso, which calculate the entire regularization path (namely, solve the problem for all values of λ at once) at the same computational cost as solving Ridge.

The Lasso is a kind of subset selection. For each value of λ, the Lasso solution has certain weights set to zero and others nonzero. Looking at the "active set" - the features with nonzero weights at the Lasso solution - is a way to know which features / variables are important for the regression. The larger λ, the fewer active variables we will have. Therefore, the Lasso has an important advantage in **interpretability** over Ridge - it selects for us a subset of the variables.

37 The ℓ_1 norm induces sparsity

A natural question that arises now is: how come the Lasso, which uses an ℓ_1 regularization term, gives sparse solutions, while Ridge, which uses an ℓ_2 regularization term, does not?

We will get some insight into this from two different directions.

37.1 Unit balls

For a norm $\|\cdot\|$ on $\mathbb{R}^d$, a ball or radius ρ (around the origin) is the set

$$\left\{\mathcal{V}w \in \mathbb{R}^d \mid \|\mathcal{V}w\| \leqslant \rho\right\}.$$

A ball for the ℓ_2 (Euclidean) norm is what we're used to calling a ball. However, a ball for the ℓ_1 norm is a very different shape.

Exercise. Prove the in $\mathbb{R}^2$, the ℓ_1 unit ball is a square (4 corners). Prove that in $\mathbb{R}^3$, it is a regular octahedron (6 corners).

For general d, in $\mathbb{R}^d$, the ℓ_1 unit ball is a **regular convex polytope** with $2d$ corners.

Now, instead of the Ridge and Lasso problems (which are unconstrained problems) , let us consider - for just a moment- the following optimization problems:

$$\begin{array}{ll} \text{minimize}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d} & \left\|w_0\mathbf{1}+X^\top\mathcal{V}w-\mathcal{V}y\right\|^2 \\ \text{subject to} & \|\mathcal{V}w\|_2^2\leqslant t \end{array}$$

and

$$\begin{array}{ll} \text{minimize}_{w_0\in\mathbb{R},\mathcal{V}w\in\mathbb{R}^d} & \left\|w_0\mathbf{1}+X^\top\mathcal{V}w-\mathcal{V}y\right\|^2 \\ \text{subject to} & \|\mathcal{V}w\|_1\leqslant t \end{array}$$

While we won't go into the details, these constrained problems are equivalent to Ridge and Lasso, in the sense that for each value of λ and solution to the **unconstrained** problem, there is a value of t for the **constrained** problem which has the same solution.

Now, fix a value of t and ask yourself where, in $\mathbb{R}^d$, will we find the solutions $\hat{\mathcal{V}w_t^{ridge}}$ and $\hat{\mathcal{V}w_t^{lasso}}$ to the constrained problem. The fidelity term is a quadratic form in $\mathcal{V}w$, so its level sets are ellipsoids. If t is really large, so there is no constrain effectively, the solution to the minimization problem will be the least squares solution (marked $\hat{\beta}$ in the figure below). As we make t smaller and smaller, we force the solution to be inside the norm ball. By definition, the solution will be found where one of these level sets touches the ball of radius t. It turns out that for ℓ_1 norm, this typically happens at one of the "corners" of the ℓ_1 ball. These corners correspond to **sparse** solutions!

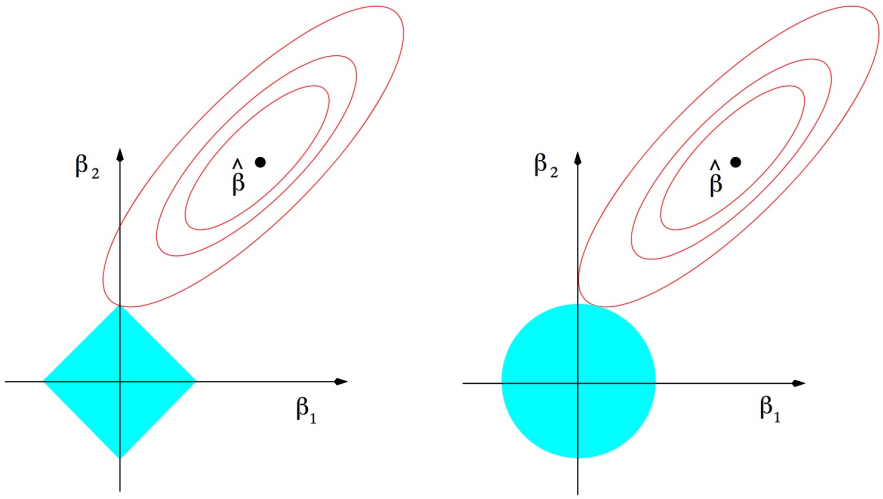


Figure 50: The constrained optimization problems in $d = 2$. Left: Lasso (ℓ_1 ball). Right: Ridge (ℓ_2 ball). Source: ESL

37.2 Orthogonal design

Here is another way to understand why Lasso solutions are sparse. Consider the case where all features are orthogonal, namely $XX^\top = I$. This is in some sense the simplest setup for regression - we write $\mathcal{V}y$ as a linear combination of **orthonormal** vectors. Statisticians call such X an **orthogonal design**.

For $x \in \mathbb{R}$ and $\lambda \geqslant 0$ let us define two important functions $\mathbb{R} \to \mathbb{R}$:

- "soft threshold at λ" is the function $\eta_\lambda^{soft} : \mathbb{R} \to \mathbb{R}$ defined by

$$\eta_\lambda^{soft}(x) = \begin{cases} x - \lambda & x \geqslant \lambda \\ 0 & \lambda > x > -\lambda \\ x + \lambda & -\lambda \geqslant x \end{cases}$$

- "hard threshold at λ" is the function $\eta_\lambda^{hard} : \mathbb{R} \to \mathbb{R}$ defined by

$$\eta_\lambda^{hard}(x) = x \cdot \mathbf{1}_{|x| \geqslant \lambda}$$

Now, for $\lambda \geqslant 0$, let $\hat{\mathcal{V}w}^{LS}$, $\hat{\mathcal{V}w}_\lambda^{ridge}$, $\hat{\mathcal{V}w}_\lambda^{lasso}$, $\hat{\mathcal{V}w}_\lambda^{subset}$ denote the least squares (standard linear regression), Ridge, Lasso and Best Subset solutions, respectively.

Claim. For a vector $\mathcal{V}w \in \mathbb{R}^d$ and a function $\eta : \mathbb{R} \to \mathbb{R}$ write $\eta(\mathcal{V}w) \in \mathbb{R}^d$ for the vector obtained by applying η to each of the coordinates of $\mathcal{V}w$ separately. Assume orthogonal design. Then the Ridge, Lasso and Best subset solutions are given by

- $\hat{\mathcal{V}w}_\lambda^{ridge} = \hat{\mathcal{V}w}^{LS}/(1+\lambda)$
- $\hat{\mathcal{V}w}_\lambda^{lasso} = \eta_\lambda^{soft}(\hat{\mathcal{V}w}^{LS})$
- $\hat{\mathcal{V}w}_\lambda^{subset} = \eta_{\sqrt{\lambda}}^{hard}(\hat{\mathcal{V}w}^{LS})$

(You will prove this in the recitation and homework.)

Namely, in this case, we can obtain the best subset, Lasso and Ridge solutions by applying **univariate shrinkage functions** to each coordinate of $\hat{\mathcal{V}w}^{LS}$ separately.

Now observe (see figure) that in the orthogonal design case, the best subset solution sets some weights to zero and leaves the rest untouched; the Lasso solution sets some weights to zero and "shrinks" the rest a little; and the Ridge solution just multiplies by a scalar. So in the orthogonal design case we see how adding regularization ℓ_0 "norm" and ℓ_1 norm induces sparse solutions, that get more and more sparse as λ grows. A similar phenomenon, yet much more complicated, happens in the general case.

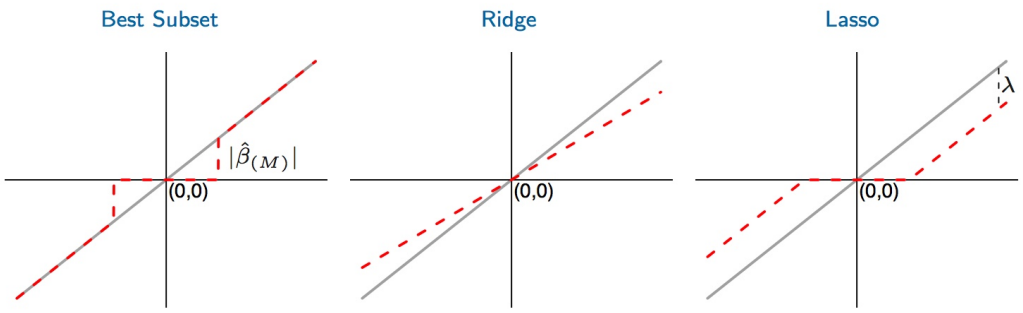


Figure 51: The constrained optimization problems in $d = 2$. Left: Lasso (ℓ_1 ball). Right: Ridge (ℓ_2 ball). Source: ESL

38 ℓ_1-regularized logistic regression

We have seen a few different regression methods using regularization. To remind us that regularization is a general principle, let's have a look at a classification method using regularization.

Recall that in logistic regression, our training sample is organized as regression matrix X and a label vector $\mathcal{V}y \in \{0,1\}^m$.

If $m \sim d$ or even $d > m$ (not to mention the really high dimensional case $d \gg m$), logistic regression will suffer the same problems as linear regression - optimization will be numerically unstable; we can have large coefficients of opposite signs for parallel feature vectors; interpretation will be hard; and so on. All these problems are alleviated by adding a regularization term.

Recall that the logistic regression classifier finds the weight vector by solving

$$\hat{\mathbf{w}} := \underset{\mathbf{w}\in\mathbb{R}^d}{\operatorname{argmax}} \sum_{i=1}^{m} \left[y_i(w_0 + \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle) - \log\left(1 + e^{w_0 + \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle}\right)\right],$$

where (unlike the classification lecture) we did not include the intercept w_0 in the vector $\mathcal{V}w$, so $\mathcal{V}w \in \mathbb{R}^d$.

So the fidelity term, which is minimized in the unregularized problem, is

$$\mathcal{F}_S(\mathcal{V}w) = \sum_{i=1}^{m} \left[\log\left(1 + e^{w_0\langle \mathbf{x_i}\,,\,\mathbf{w}\rangle}\right) - y_i(w_0 + \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle)\right]$$

While ℓ_1-regularized linear regression has a fancy nickname (Lasso), ℓ_1-regularized logistic regression does not have a nickname. This learner simply adds the regularization term $\mathcal{R}(\mathcal{V}w) = \|\mathcal{V}w\|_1$ - just like the Lasso - to the fidelity term.

So the ℓ_1-regularized logistic regression classifier solves

$$\operatorname{argmin}_{\mathcal{V}w\in\mathbb{R}^d} \left[\sum_{i=1}^{m} \left[\log\left(1 + e^{w_0 + \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle}\right) - y_i(w_0 + \langle \mathbf{x_i}\,,\,\mathbf{w}\rangle)\right] + \lambda \|\mathcal{V}w\|_1 \right]$$

This is a convex optimization problem, and fast specialized solvers are available. ℓ_1-regularized logistic regression is a great classifier for $\mathcal{X} = \mathbb{R}^d$ - it has low variance (as it is a linear classifier), we can control the bias-variance tradeoff by choosing λ; it is very (very!) interpretable, and so on.

39 Practical considerations

39.1 Choosing lambda

By now you're probably asking yourself how to choose λ, the regularization parameter, based on the training sample alone. We'll address this extensively in the next lecture.

39.2 Intercept

Notice that we never include the intercept w_0 in the regularization term. One way to simplify everything would be to **center** the rows of X, and then we can disregard the intercept (see ESL exercise 3.5).

39.3 Software implementation

There are various software implementations for CART regression trees and random forests for regression. There are various software implementations for best subset selection (for small d). For Ridge, Lasso and ℓ_1 regularized logistic regression, we mention the package `glmnet` (with bindings to both `python` and `R`, which offers a correct, fast implementation solving the entire regularization path. It also has other goodies such as cross-validation for choosing λ (which we will discuss in the next lecture), a method for plotting the regularization path, combining ℓ_1 and ℓ_2 regularization terms, and more.

Suggested Reading:

- Model selection and model evaluation: ESL 7.2
- Bias-Variance Tradeoff and Decomposition: UML 5.2, ESL 7.3
- k-Fold Cross Validation: ESL 7.10
- The "snooping mistake" in Cross Validation for model evaluation: ESL 7.10.2
- Bootstrap for model evaluation: ESL 7.11
- Feature selection

40 Introduction

40.1 Model Selection

So far in this course we've seen learning algorithms (for classification and regression) and meta-algorithms that can be applied on top of an existing learning algorithm. The next step in our expertise is to be able to know

- Which of the algorithms we have at our disposal to use on a certain problem?
- How to choose the tuning parameters?

This is called **Model Selection.**

What do we mean by the term "tuning parameters"? As we've seen, "choosing an algorithm" typically means choosing a **family** of learning algorithms, namely a collection of learning algorithms indexed by one or more parameters. We can call these tuning parameters. Some examples we've seen:

- choosing k in k-Nearest Neighbors
- choosing the regularization parameter λ in soft-SVM
- choosing the regularization parameter λ in Lasso, Ridge Regression or ℓ_1-regularized logistic regression
- choosing the maximal tree depth and pruning regularization parameter in CART classification / regression trees

Also if we've using a meta-algorithm such as bagging or boosting, we'll need to choose any tuning parameters of the base learning **and also** to choose the number of bagging / boostrap iteration B. If we're using de-correlation in bagging, there might be tuning parameters associated with that (e.g. the number k of allowed coordinates for a split in Random Forest).

So model selection is the choice of model and any tuning parameters. When we are finished with our model selection, we have a completely specified learning algorithm that we can train on our training sample, and use for prediction on new samples.

40.2 Model Evaluation

Another next step in our expertise is to be able to estimate the performance (generalization error) of the model we've chosen - before we actually go ahead and use it on new samples. This is important for a number of reasons:

- One, if our estimate for the generalization error of our selected model is poor, we may be working with the wrong learning algorithm for this problem - maybe it would be best to go back to the model selection stage and come up with new candidates.
- Two, very often when using machine learning to solve real problems, we'll want to know how our chosen learner will perform before we actually start using it. (For example, think about a credit model learning system predicting which customers will default on their loans.)

40.3 What is the generalization error, exactly?

Assume we have a loss function $\ell(\cdot, \cdot)$. The **empirical risk** of an hypothesis $h : \mathcal{X} \to \mathcal{Y}$ is always defined simply by

$$L_S(h) = \sum_{i=1}^{m} \ell(h(x_i), y_i))$$

where $S = \{(x_i, y_i\}_{i=1}^m$ is our training sample.

However, our definition of **generalization error** depends on our data-generation model. We've actually seen several definitions of **generalization error** in this course already:

- **No data-generation model.** If we don't assume anything about how data is generated, the only definition of generalization error is on a new test set, unseen by the learner during training. If we have such a test set $T = \{(x_j, y_j)\}_{j=1}^{|T|}$ then the generalization error of a chosen hypothesis h is simply the average

$$L(h) = \sum_{j=1}^{|T|} \ell(h(x_j), y_j))$$

- **Probability distribution on $\mathcal{X}$, and unknown labeling function $f : \mathcal{X} \to \mathcal{Y}$** If make the same assumption as the PAC model, the generalization error is

$$L_{\mathcal{D},f}(h) = \mathbb{E}_{x\sim\mathcal{D}}[\ell(h(x), f(x)]$$

- **Probability distribution on $\mathcal{X} \times \mathcal{Y}$** If make the same assumption as the Agnostic PAC model, the generalization error is

$$L_{\mathcal{D}}(h) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[\ell(h(x), y]$$

Just be mindful and pay attention about the actual definition being used in different cases.

41 Bias-Variance

It's crucial to understand the behavior of generalization error as we make our hypothesis $\mathcal{H}$ smaller or larger, and as we change the learning algorithm $\mathcal{A}$ that chooses $h_S \in \mathcal{H}$.

41.1 Bias-Variance decomposition for square error loss

In a regression problem, when using the square error loss, the generalization error can be written as the sum of the variance and the square of the bias.

Assume we have a training set $S = (x_i, y_i)$. We obtain a test set $T = (\tilde{x}_j, \tilde{y}_j)$. We form the response vector of the test set, $\tilde{\mathcal{V}}y = (\tilde{y}_1, \ldots, \tilde{y}_{|T|})$. We use a learning algorithm trained on S to obtain h_S, and write $\hat{\tilde{y}}_j = h_S(\tilde{x}_j)$ for the predictions of h_S on the test set T. We have a vector $\hat{\tilde{\mathcal{V}}}y$ of predictions. If we use the square error loss, then the generalization error (as measured on the test T) is simply $\left\|\hat{\tilde{\mathcal{V}}}y - \tilde{\mathcal{V}}y\right\|^2$. Now, since the training sample S is a random set, h_S is a random hypothesis, and therefore $\hat{\tilde{\mathcal{V}}}y$ is a random vector - all with respect to the random choice of S. Let us write $\mathbb{E}$ for the expectation with respect to the random choice of S. (This could be a random choice of $x_1, \ldots x_m$, or a random choice of $(x_1, y_1), \ldots, (x_m, y_m)$, for example.)

We have:

$$\begin{aligned}\mathbb{E}\left\|\tilde{\mathcal{V}}y-\hat{\tilde{\mathcal{V}}}y\right\|^2 &= \mathbb{E}\left\|\tilde{\mathcal{V}}y-\mathbb{E}\hat{\tilde{\mathcal{V}}}y+\mathbb{E}\hat{\tilde{\mathcal{V}}}y-\hat{\tilde{\mathcal{V}}}y\right\|^2 \\ &= \mathbb{E}\left\|\tilde{\mathcal{V}}y-\mathbb{E}\hat{\tilde{\mathcal{V}}}y\right\|^2+\mathbb{E}\left\|\mathbb{E}\hat{\tilde{\mathcal{V}}}y-\hat{\tilde{\mathcal{V}}}y\right\|^2+2\mathbb{E}\left\langle\tilde{\mathcal{V}}y-\mathbb{E}\hat{\tilde{\mathcal{V}}}y\,,\,\mathbb{E}\hat{\tilde{\mathcal{V}}}y-\hat{\tilde{\mathcal{V}}}y\right\rangle \\ &= \left\|\tilde{\mathcal{V}}y-\mathbb{E}\hat{\tilde{\mathcal{V}}}y\right\|^2+\sum_{i=1}^{m}Var(\hat{\tilde{y}}_i)\end{aligned}$$

What did we find?

- The term $\left\|\tilde{\mathcal{V}}y-\mathbb{E}\hat{\tilde{\mathcal{V}}}y\right\|$ is the **bias** of the learner we used. It quantifies the distance between the typical prediction $\mathbb{E}\hat{\tilde{\mathcal{V}}}y$ (typical over the choice of training sample S) and the true label $\tilde{\mathcal{V}}y$.
- The term $\sum_{i=1}^{m}Var(\hat{\tilde{y}}_i)$ is the **variance** of the learner we used. It quantities the variability in the predicted labels, as a result of the randomness in the training sample S.

So we have decomposed the generalization error - the performance of predicting new labels - into the square of the bias, plus the variance.

Here is an example you can do yourself in simulation - in polynomial fitting:

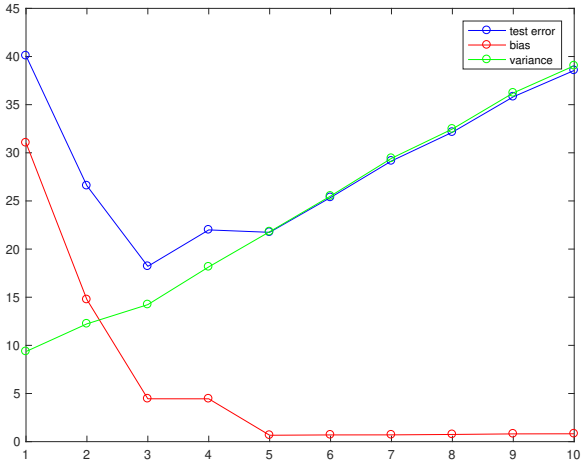


Figure 52: Bias-Variance decomposition in polynomial fitting. Horizontal axis: fitted polynomial degree.

Here are two other examples of the bias-variance decomposition for square loss -

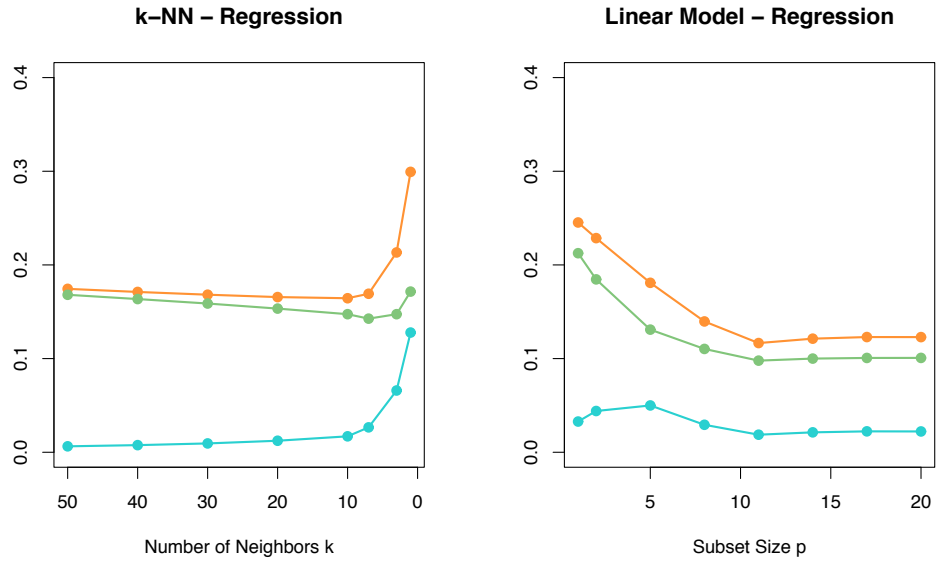


Figure 53: Bias-Variance decomposition in simulation. Left: k-NN regression. Right: Best-subset regression. Horizontal axis - left: k the number of nearest neighbors, right: p the subset size. Orange - generalization error, green - square bias, blue - variance. Source: ESL

There are other bias-variance decompositions (see the textbooks). It is possible to derive bias-variance decompositions for classifiers, but they are not so elegant.

41.2 The bias-variance tradeoff

Even when we don't have an elegant decomposition of the generalization error as a sum of bias and variance, the **bias-variance tradeoff**, which was mentioned every lecture so far in this course, is a general phenomenon. In any way we choose to measure the complexity of the hypothesis class $\mathcal{H}$ (VC-dimension, degree of polynomial, linear dimension of linear model, maximal decision tree depth, etc ...) or the learner $\mathcal{A}$ (regularization parameter, k in k-NN, etc) we always observe a tradeoff of this form:

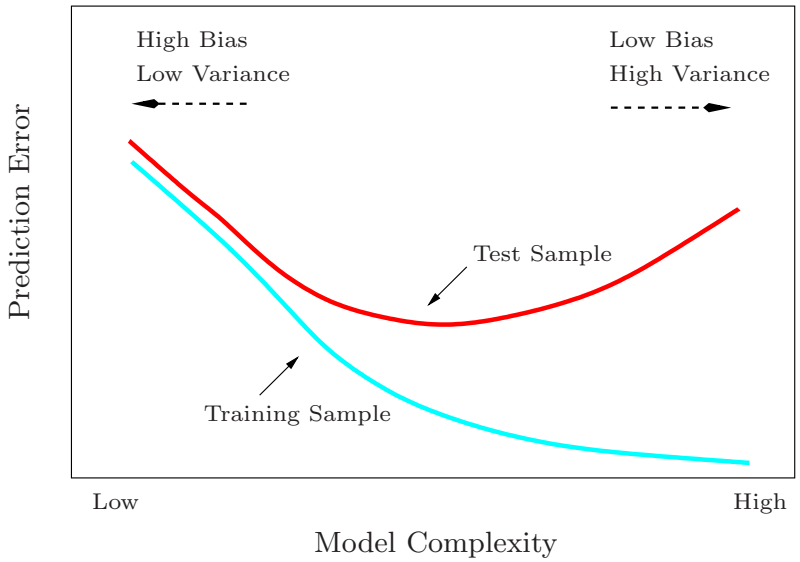


Figure 54: The general phenomenon of Bias-Variance tradeoff over model complexity. Source: ESL

For example, here are bias-variance tradeoffs in classification (here the error is not simply the sum of bias squared and variance, but the phenomenon is still evident: Here are two other examples of the bias-variance decomposition for square loss -

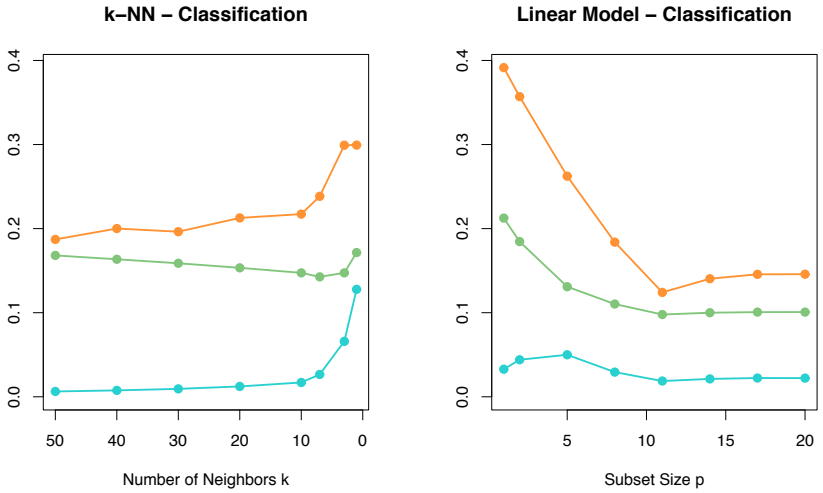


Figure 55: Bias-Variance tradeoff in simulation. Left: k-NN classification. Right: Classification using Best-subset regression. Horizontal axis - left: k the number of nearest neighbors, right: p the subset size. Orange - generalization error, green - square bias, blue - variance. Source: ESL

So when we have control over the model complexity (which is usually the case), the challenge of model selection is to locate the "sweet-spot" where the bias and the variance are not too

high - the point where we get the lowest generalization error.

But how do we do that?

42 Can't naively use training sample for model selection / evaluation

We are in supervised batch learning setting. This means we have a training sample S. The most naive approach for model selection and evaluation is simply to use S for everything. This would look something like that. Given a family of learning algorithms $\{\mathcal{A}_\alpha\}$, do the following:

- **(Training stage.)** Train each of them on S and obtain $h_\alpha = \mathcal{A}_\alpha(S)$
- **(Model selection stage.)** Choose the best one using

$$\alpha^* := \text{argmin}_\alpha L_S(h_\alpha)$$

- **(Model evaluation stage.)** Estimate the generalization error of the chosen trained model h_{α^*} by its empirical risk on S, namely estimate $L_{\mathcal{D}}(h_{\alpha^*})$ by $L_S(h_{\alpha^*})$

As we know already, **this will fail miserably**: the model we select α^* will not have the best generalization, and the estimate $L_S(h_{\alpha^*})$ will be very optimistic for the actual generalization error we will observe. (In fact **optimism** is the technical term for the difference between the empirical risk and the generalization error.)

Why? The learner tries to adapt as much as possible (within the choice of $h \in \mathcal{H}$ allowed to it) to the training data. The more variance the learner has, the more it will adapt to particular properties of the training sample S.

The following figure shows this general phenomenon.

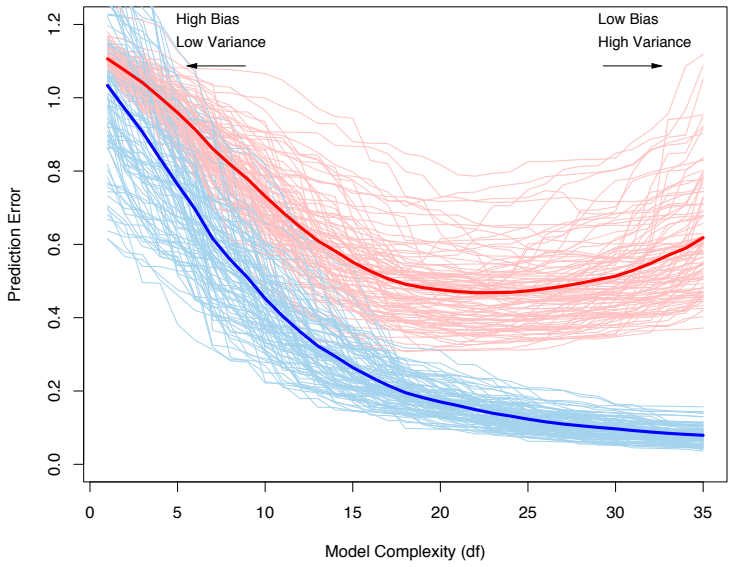


Figure 56: Train error and generalization error over model complexity. Blue is training error (empirical risk), Red is generalization error. Each thin line is a different random sample of training sample (blue) and test sample (red). Source: ESL

43 Model selection and evaluation with unlimited data

So using S naively for everything won't work. Let's go to the other extreme for a moment: If we had unlimited data, how would we approach model selection and model evaluation? We would use three different data sets for the three tasks of training, model selection and model evaluation. Suppose we have three such samples S (training), V (validation), T (test). We would do the following:

- **(Training stage.)** Train each of the models on S and obtain $h_\alpha = \mathcal{A}_\alpha(S)$
- **(Model selection stage.)** Choose the best model using

$$\alpha^* := \text{argmin}_\alpha L_V(h_\alpha)$$

  namely the average loss on the validation set V.

- **(Model evaluation stage.)** Estimate the generalization error of the chosen trained model h_{α^*} by its empirical risk on T, namely estimate $L_{\mathcal{D}}(h_{\alpha^*})$ by $L_T(h_{\alpha^*})$

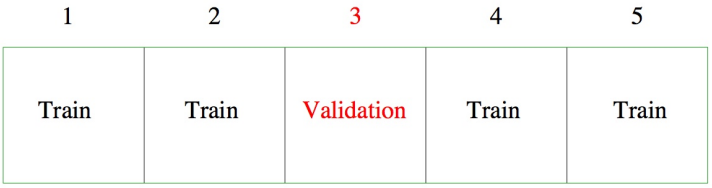
However, in batch learning we are just given our one training sample S. If we split S into a smaller training sample $\tilde{S}$, a validation set V and a test set T, we will be working with much smaller sample sizes.

So what can we do? Is there a way to use S somehow and still do proper model selection and evaluation? We've already seen such magic in the Bagging lecture, where we used our one training sample S as if we had a much larger dataset! We'll do the same here.

44 k-fold Cross Validation

The idea of cross-validation is very simple. Imagine we left out a 1/5 (say) of the training sample S as a held-out set, and trained the model on the remaining 4/5. For a candidate model h, we use the held-out set (that 1/5 of S that we left out) to calculate the prediction error. But hey, why not repeat this same trick five times, each time leaving a different chunk out and training on the rest?

In k-fold Cross Validation we divide the training set S into k equal parts, at random.



- We then proceed as follows. For $i = 1 \cdots k$:
  - Train the i-th model on all the data **except** the i-th fold
  - Calculate the loss of the trained i-th model on the i-th fold, acting as a test set
- Report the estimated mean and standard deviation of the k losses obtained.

(Do you remember how to estimate standard deviation?)

44.1 Cross validation for model selection

Cross validation is the most popular method for choosing tuning parameters. To do this, we train each of the candidate learners $\mathcal{A}_\alpha$ k times - each time leaving out one of the k folds. (So you see this can be pretty computationally intensive.) We then choose the learner $\mathcal{A}_\alpha$ whose average error (over the k test sets) was lowest.

Many software packages (e.g. for regularized learners) offer implementation of cross validation for choosing the regularization parameters.

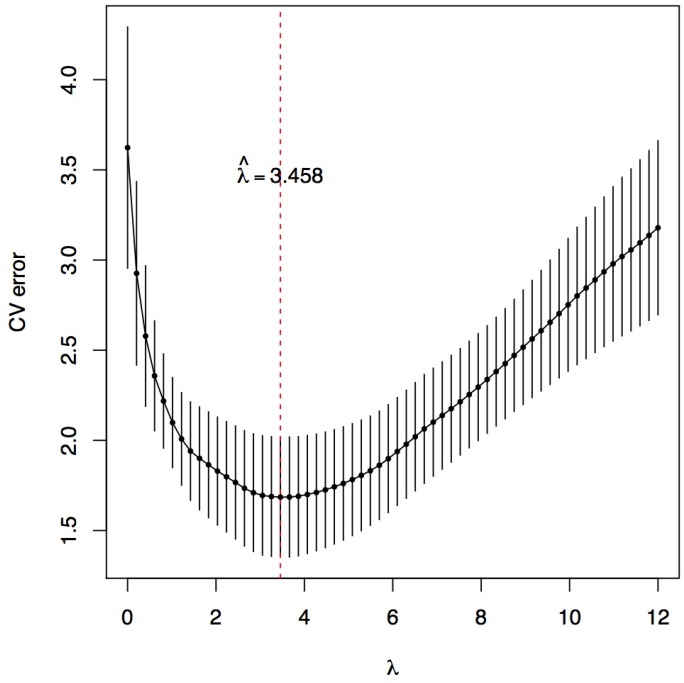


Figure 57: Plot produced by the package `glmnet` for choosing λ for regularized regression by cross validation

Important note. After we finish model selection with cross validation, and have a fully specified learner $\mathcal{A}$, we train $\mathcal{A}$ **again** on the entire training sample S. We would like to train $\mathcal{A}$ on as many points as possible, so for the actual prediction we do not use any of the models trained on parts of S.

44.2 Cross validation for model evaluation

Cross validation is also used for model evaluation. Once we have a final, fully specified learner $\mathcal{A}$, we run k-fold cross validation and report both the average error over the k folds, and its standard deviation. So we estimate the generalization loss and also provide some measure of accuracy of this estimate.

Does it work? Here is a simulation experiment comparing the true generalization loss to the estimate obtained by 10-fold cross validation:

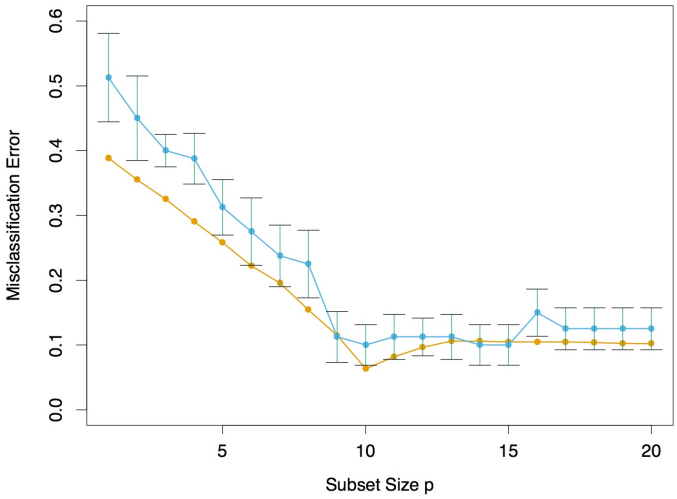


Figure 58: Generalization error (orange) and 10-fold cross-validation error (with errorbars, blue) over model complexity for some learning problem. Source: ESL

44.3 Considerations in choosing number of folds k

- If $k = 1$ then no CV
- If $k = 2$ ("split-sample CV") we only train on half the data, so CV error will be larger than true out-of-sample error
- Small k - we may be training on a dataset too small so again CV error may be **biased upwards**
- Large k - All training sets are almost the same, so we're in CV errors we're averaging highly correlated random variables. May introduce high variance
- if $k = m$ (sample size) this is **leave-one-out** CV
- Since we train the model k times, large k can be computationally intensive

45 Bootstrap

Recall the bootstrap - the basis for the Bagging meta-algorithm.

The Bootstrap can also be used for estimating generalization error - for model selection or evaluation - based on just the one training sample S that we have.

To estimate generalization error of a learner $\mathcal{A}$, we draw B bootstrap samples (each bootstrap sample has m resamples from S with replacements). Call the b-th Bootstrap sample $S^{(b)}$. Define the b-th test set by $T^{(b)} := S \backslash S^{(b)}$ - namely, all the points of S that were **not** chosen in the b-th bootstrap sample. (The samples that were not included are called **out-of-bag** samples.) Now train $\mathcal{A}$ on $S^{(b)}$ and test it on $T^{(b)}$. Report the estimated mean and standard deviation of the generalization errors, as measured on the B test sets.

46 Two common mistakes in model evaluation and how to avoid them

There are two very common mistakes when using cross validation or bootstrap for model evaluation. One mistake causes us to over-estimate the generalization error (so our estimate is too high, too pessimistic). The other causes us to under-estimate the generalization error (so our estimate is too low, too optimistic).

46.1 Over-estimating generalization error

Take for example cross validation, used for model evaluation of a fully specified learner $\mathcal{A}$. In cross validation, we train $\mathcal{A}$ on a training set smaller than S - in fact we train it on $m(k-1)/k$ where $|S| = m$ and k is the number of folds we use in our cross-validation. If k is not large, this means that each time we train during cross validation, we reduce the number of available samples.

Recall from PAC theory (for example) that there is a minimal number of samples needed to learn an hypothesis from a given hypothesis class. What if $|S| = m$ is enough, but $m(k-1)/k$ is not enough? In this case the cross-validation estimate will over-estimate the generalization error.

We realize that it's a good idea to know how the number of training samples affects the learning algorithm we are studying. For every learner there is a curve describing generalization error as a function of training sample size m. If we can somehow estimate this curve, we can be smarter about choosing the number of folds k given the training sample size m that we have.

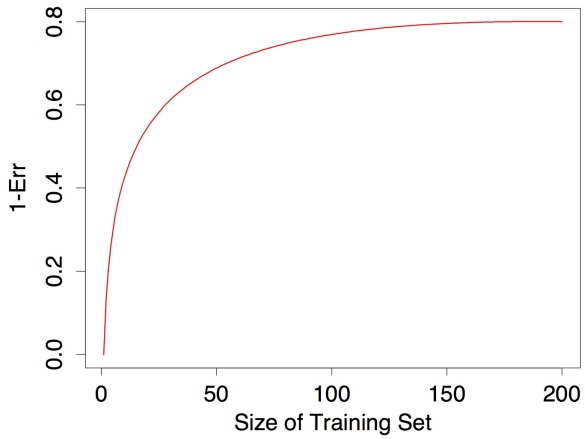


Figure 59: **Exercise**: Suppose this is the generalization error of a learner $\mathcal{A}$ as function of training sample size. We would like to estimate generalization error of $\mathcal{A}$ using k-fold cross validation. Suppose our data has $m = 200$ samples. Will 5-fold CV give a good estimator of out-of-sample error? Now suppose $m = 50$. Has anything changed? Source: ESL

46.2 Under-estimating generalization error

A much more common problem is that cross-validation estimates for generalization error are too optimistic. This is a subtle yet very important point for you to understand.

The following is a very common process:

- Suppose we have a dataset with m samples and cannot get more data
- We start trying out models, each with its own tuning parameters. This can be called **model snooping**
- We change the training sample S - remove features, create new features, remove "problematic" points, etc. This can be called **data snooping.**
- Eventually we find a tuned model and a "clean training sample" that "work well" ...
- ... namely has low training error, fits the data well, etc
- **Now** we use cross validation or bootstrap to estimate the generalization error of our chosen learner.

What's the problem?

There are two problems here:

- One, the problem with model snooping. By obsessively tuning and tuning and trying out many learners, we overfit to the training sample. And if we use a validation set, after using it many many times, we overfit to the validation set as well.
- Two, the problem with data snooping. When new, unseen data comes in, it did not receive this royal treatment we gave to the training sample S. So the cross validation procedure under-estimates performance on new data.

You should be familiar with the **mechanism of self-deception** in machine-learning. It works as follows:

- You start working on a dataset and really want "good results"
- You try many methods, many tuning parameters
- You make some changes in the training sample S
- Gradually as you overfit you sink into self-deception
- This is called "Torture the data until they confess"

46.3 What can we do to avoid under-estimating generalization error?

1. Avoid manual data snooping. **Code the entire pre-processing stage**. At each iteration of Bootstrap or cross-validation, run the entire pre-processing step - just like it would run when you predict on new data.

2. Limit model snooping and data snooping to a small subset of S which you know will be "contaminated with optimism". If keeping a held-out set for use in a later stage of the learner design, lock it and guard the key with your life. Remember you will be tempted to peek at this set, and resist.

Lecture 8: Unsupervised Learning

Suggested Reading:

- Principle Component Analysis (PCA): UML 23.1, ESL 14.5
- Clustering ESL 14.3
- k-means clustering: UML 22.2, ESL 14.3.6

47 Unsupervised learning: Introduction

So far in his course we worked on **supervised batch learning**, namely the case where we had a sample space $\mathcal{X}$ and a label / response space $\mathcal{Y}$. We were interested in **prediction**: how to predict the label $y \in \mathcal{Y}$ corresponding to a new, unseen sample $x \in \mathcal{X}$, based on a training set of labeled samples $\{(x_i, y_i)\}_{i=1}^m$.

However, there are many other kinds of learning problems in the world. Today for the first time in this course we will look at problems with a different setup - problems which do not fall into the framework of supervised batch learning. We'll look at problems where there is no label y at all! The training data will consist of points $\{x_i\}_{i=1}^m$ with $x_i \in \mathcal{X}$. Learning problems with such data are called **unsupervised** learning problems.

If you were looking at supervised problems for too long, you may be wondering what can be done with unlabeled data. Here are three examples:

1. **Uncovering low-dimensional structure**

   Sometimes data in $\mathbb{R}^d$ with d large only **appears** to be high-dimensional. Consider a dataset of images of the same person looking at different directions. If each image is 1024-by-1024 pixels (say), then each image lives in $\mathbb{R}^{1024\times 1024}$. (In fact even more if the image has color.) But the only degrees of freedom in the dataset is the direction at which t person is looking. So we can imagine that this dataset occupies just a very simple subset of $\mathbb{R}^{1024\times 1024}$ - in fact a two-dimensional surface.

Figure 60: Head tracking data. (Source: visagetechnologies.com)

As another example, consider handwritten digits. There are some very famous datasets in machine learning of hand-written digits, scanned from zipcodes people wrote on envelopes. Here again each image may be 28-by-28 pixels, but the data does not occupy the entire space $\mathbb{R}^{28\times 28}$. Instead, there are very few variations on how people write digits, and if we allow rotation, translation and dilation of the digit we can describe each digit using just a few real numbers.

Figure 61: Handwritten digits. (Source: Liu et al, Handwritten digit recognition, Pattern Recognition 37(2) 2004)

It would be nice if we could somehow organize the data points in a space that's smaller and simpler than $\mathbb{R}^{28\times 28}$. For example, it would be nice if we could have something like this:

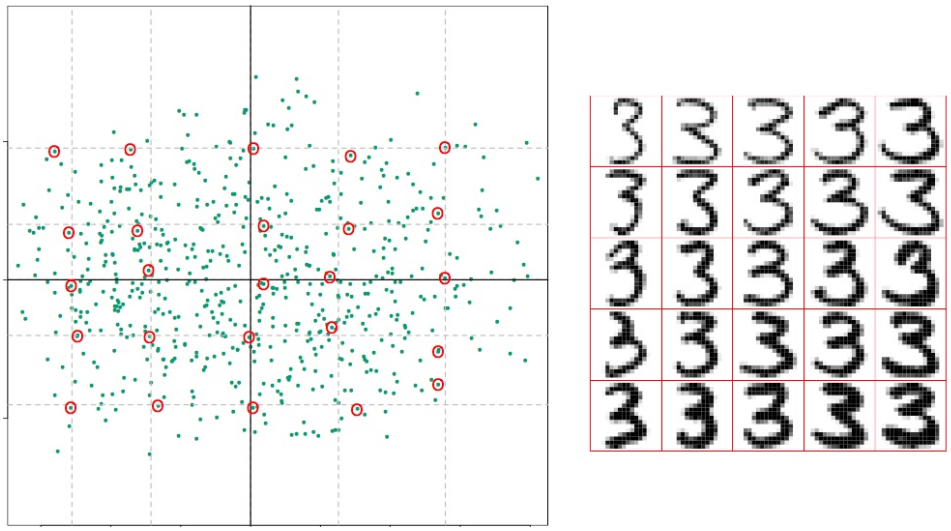


Figure 62: Digit 3 in a two-dimensional space. Green dots correspond to images of handwritten "3" in the dataset. Red circles correspond to images shown. (Source: ESL)

Here, the handwritten digits "3" are all organized on a two-dimensional space, meaning that each is determined by just two real parameters, and we understand how changing the parameter moves us across the dataset.

Such a task is called **dimension reduction.** We map each point $\mathcal{V}x_i$, which lives in a high-dimensional Euclidean space $\mathbb{R}^d$, to a point $W(\mathcal{V}x_i)$, which lives in a low-dimensional Euclidean space. We use the low-dimensional representation for understanding the structure of the dataset; for visualization; for preprocessing; and more.

2. **Clustering.**

   Suppose now that we obtain a dataset of handwritten digits and want to know simply how many different digits are in there, and which images represent the same digit. Or suppose we obtain data of a social network and want to know how many communities are in there, and the members of each community. We don't have labels to help us, and we don't know in advance how many digits (or communities) we will find. Such a task is called **clustering.**

   Another example: We obtain images of faces, and don't know how many different persons there are in the dataset. We would like to know how many different persons there are, and group together images that belong to the same person.

Figure 63: Faces of four people from the Yale face dataset

It would be nice if we could group the images together and count how many different people were photographed.

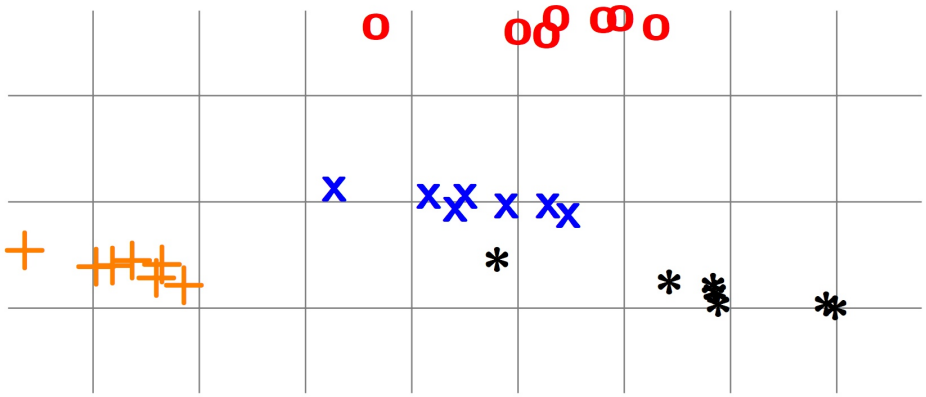


Figure 64: Faces above - dimension reduced to 2, and clustered. Each of the four people is marked with a different marker. (Source: UML)

So **clustering** means dividing the unlabeled dataset $\{\mathcal{V}x_i\}_{i=1}^m$ into k clusters, where each cluster has something in common (for example, each cluster is the same digit, each cluster is the same person, etc) **without using any labels**.

3. **Anomaly detection.**

   Another unsupervised learning task has to do with detecting when a system is not behaving "as usual". Consider an air conditioning system or even a power plant. We place sensors in the system and would like to get a warning when something is behaving "strange". A strange behavior could come from a mechanical malfunction, a software problem, or even a cyber-attack - we don't know. So we monitor the state of the system all the time. If we have d sensors installed, each time we take a reading of the system we get a sample $\mathcal{V}x_i \in \mathbb{R}^d$. We train our learning systems on readings $\mathcal{V}x_1, \ldots, \mathcal{V}x_m \in \mathbb{R}^d$ where we believe everything was normal. Now a new reading comes a long, $\mathcal{V}x \in \mathbb{R}^d$. Is it normal? or is there something wrong? we are required to make a decision without using any labeled data - namely, without having seen the system in a "wrong", "abnormal" or "strange" state before.

47.1 This lecture

As a brief introduction to unsupervised learning, we will see

- The simplest, most popular method for dimension reduction: **Principal Component Analysis** (PCA)
- The simplest, most popular method for clustering: **k-means clustering.**

48 Dimension reduction

Let $\mathcal{V}x_1, \ldots, \mathcal{V}x_m \in \mathbb{R}^d$ be our dataset. In dimension reduction we are looking for a map $W : \mathbb{R}^d \to \mathbb{R}^k$ (with $k < d$) such that anything we would like to do with $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ we can

do, in some sense, with $W(\mathcal{V}x_1), \ldots, W(\mathcal{V}x_m)$. When the map W is a linear map, this is called **linear dimension reduction.** When W is non-linear, this is called **nonlinear dimension reduction.** In this lecture we'll only think about linear dimension reduction.

There are many reasons to study dimension reduction:

- **Learning:** As we have seen, some learning algorithms on $\mathbb{R}^d$ work better when the dimension d is small compared with the training sample size m, and some fail if d is too large.
- **Visualization:** If we can reduce dimension to $k \leqslant 4$ then we can plot the data on a page (possibly on a 3-dimensional axis. The forth dimension can be represented by color).
- **Computation:** As we have seen, the time and space complexity of many learning algorithms grows with the dimension d. By reducing dimension as part of our preprocessing, we can use fewer computational resources for the same task.

48.1 Linear dimension reduction

Suppose that our dataset is $\mathcal{V}x_1, \ldots, \mathcal{V}x_m \in \mathbb{R}^d$ where d is large. But suppose that our data do not actually occupy the entire space $\mathbb{R}^d$. Suppose instead that they live on a k-dimensional linear subspace of $\mathbb{R}^d$, or very close to a k-dimensional linear subspace:

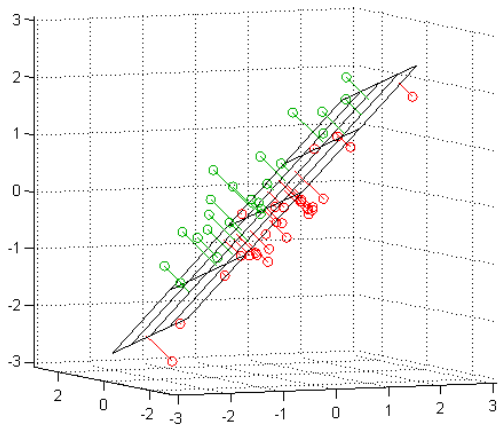


Figure 65: Data close to a linear subspace of R^d

It would make sense to somehow project the data to this subspace and choose the map $W : \mathbb{R}^d \to \mathbb{R}^k$ accordingly. But there are two problems:

1. We don't know this subspace. We just have the data. We need to find a basis for this subspace.
2. Even if we knew the subspace, simply projecting a point $\mathcal{V}x \in \mathbb{R}^d$ onto this subspace does not reduce dimension: the projected point is still in $\mathbb{R}^d$. What we actually want is to find an orthonormal basis for this subspace, and then find the k coordinates of projected point according to the subspace. This would result in k real numbers, so would mean we indeed reduced dimension to k.

So we see that the problem of linear dimension reduction means finding an orthonormal basis for the subspace that best approximates the training data. Can this be done?

48.2 Principal Components Analysis (PCA)

PCA is the most well-known **linear** dimension reduction method. We choose k (the reduced dimension) and look for a linear map $W : \mathbb{R}^d \to \mathbb{R}^k$. How shall we choose W? With W we will also look for an "inverse" map $U : \mathbb{R}^k \to \mathbb{R}^d$, mapping the reduced points back to the original space $\mathbb{R}^d$. Let's measure the error incurred by the whole process using sum of squares

$$\sum_{i=1}^{m} \|\mathcal{V}x_i - UW\mathcal{V}x_i\|^2$$

which we seek to minimize.

The PCA problem is therefore: Given points $\mathbf{x}_1, \ldots, \mathbf{x}_m \in \mathbb{R}^d$, find

$$\operatorname*{argmin}_{W \in \mathbb{R}^{k,d}, U \in \mathbb{R}^{d,k}} \sum_{i=1}^{m} \|\mathbf{x}_i - UW\mathbf{x}_i\|^2 .$$

Theorem. (The solution to the PCA problem.) Let $A = \sum_{i=1}^{m} \mathbf{x}_i \mathbf{x}_i^\top$ and let $\mathbf{u}_1, \ldots, \mathbf{u}_n$ be the n leading eigenvectors of A. Then, the solution to the PCA problem is as follows. Let U be the m-by-k matrix whose columns are $\mathbf{u}_1, \ldots, \mathbf{u}_k$ and let $W = U^\top$. Then U and W solve the PCA problem for $\mathbf{x}_1, \ldots, \mathbf{x}_m$.
Proof.

- UW is a d-by-d matrix of rank k, therefore its image space is subspace of $\mathbb{R}^d$ whose dimension is at most k. Let us denote this subspace by $S = Im(UW)$.
- The transformation $\mathbf{x} \mapsto UW\mathbf{x}$ moves $\mathbf{x}$ to the subspace S.
- Thefore the quantity $\|\mathcal{V}x_i - UW\mathcal{V}x_i\|^2$ is minimized if and only if we choose U and W such that UW is an orthogonal projection on $S = Im(UW)$.
- The point in S which is closest to $\mathbf{x}$, namely the orthogonal projection of x on S, is given by $VV^\top\mathbf{x}$, where the columns of V span an orthonormal basis of S.
- Therefore, we can assume w.l.o.g. that $W = U^\top$ and that the columns of U are orthonormal.
- Now observe that

$$\begin{aligned} \|\mathbf{x} - UU^\top\mathbf{x}\|^2 &= \|\mathbf{x}\|^2 - 2\mathbf{x}^\top UU^\top\mathbf{x} + \mathbf{x}^\top UU^\top UU^\top\mathbf{x} \\ &= \|\mathbf{x}\|^2 - \mathbf{x}^\top UU^\top\mathbf{x} \\ &= \|\mathbf{x}\|^2 - \operatorname{trace}(U^\top\mathbf{x}\mathbf{x}^\top U) , \end{aligned}$$

- Therefore, an equivalent problem to the PCA problem is

$$\operatorname*{argmax}_{U \in \mathbb{R}^{d,k} : U^\top U = I} \operatorname{trace}\left(U^\top \left(\sum_{i=1}^{m} \mathbf{x}_i \mathbf{x}_i^\top \right) U \right) .$$

- Denote $A = \sum_{i=1}^{m} \mathbf{x}_i \mathbf{x}_i^\top$. Then we need to solve

$$\underset{U \in \mathbb{R}^{d,k}: U^\top U = I}{\operatorname{argmax}} \operatorname{trace}\left(U^\top \cdot A \cdot U\right) .$$

- A is symmetric and in fact positive semidefinite. Now let $A = VDV^\top$ be the spectral decomposition of A, where the diagonal of D are the eigenvalues of A in decreasing order and the columns of V are the corresponding eigenvectors.

- Now comes the following claim: for any d-by-k orthonormal matrix U, we have

$$\operatorname{trace}\left(U^\top \cdot A \cdot U\right) \leqslant \sum_{i=1}^{k} D_{i,i} .$$

  (See proof in UML page 325). Observe that the bound on the right hand side is just the sum of the k leading eigenvalues of A.

- But if we take U to be $\tilde{U}$, the m-by-d matrix whose columns are the k leading eigenvectors of A, then U is orthonormal and

$$\operatorname{trace}\left(\tilde{U}^\top \cdot A \cdot \tilde{U}\right) = \sum_{i=1}^{k} D_{i,i} .$$

■

48.3 PCA as Variance Maximization

Actually, there is no reason why the linear subspace, on which (or close to which) the data points are located in $\mathbb{R}^d$, should go through the origin. In other words, we would like to allow the map W to be **affine** (meaning a linear transformation plus a constant, an intercept) - not just linear. So let's generalize the above and consider $W : \mathbb{R}^d \to \mathbb{R}^k$ to be of the form $W(\mathcal{V}x) = \tilde{W}(\mathcal{V}x - \mu)$ where $\mu \in \mathbb{R}^d$ and $\tilde{W} : \mathbb{R}^d \to \mathbb{R}^k$ a linear map. This allows us to "shift" the data before applying the linear map $\tilde{W}$. It turns out that if we add an optimization also over μ to the PCA problem, then the best μ is given by

$$\overline{\mathcal{V}x} = \frac{1}{m} \sum_{i=1}^{d} \mathcal{V}x_i$$

(see ESL 14.5.1). Note that this is just the empirical mean of the data.

Making this change, we see that the matrix A above is now defined as

$$A = \sum_{i=1}^{m} (\mathbf{x}_i - \overline{\mathcal{V}x})(\mathbf{x}_i - \overline{\mathcal{V}x})^\top$$

instead of $A = \sum_{i=1}^{m} \mathbf{x}_i \mathbf{x}_i^\top$. We recognize this matrix as the **sample covariance**! We thus become curious if there is some connection between PCA and variance / covariance of the data.

Not surprisingly, there is. In the recitation you'll see how PCA can also be derived by looking for a set of orthogonal directions along which the variance of the data is maximized. This is a different derivation leading to the same algorithm we have found (namely, diagonalizing the empirical covariance matrix and projecting the data on its k leading eigenvectors). In that derivation we effectively decompose the empirical covariance matrix A to its eigenvector, where each eigenvector represents a different component of the covariance.

48.4 PCA - formal definition.

So let's write a formal definition of the Principal Components Analysis. With some minor missing steps, we have proved:

Theorem: Suppose we decide to reduce from dimension d to dimension k using a linear dimension reduction, and measure the quality of dimension reduction with sum of squares. Then

$$\text{argmin}_{U,W} \sum_{i=1}^{m} \|\mathcal{V}x_i - UW\mathcal{V}x_i\|^2$$

over matrices $U \in \mathbb{R}^{d\times k}$ and $W \in \mathbb{R}^{k\times d}$ is achieved when the columns of U are the first k eigenvectors of the sample covariance matrix

$$S = \frac{1}{m}\sum_{i=1}^{m}(\mathcal{V}x_i - \overline{\mathcal{V}x})(\mathcal{V}x_i - \overline{\mathcal{V}x})^\top$$

and $W = U^\top$.

Definition: Let be the sample covariance matrix of the training data $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ as above. Let $\mathcal{V}u_1, \ldots, \mathcal{V}u_d$ be the eigenvectors of S corresponding to eigenvalues $\lambda_1, \ldots, \lambda_d$, ordered such that $\lambda_1 \geqslant \lambda_2 \geqslant \ldots \geqslant \lambda_d \geqslant 0$.

- The number λ_i is called the **i-th Principal Value** of $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$.
- The vector $\mathcal{V}u_i$ is called the **i-th Principal Vector** of $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$.

Since the empirical covariance S is a d-by-d positive semidefinite matrix, its eigenvectors $\mathcal{V}v_1, \ldots, \mathcal{V}v_d$ (which we called the Principal Components) form an orthonormal basis for $\mathbb{R}^d$.

Definition: Fix k. The PCA dimension reduction to k dimensions maps each point $\mathcal{V}x_i$ to $U^\top\mathcal{V}x_i$, where the columns of U are the k leading principal vectors $\mathcal{V}v_1, \ldots, \mathcal{V}v_k$. **Note that $U^\top\mathcal{V}x_i$ is just the vector of k coordinates of $\mathcal{V}x_i$ according to $\mathcal{V}v_1, \ldots, \mathcal{V}v_k$,** namely, the coordinates of the projection of $\mathcal{V}x_i$ onto the optimal subspace, which is given by $Span\{\mathcal{V}v_1, \ldots, \mathcal{V}v_k\}$.

48.5 Applying PCA dimension reduction to an arbitrary vector in $\mathbb{R}^d$

As $\mathcal{V}v_1, \ldots, \mathcal{V}v_d$ (the principal vectors) are an orthonormal basis for $\mathbb{R}^d$, we have the following:

1. Any data point $\mathcal{V}x \in \mathbb{R}^d$ can be written as a linear combination of $\mathcal{V}v_1, \ldots, \mathcal{V}v_d$.
2. Let $\mathcal{V}x = \sum_{i=1}^{d}\langle\mathcal{V}x, \mathcal{V}v_i\rangle\mathcal{V}v_i$ be the unique decomposition of some point $\mathcal{V}x \in \mathbb{R}^d$ - not necessarily in our training dataset - on the principal components. We've seen that for each $k < d$, the best linear dimension reduction of $\mathcal{V}x_i$ $(i = 1, \ldots, m)$ in the least squares sense above, is given by the column vector $(\langle\mathcal{V}x, \mathcal{V}v_1\rangle, \ldots, \langle\mathcal{V}x, \mathcal{V}v_k\rangle)^\top$. We can use this map on any other vector $\mathcal{V}x$ and reduce its dimension with the same linear map, mapping it to the column vector $(\langle\mathcal{V}x, \mathcal{V}v_1\rangle, \ldots, \langle\mathcal{V}x, \mathcal{V}v_k\rangle)^\top \in \mathbb{R}^k$. So while the subspace we project on corresponds to the training sample $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$, and the linear dimension reduction is optimal for these points, we can use it for any point in $\mathbb{R}^d$.

48.6 The subtle difference between the projected points and their coordinates.

It is very important to understand the difference between the projection on the span of the first k principal vectors (projection on the optimal subspace) and the coordinates of that projection according to the principal vectors.

Suppose for a moment that $d = 3$ and $k = 2$. When we run PCA on $\mathcal{V}x_1, \ldots, \mathcal{V}x_m \in \mathbb{R}^3$, we find a two-dimensional subspace and project each $\mathcal{V}x_i$ on that subspace. The best subspace is the one that minimizes the sum of squared distances between each $\mathcal{V}x_i$ and its projection. As we just proved, it is spanned by the two leading eigenvectors of the 3-by-3 matrix $S = \frac{1}{m}\sum_{i=1}^{m}(\mathcal{V}x_i - \overline{\mathcal{V}x})(\mathcal{V}x_i - \overline{\mathcal{V}x})^\top$.

Now, we are not just interested in projecting the points in $\mathbb{R}^3$ on the 2-dimensional subspace spanned by these two leading eigenvectors. We would like to actually reduce dimension, namely, find the map $W : \mathbb{R}^3 \to \mathbb{R}^2$ and work with the dimension-reduced dataset $W(\mathcal{V}x_1), \ldots, W(\mathcal{V}x_m)$. As we proved above, in fact $W = U^\top$ where the columns of U are the two leading eigenvectors of A (taken to be orthonormal, since A is positive semidefinite). So the dimension-reduced version of the point $\mathcal{V}x_i \in \mathbb{R}^d$ is just $U^\top \mathcal{V}x_i$ - the **coordinates** of the vector $\mathcal{V}x_i$ according to the orthonormal set of k leading eignevectors of A.

If $\mathcal{V}v_1, \mathcal{V}v_2, \mathcal{V}v_3$ are all three principal components (so that the two leading ones are $\mathcal{V}v_1$ and $\mathcal{V}v_2$) then a vector $\mathcal{V}x_i$ in our data decomposes as $\mathcal{V}x = \sum_{i=1}^{3}\langle \mathcal{V}x, \mathcal{V}v_i\rangle \mathcal{V}v_i$. The **projection** of $\mathcal{V}x_i$ on the optimal two-dimensional subspace is given by $\mathcal{V}x = \sum_{i=1}^{2}\langle \mathcal{V}x, \mathcal{V}v_i\rangle \mathcal{V}v_i$, while the **coordinates** of $\mathcal{V}x_i$ according to the two leading principal vectors are given by $(\langle \mathcal{V}x, \mathcal{V}v_1\rangle, \langle \mathcal{V}x, \mathcal{V}v_2\rangle)$. In matrix form, these coordinates are given by $U^\top \mathcal{V}x_i$ where the first column of U is $\mathcal{V}v_1$ and the second column of U is $\mathcal{V}v_2$.

The difference between the projected point (which is still a point in R^3) and the two coordinates according to the orthonormal basis spanning the subspace is made clear in the following image:

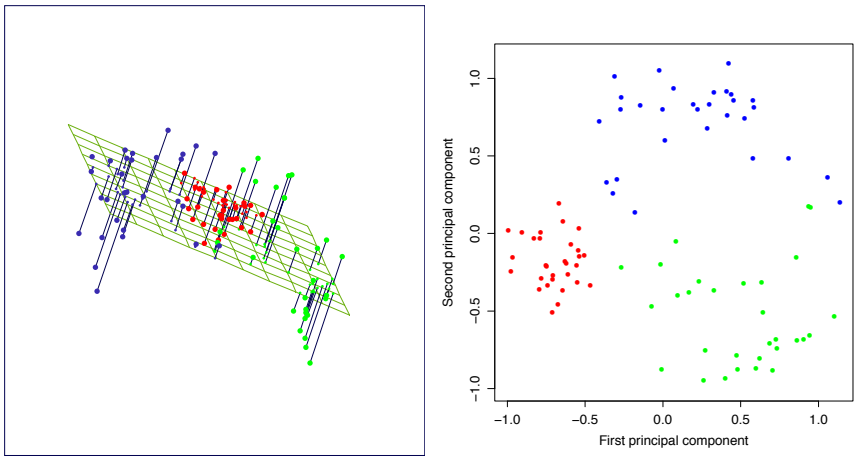


Figure 66: Left: points $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ are projected on the subspace spanned by the two leading eigenvectors of A. Right: coordinates of the projected points according to these two orthonormal vectors. (Source: ESL)

If you don't fully grasp the difference between the left panel and the right panel, PCA may remain a mystery to you. Spend time making sure you understand the difference.

(Recall our lecture on linear regression, where we first projected a vector on a subspace, and then looked for the coordinates of the projected point according to a set of vectors spanning the subspace. There, the spanning set was usually not orthonormal. Here, it is always orthonormal.)

Here is another example for $d = 3$ and $k = 2$. The dataset (left panel) is a point cloud in $\mathbb{R}^3$ shaped like a bagel. In the dimension-reduced dataset in $\mathbb{R}^2$ we replace each point with the **coordinates** of its projection on the two-dimensional "principal subspace" $Span\{\mathcal{V}v_1, \mathcal{V}v_2\}$.

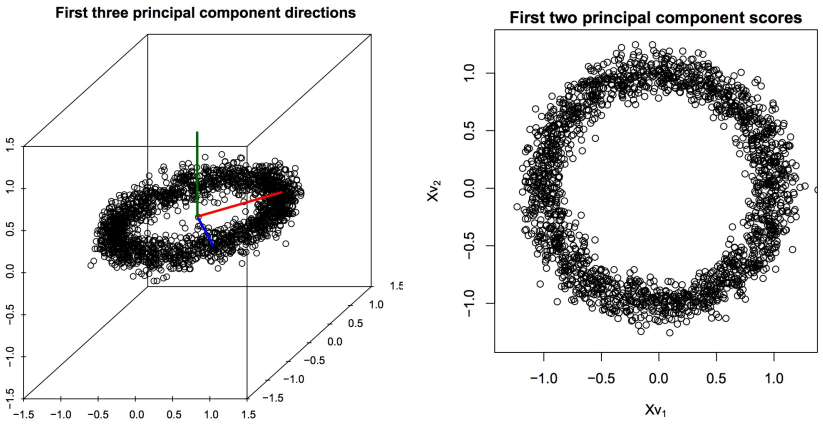


Figure 67: (Source: CMU 36-462/36-662)

In contrast, look at the figure below. We can look at the **projection** of the bagel dataset on the principal subspace for $k = 1$ (left panel), $k = 2$ (middle panel) and $k = 3$ (right panel). All these sets are point clouds in $\mathbb{R}^3$. There is not dimension reduction here. Of course, for $k = 3$ we project on the whole space, so we get the original dataset.

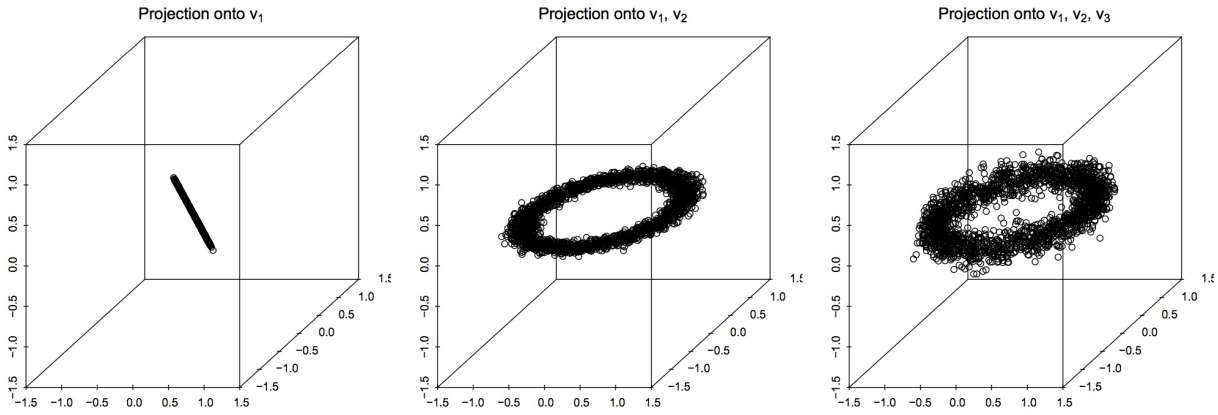


Figure 68: (Source: CMU 36-462/36-662)

Now, let's generalize everything we've said to general d and some given $k < d$. The dimension-reduced dataset $\left\{U^\top \mathcal{V} x_1, \ldots, U^\top \mathcal{V} x_m\right\} \subset \mathbb{R}^k$ replaces each point $\mathcal{V} x_i$ with its **coordinates**. But we can still look at the **projected** points - there are still points in the original dimension $\mathbb{R}^d$. For example, each of these images is greyscale of resolution 50-by-50 pixels. So we can represent each image as a vector in $\mathbb{R}^{50 \times 50} = \mathbb{R}^{2500}$

Figure 69: Faces of four people from the Yale face dataset

Now we run PCA with $k = 10$. In the dimension-reduced version each image is represented by a vector in $\mathbb{R}^{10}$. These are the **coordinates.** We can't plot them as an image. But if we look at the **projections**, which are still vectors in $\mathbb{R}^{2500}$ (that live on a 10-dimensional subspace in $\mathbb{R}^{2500}$) we can plot them as images:

Figure 70: Faces above projected to 10 dimensional subspace found by PCA. Source: UML

This is a simple form of **compression**: to store the dimension-reduces images, we need to store just 10 principal vectors (each in $\mathbb{R}^{2500}$), and m vectors in $\mathbb{R}^{10}$.

In contrast, if we were to look at the **coordinates** in $\mathbb{R}^{10}$ we can't plot them as images, but we can still hope that the dimension-reduced points of images of same people stay together in $\mathbb{R}^{10}$. Here is a plot of the first two coordinates of each image in this dataset:

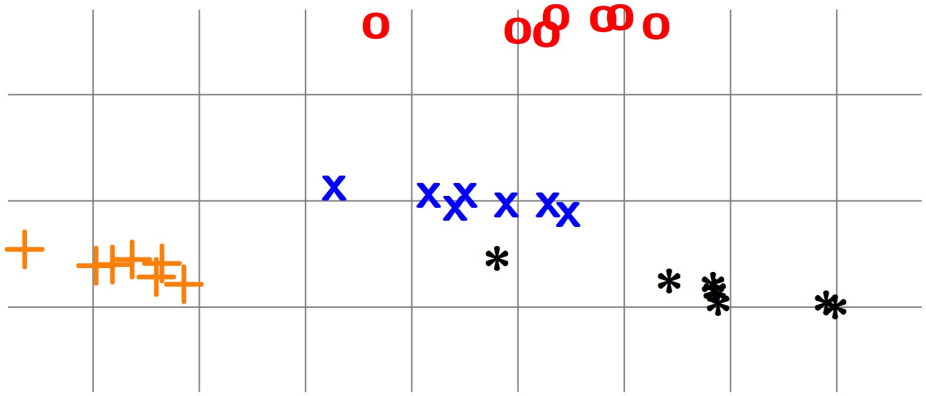


Figure 71: Faces above - dimension reduced to 2. Each of the four people is marked with a different marker. (Source: UML)

48.7 Interpreting and using the principal vectors as "typical data points"

Interestingly, the principal vectors themselves are vectors in $\mathbb{R}^d$. So they have the same dimension as the datapoints $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$. But they are not datapoints. They are orthonormal vectors in $\mathbb{R}^d$, carefully chosen such that the first k vectors provide the best linear approximation of dimension k to the dataset. In this sense, they are "typical" datapoints, maximally different from each other (since they are orthonormal). It is often very interesting to see what they would represent as datapoints.

For example, in the digit dataset, suppose we only look at the digit "3" and run PCA with $k = 2$. Then we can write any digit "3" as a sum of the mean $\overline{\mathcal{V}x}$, and the first two principal vectors:

$$\begin{aligned} \hat{f}(\lambda) &= \bar{x} + \lambda_1 v_1 + \lambda_2 v_2 \\ &= \end{aligned}$$

Figure 72: Source: ESL

The first principal component $\mathcal{V}v_1$ and the second principal component $\mathcal{V}v_2$ are orthogonal and

represent two maximally different "typical" digit "3". They are **not** digits in the dataset - they are basis vectors.

Another example: in the Yale face dataset, what would happen if we plot - as an imgae - each the leading principal vectors?

Figure 73: Source: Hao et al, Facial Recognition Using Tensor-Tensor Decompositions, SIAM Journal on Imaging Sciences 2013

These are sometimes called "eigenfaces". Any image can written as a linear combination of these orthonormal vectors, and any image in the dataset can be writte - with small error - as a linear combination **of the first few.** So they represent maximally different crucial features that appear in the faces in the dataset.

48.8 Practical considerations.

48.8.1 Fast computation of PCA

Recall that as machine learners we are also interested in numerical linear algebra, namely, algorithms that implement or linear algebra ideas in code efficiently and stably.

When $d \gg m$, so that we are working with a very high-dimensional dataset, there is a good reason to try dimension reduction as part of our preprocessing. However, exact diagonalization of the d-by-d empirical covariance costs $O(d^3)$. (This is a fact from numerical linear algebra you should remember.) So if d is huge, $O(d^3)$ can be a prohibitively expensive computation. When d is large and $d \gg m$ we can use a nice trick and calculate the eigenvectors of an m-by-m matrix instead. This costs $O(m^3)$. The following algorithm calculates PCA in the cost of $O(m^2 \cdot d)$ (see UML 23.1.1). Here is how you should remember it: our data matrix X (remember linear regression and logistic regression?) is m-by-d, and PCA costs order of $(small)^2 \cdot (large)$ steps.

PCA

input
A matrix of m examples $X \in \mathbb{R}^{m,d}$
number of components n
if $(m > d)$
$A = X^\top X$
Let $\mathbf{u}_1, \ldots, \mathbf{u}_n$ be the eigenvectors of A with largest eigenvalues
else
$B = XX^\top$
Let $\mathbf{v}_1, \ldots, \mathbf{v}_n$ be the eigenvectors of B with largest eigenvalues
for $i = 1, \ldots, n$ set $\mathbf{u}_i = \frac{1}{\|X^\top \mathbf{v}_i\|} X^\top \mathbf{v}_i$
output: $\mathbf{u}_1, \ldots, \mathbf{u}_n$

48.8.2 Choosing k

In practice, no one tells us the dimension k to which we should reduce. How shall we choose k? If we choose k too high, we are "wasteful" in that we don't need so many dimensions. If we choose k too low, we throw away essential parts of the data, so that the dimension-reduced dataset does not capture the essential features of the dataset.

Let's begin by a noticing that PCA tells us when the data sits **exactly** on a k-dimensional linear subspace of $\mathbb{R}^d$.

Exercise. Suppose that our training sample $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ is contained in a subspace $V \subset \mathbb{R}^d$, with $dim(V) = k$. Let S be the d-by-d empirical covariance matrix of the training sample. Show that $rank(S) \leqslant k$.

This means that if indeed the training sample lies exactly on a k-dimension linear subspace of $\mathbb{R}^d$, then we will have k nonzero principal values (recall that the principal values are just the eigenvalues of the empirical covariance S) and the rest will be zero.

Now imagine that the training sample lies close to such a subspace. What would happen? You guessed correctly - we will have k large principal values, and he rest will be non-zero but small. (Recall that we've seen something like this in linear regression.) So the best way to choose k from the data would be somehow related to finding how many "large" principal values there are. We don't discuss a formal algorithm for choosing k, but this is the general idea. The most popular informal method (which is not in fact an algorithm) involves plotting the principal components in decreasing order simply over their index, and making an "intuitive guess" about the number of "large" principal values. This number is chosen to be k. Such a plot is sometimes called a "Scree Plot".

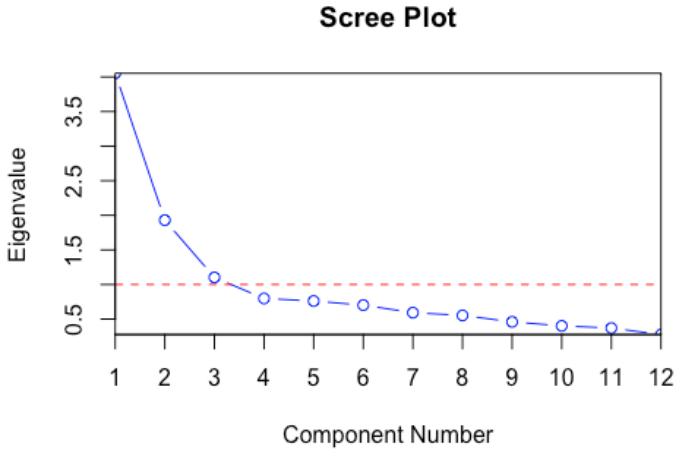


Figure 74: Typical plot of principal components over their index. (Source: Wikipedia)

Exercise. Write a simulation to observe this phenomenon. In your simulation, start with data in $\mathbb{R}^d$ that lie exactly on a k-dimensional linear subspace of $\mathbb{R}^d$, and plot the principal values.

Now move the data slightly away from the subspace and plot the principal values.

49 Clustering

To get another taste of an unsupervised learning problem, let's look at **clustering.** We are given an unlabeled training sample $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ and wish to partition the m points into k sets, or **clusters.** Sometimes we have a clear idea of what k should be (how many clusters we should be looking for), and sometimes we have to figure out k ourselves based on the data.

Why cluster?

- Clustering partitions the dataset into subsets of "similar" points. As an exploratory step, we might be interested in what are the common properties of each cluster found by out clustering algorithm.
- We may want to validate an existing partition someone gave us, and see if an unsupervised clustering algorithm (which uses no labels and no knowledge other than the points themselves) produces the same partition.

Note that there are no labels for us to use. This means that, unlike supervised classification, **there is no ground truth**. (Compare this to the opposite situation - in supervised classification, under the realizability assumption, there was a "ground truth" function f we were trying to predict.) Therefore, clustering is not a well-defined problem with a single answer. For example, the following dataset in $\mathbb{R}^2$

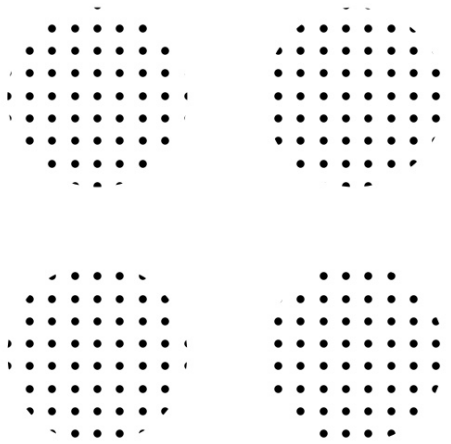


Figure 75: (Source: Shai Shalev-Shwartz slides)

can be clustered like this

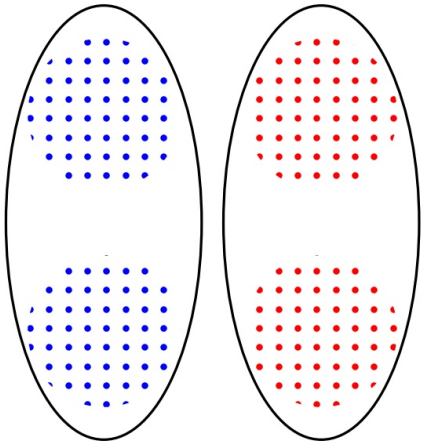


Figure 76: (Source: Shai Shalev-Shwartz slides)

or like this

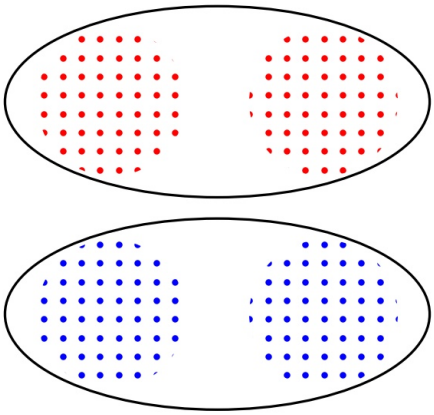


Figure 77: (Source: Shai Shalev-Shwartz slides)

Even though the problem is not well-defined, we can nonetheless develop a systematic approach to clustering. This systematic approach can be defined on any sample space, not just $\mathbb{R}^d$:
~

Assume we have a distance measure (a metric) on our sample space. (For example, if $\mathcal{X} = \mathbb{R}^d$ we may consider the Euclidean norm and take $d(\mathcal{V}x_1, \mathcal{V}x_2) = \|\mathcal{V}x_1 - \mathcal{V}x_2\|$.)

- Definition: A clustering of the dataset $\{\mathcal{V}x_1, \ldots, \mathcal{V}x_m\}$ into k clusters is simply a partition $\{\mathcal{V}x_1, \ldots, \mathcal{V}x_m\} = \biguplus_{j=1}^{k} C_j$

- Define a **cost function** for this clustering:

$$G(C_1, \ldots, C_k) = \min_{\mu_1, \ldots, \mu_k \in \mathbb{R}^d} \sum_{j=1}^{k} \sum_{\mathcal{V}x \in C_j} d(\mathcal{V}x, \mu_j)^2$$

- The quantity

$$\text{argmin}_{\mu_j \in \mathbb{R}^d} \sum_{\mathcal{V}x \in C_j} d(\mathcal{V}x, \mu_j)^2$$

  is called the **centroid** of cluster C_j.

Exercise. Assume $\mathcal{X} = \mathbb{R}^d$ and d is the Euclidean norm. Show that the centroid of a set C_j is simply the average

$$\mu_j := \frac{1}{|C_j|} \sum_{\mathcal{V}x \in C_j} \mathcal{V}x \,.$$

The quantity μ_j has a name in Mechanics. How is it called? It has a name in Statistics. How is it called? The quantity $\sum_{\mathcal{V}x \in C_j} d(\mathcal{V}x, \mu_j)^2$ has a name in Mechanics. How is it called? It has a name in Statistics. How is it called?

How do we find the minimum of G? Fix k. The objective function G is a **function of partitions**: minimizing it would navigating the space of all possible partitions of m objects into k subsets. (How many are there?) Not surprisingly, it can be shown that minimizing the cost function G is NP-hard, and we must resort to heuristics. The most famous heuristic for minimizing G is an iterative algorithm known as **k-means clustering.**

49.1 k-means

k-means clustering uses **Lloyd's Algorithm**, a heuristic approach that attempts to minimize G. Let's assume for simplicity that $\mathcal{X} = \mathbb{R}^d$ and the distance function d is just the Euclidean norm. (This heuristic can be adapted to other spaces as well.)

- Input: Set $\mathcal{V}x_1, \ldots, \mathcal{V}x_m$ and number of clusters k
- Step 0. Choose initial centroids $\mu_1, \ldots, \mu_k$
- Repeat until convergence:
  (i) set C_j to be the points $\mathcal{V}x_i$ closer to μ_j than to any other centroid.
  (ii) update μ_j the centroid of C_j, which as you showed in the exercise above, is simply

$$\mu_j := \frac{1}{|C_j|} \sum_{\mathcal{V}x \in C_j} \mathcal{V}x$$

What's going on here? If we choose the centroids, this induces a simple partition of the dataset - each point is associated to a subset according to its nearest centroid. (These subsets are called **Voronoi cells.**) And if we choose a partition, we define the centroids as the subset averages.

So Lloyd's algorithm is an iterative **alternating** algorithm. We start from some initial guess of the centroids, then use them to induce a partition, then use this partition to define centroids,

and so on. We hope that the algorithm converges; we hope that won't be too sensitive to the initial choice of centroids.

Here is an example in $d = 2$ dimensions, on $m = 300$ points, and we cluster into $k = 3$ clusters:

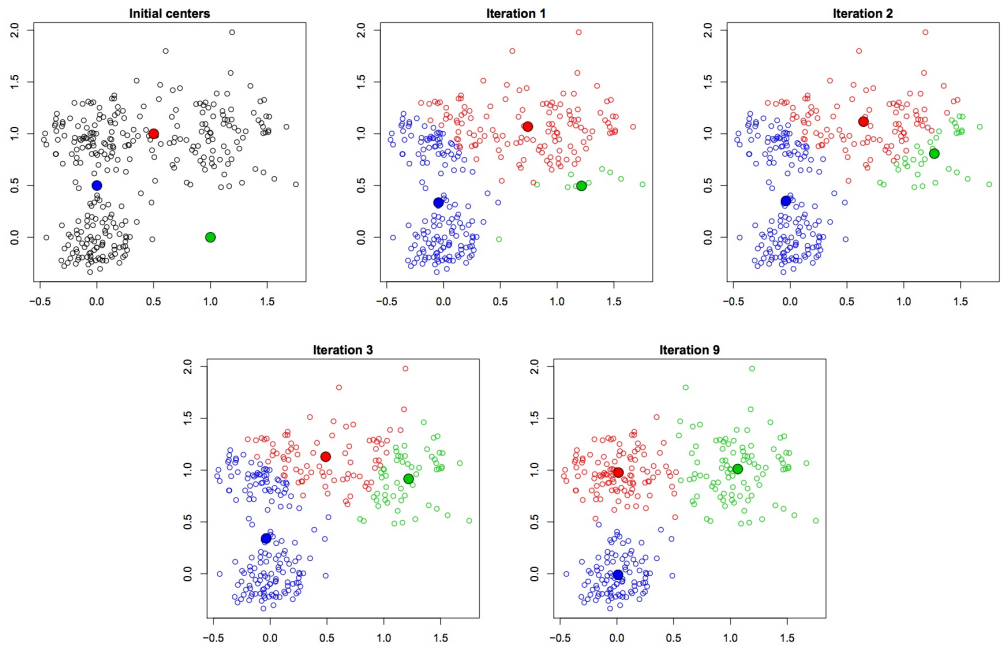


Figure 78: (Source: CMU 36-462/36-662)

Here are some properties of Lloyd's algorithm, which we won't prove. You are encouraged to play with them in simulation and observe them:

- Variation within each cluster always decreases from iteration to iteration.
- The algorithm always converges - but may converge very slowly.
- The end result can drastically depend on the initialization. In other words, the algorithm converges to a local minimum of the function G which depends on the initial choice of centroids.

Here is an example using the same dataset as in the image above. This time we specify $k = 4$. Each panel has different initial conditions:

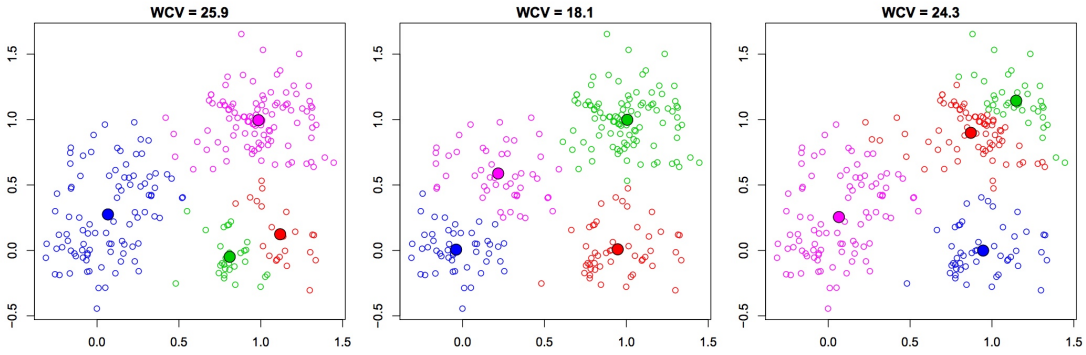


Figure 79: (Source: CMU 36-462/36-662)

Since we are attempting to minimize the global cost function G, one way to handle the dependence on initial conditions is to run the algorithm a few times and take the result which achieved lowest cost.

Finally, how do we choose the number of clusters k? In fact k is a bias-variance parameter. The higher k (the more subsets we take) the lower the value of G we will obtain. This is analogous to overfitting in supervised learning as we increase the complexity of the model:

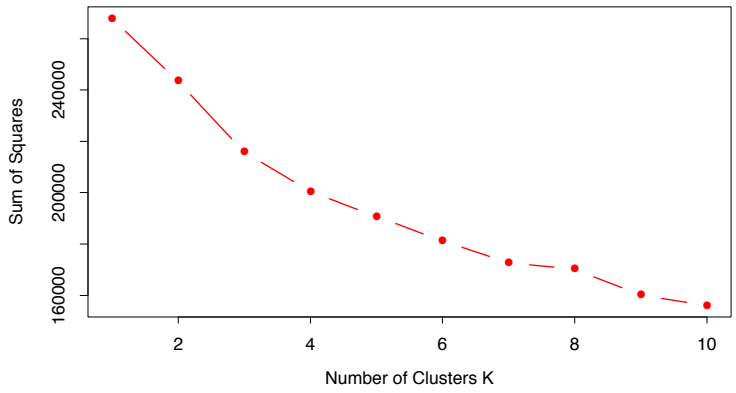


Figure 80: Source: ESL

Lecture 9: ML Actually

Lecture 10: Convex Optimization and SGD

Lecture 11: Online Learning and RL

Lecture 12: Deep Learning