

CS 189

Introduction to Machine Learning

Notes

Druv Pai

Contents

1	Logistics	3	7	Discriminant Analysis	16
2	Introduction	3	8	Parameter Estimation	17
3	Linear Classifiers	4	9	Anisotropic Multivariate Gaussians	18
4	Abstractions	11	10	Regression	20
5	Optimization Algorithms	12	11	Decision Trees	27
6	Decision Theory	14			

1 Logistics

Homework is on a Wednesday-Wednesday cycle. The class textbook is ostensibly *Elements of Statistical Learning* by Tibshirani.

2 Introduction

The core of the class is to find patterns in data, then using the patterns to make predictions about new data. Models and statistics help us understand patterns. Optimization algorithms “learn” the patterns. Data drives everything, and you cannot learn much if the data is small or is poor-quality, but algorithms with lots of data are very successful.

In general, sample points are elements of a set \mathcal{X} and responses (labels) are elements of a set \mathcal{Y} . The true joint distribution of the data is generally given by $(X, Y) \sim \mathcal{D}$; the empirical joint distribution (the uniform distribution over the given pairs $(X_i, Y_i)_{i=1}^n$) is $\mathcal{D}_{\text{data}}$.

There are many different machine learning tasks; one of the simplest and best is classification.

Definition 1 (Classifier). A *classifier* partitions data into several classes.

Example 2. Say we have training data $((X_i, Y_i), Z_i)_{i=1}^m$, where X_i is credit card balance, Y_i is income, and Z_i is whether the person has defaulted on their card (as, say, an indicator variable). We wish to predict the Z value Z_{pred} for new data points $(X_{\text{new}}, Y_{\text{new}})$.

How do we solve this problem? We

- first, collect data,
- then, build a classifier based on the data,
- finally, classify any new points using the classifier.

One possible classifier is to take the k nearest neighboring points $(X_{n_i}, Y_{n_i})_{i=1}^k$ for the new point $(X_{\text{new}}, Y_{\text{new}})$ in \mathcal{X} , and average their Z values, then round to get the predicted Z value Z_{pred} , so $Z_{\text{pred}} = \text{round}\left(\sum_{i=1}^k Z_{n_i}\right)$. Accordingly, this is called the *k-nearest-neighbor classifier*.

For small k , the k -nearest-neighbor classifier is prone to having really awkward decision boundaries.

Definition 3 (Overfitting). A classifier is *overfitting* when it does not does much more poorly on new data points than it does on old data points.

It’s easy to see that, for example, 1-nearest-neighbors classifiers overfit on any noisy training data; on the other hand, for large k , the k -nearest-neighbor classifier does not overfit and the decision boundary is much more smooth, but it may degenerate into being a bad classifier with an incorrect decision boundary.

Another classifier is a *linear classifier*, which is somewhat of a misnomer. In a binary classification situation with a classification function $f: \mathbb{R}^d \rightarrow \{-1, 1\}$, a linear classifier f is an $(d - 1)$ -dimensional affine hyperplane. This classifier is a smooth solution which may oversimplify the real scenario, but will not overfit since the model is very constrained.

Definition 4 (Hyperparameter). A *hyperparameter* is a property of a machine learning model that is not explicitly learned from data.

Hyperparameters usually control the balance between overfitting and efficacy of the model.

To create a classifier model, given classified training data and test data, we:

- train a classifier on the training data, so it learns to distinguish classes from each other.
- test the classifier on the test data.

There are two kinds of errors associated with a model.

Definition 5 (Errors). The *training set error* is the fraction of misclassified data in the training set. The *test set error* is the fraction of misclassified data in the test set.

Definition 6 (Outliers). *Outliers* are points in the training data which are atypical in some way or just misclassified.

We don't really care about these outlier points very much; algorithms that overfit will sometimes weigh outliers a lot, and this is a way to create an overfitting model.

Hyperparameters generally control the balance between overfitting and underfitting. To select optimal hyperparameters, we do something called the validation process:

Definition 7 (Validation). The validation process produces a machine learning model with good hyperparameters. In the validation process, we:

- hold back a subset of the labeled data, called the validation set
- train the classifier multiple times with different hyperparameter settings
- choose the settings that work best on the validation set

Now there are three data sets:

- the training set, used to learn model weights.
- the validation set, which is used to tune hyperparameters and choose among different models.
- the test set, used as the final evaluation of the model.

There are two types of overarching problem classifications, and problems under them:

- *Supervised learning*, when the training data has labels
 - *Classification*: data labels are discrete and unordered
 - *Regression*: data labels are continuous or ordered
- *Unsupervised learning*, when the training data has no labels
 - *Clustering*: observing clusters in (transformed) Euclidean space with respect to the data points, hinting at the data generation process.
 - *Dimensionality reduction*: finding projections of high dimensional data onto low dimensional manifolds.

3 Linear Classifiers

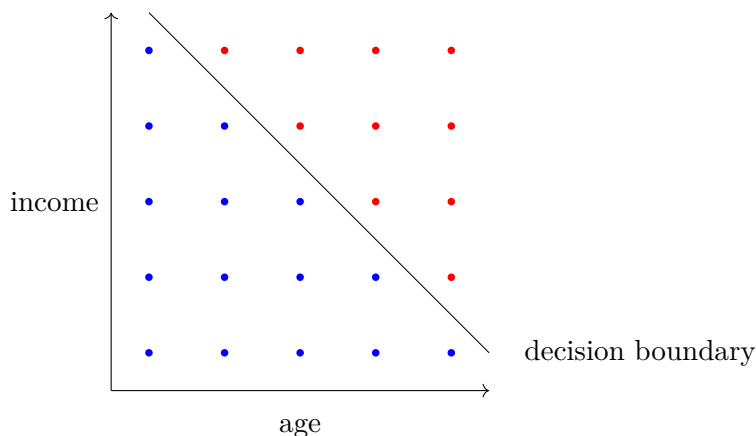
We are given a dataset of n observations, each with d features. The reason why we use the letter d is that it reflects the dimension of our data points. Some observations belong to a class C ; some do not, and belong

to another class X .

Example 8. Observations can be bank loans, features can be income and age ($d = 2$). Some observations are in the class `DEFAULTED`, and some are not. The goal is to predict whether future borrowers will default, based on their income and age.

The way we think about the dataset is that we represent each observation as a point in a d -dimensional space. Each sample point is referred to as a feature vector, or an independent variable vector.

For the example, one may have the dataset



Definition 9 (Decision Boundary). The *decision boundary* is the boundary chosen by the classifier to separate items in the class C from those that do not.

Overfitting, in this case, refers to a sinuous decision boundary that fits sample points so well that it doesn't classify future points well.

For multiclass classifiers, some classifiers train a classification for each class and weight their output according to their returned probability. Others train an “*all-for-one*” method, which allows simultaneous probability computation for each class.

Definition 10 (Decision Function). A *decision function* is a function $f(x)$ that maps a sample point x to a scalar such that

$$f(x) > 0 \text{ if } x \in C, \quad f(x) \leq 0 \text{ otherwise}$$

This is also known as a *predictor function* or *discriminant function*.

For these classifiers, the *decision boundary* is $\{x \in \mathcal{X} : f(x) = 0\}$. Usually, if $\mathcal{X} = \mathbb{R}^d$, this set is a $(d - 1)$ -dimensional manifold in \mathbb{R}^d .

Definition 11 (Isosurface and Isovalue). The set $\{x \in \mathcal{X} : f(x)\}$ is called an *isosurface* of f for the *isovalue* 0. In particular, f has other isosurfaces for other isovalues, e.g. $x : f(x) = 1$.

Sometimes there are needs for explicit nonzero isovalues.

Linear classifiers have linear decision boundaries, where the boundary manifold is a $(d - 1)$ -dimensional hyperplane $\{x : w \cdot x = \alpha\}$. Usually, linear classifiers also use a linear decision function.

Claim 12. Let $x, y \in H = \{x: w \cdot x = -\alpha\}$. Then $w \cdot (y - x) = 0$.

Proof. By computation,

$$w \cdot (y - x) = -\alpha - (-\alpha) = 0$$

as desired. \square

In this case w is called the normal vector of H , because as the theorem shows, w is normal (perpendicular) to H . If w is a unit vector (i.e. $\|w\|_2 = 1$), then $w \cdot x + \alpha$ is the signed distance from x to H . In particular, $\text{sign}(w \cdot x + \alpha) > 0$ if w is on the same side of H as x , and 0 otherwise. If $\|w\|_2 \neq 1$, then the same concept holds, except we need to normalize w and α to find the distance.

Definition 13 (Weights). The coefficients in w , plus α , are called *weights* (or parameters or regression coefficients).

Sometimes we talk about when input data is linearly separable.

Definition 14 (Linearly Separable). The input data is *linearly separable* if there exists a hyperplane that separates all the sample points in class C from all the points not in class C .

Examples of Simple Linear Classifiers

Definition 15 (Centroid Method). A simple classifier is the centroid method. We compute the mean μ_C of all points in class C , and the mean μ_X of all points not in class C . The decision function is

$$f(x) = \underbrace{(\mu_C - \mu_X)}_{\text{normal vector}} \cdot x - (\mu_C - \mu_X) \cdot \underbrace{\frac{\mu_C + \mu_X}{2}}_{\text{midpoint}(\mu_C, \mu_X)}$$

More explicitly, the algorithm for the centroid method is

Algorithm 1 The centroid method.

Input: A set of data points $(X_i, Y_i)_{i=1}^n$ where $Y_i \in \{C_1, C_2\}$ and $Y_i = 2\mathbb{1}(X_i \in C_1) - 1$.

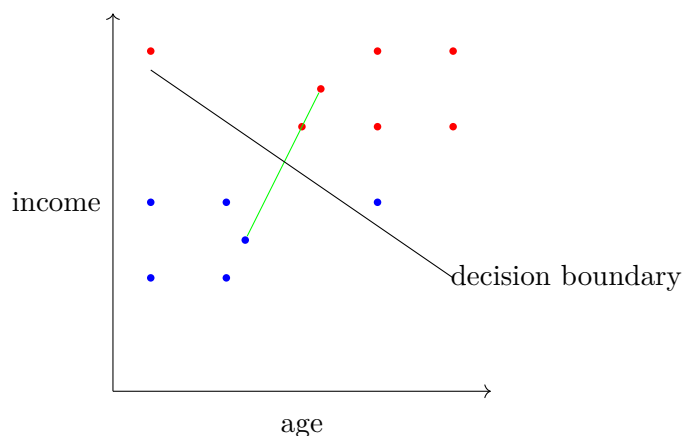
Output: A decision function $f(x)$.

$$\mu_{C_1} \leftarrow \frac{1}{|C_1|} \sum_{Y_i=C_1} X_i$$

$$\mu_{C_2} \leftarrow \frac{1}{|C_2|} \sum_{Y_i=C_2} X_i$$

$$\textbf{return } f(x) \stackrel{\text{def}}{=} (\mu_{C_1} - \mu_{C_2}) \cdot x - (\mu_{C_1} - \mu_{C_2}) \cdot \frac{\mu_{C_1} + \mu_{C_2}}{2}$$

We want the decision boundary to go exactly between the mean vectors μ_C and μ_X ; this is achieved when $x = 0$. The decision boundary is the hyperplane that bisects the line segment with endpoints μ_C and μ_X .



Question 16. Is it possible to design a dataset that the centroid method misclassifies every data point but one?

It turns out that it is possible.

Now we move to the perceptron algorithm, which uses gradient descent and is correct for linearly separated points.

Definition 17 (Perceptron Classifier). Consider n sample points X_1, \dots, X_n . For each sample point, let the label $Y_i = 1$ if $X_i \in C$ and $Y_i = -1$ if $X_i \notin C$. For simplicity, consider only decision boundaries that pass through the origin. Then the goal is to find weights w such that $X_i \cdot w \geq 0$ if $Y_i = 1$, and $X_i \cdot w \leq 0$ if $Y_i = -1$. Putting these together, we get $Y_i X_i \cdot w \geq 0$ for every i .

We may conceptualize hyperplanes in terms of x -space V , or feature space, and w -space, or weight space W . In particular, there is a natural injection between hyperplanes $H \in V$ and vectors $w \in W$ given by $\{x: w \cdot x = 0\} \mapsto w$. Correspondingly, there is a natural injection from hyperplanes $H \in W$ and vectors $x \in V$ given by $\{w: x \cdot w = 0\} \mapsto x$. If vector $x \in \{z: w \cdot z = 0\} \subseteq V$, then $w \cdot x = 0$, so $w \in \{y \cdot x \cdot y = 0\} \subseteq W$. If we want to enforce the inequality $x \cdot w \geq 0$, then

- in V , x should be on the same side of $\{z: w \cdot z = 0\}$ as w , and
- in W , w should be on the same side of $\{z: x \cdot z = 0\}$ as x .

Enforcing these constraints gives a set of feasible weight vectors, each of these giving a classifier.

How do we solve this optimization problem? The idea is that we define a risk function R that is positive if some constraints are violated, and zero otherwise. Then we use optimization to choose w that minimizes R .

Define the loss function

$$L(z, Y_i) = \begin{cases} 0, & Y_i z \geq 0 \\ -Y_i z, & \text{otherwise} \end{cases} = -\min(0, Y_i z)$$

where z is the classifier's prediction, and Y_i is the training label. If $\text{sign}(z) = \text{sign}(Y_i)$, the loss function is 0. If z has the wrong sign, the loss function is positive. Now define the risk function given by

$$R(w) = \sum_{i=1}^n L(X_i \cdot w, Y_i) = \sum_{i \in V} -Y_i X_i \cdot w$$

where $V = \{i: Y_i X_i \cdot w < 0\}$ is the set of indices of misclassified sample points.

If w classifies all X_1, \dots, X_n correctly, then $R(w) = 0$. Otherwise, $R(w) > 0$, and we want to find a better w . Our goal is to solve this optimization problem. To solve this optimization problem, we use an optimization algorithm.

Gradient Descent

The ubiquitous gradient descent algorithm is

Algorithm 2 Regular gradient descent algorithm.

Input: A convex objective function $J(w): \mathbb{R}^d \rightarrow \mathbb{R}$ to be minimized, a step size $\varepsilon \in \mathbb{R}_{>0}$.

Output: The minimizer w^* of $J(w)$.

$w \leftarrow$ randomly initialized weight vector.

while $\nabla_w J(w) > 0$ **do**

$w \leftarrow w - \varepsilon \nabla_w J(w)$

return w

In the case of perceptrons, a random weight initialization is not always that good; sometimes we want w to be set at some $Y_j X_j$, so that the starting weight classifies at least one point correctly.

Also, as in the case of perceptrons, this algorithm is really slow, taking $\mathcal{O}(nd)$ time per iteration.

We amend this algorithm to account for this by only evaluating the loss function on one data point per iteration. To clarify, $L_w(X, Y)$ is the loss incurred on a training example (X, Y) with parameter w .

Algorithm 3 Stochastic gradient descent algorithm.

Input: A loss function $L_w(X, Y) \geq 0$, a step size $\varepsilon \in \mathbb{R}_{>0}$.

Output: The minimizer w^* of $\sum_{i=1}^n f_{X_i}(w)$.

$w \leftarrow$ randomly initialized weight vector.

while $\exists (X_i, Y_i)$ s.t. $L_w(X_i, Y_i) > 0$ **do**

$w \leftarrow w - \varepsilon \nabla_w L_w(X_i, Y_i)$

return w

If the separating hyperplanes are all affine, we can just add on an extra 1 entry to each data point X , so

$X_{\text{new}} = \begin{bmatrix} X \\ 1 \end{bmatrix}$, and the w vector will learn the affine intercept naturally.

Note that for the perceptron, $\nabla_w L_w(X_i, Y_i) = -\nabla_w \min(0, Y_i X_i \cdot w) = -\min(0, Y_i X_i)$, and inputting this into stochastic gradient descent yields the perceptron algorithm.

Algorithm 4 The perceptron algorithm, via stochastic gradient descent.

Input: A set of linearly separable sample points $(X_i, y_i)_{i=1}^n$ where $Y_i \in \{C_1, C_2\}$, and a step size $\varepsilon \in \mathbb{R}_{>0}$.

Output: A linear decision function $f(x)$.

$w \leftarrow Y_j X_j$ for some $j \in \{1, \dots, n\}$.

$V \leftarrow \{(X_i, Y_i) : Y_i X_i \cdot w < 0\}$

while $|V| > 0$ **do**

$(X_i, Y_i) \leftarrow$ entry of V

$w \leftarrow w + \varepsilon \min(0, Y_i X_i)$

$V \leftarrow \{(X_i, Y_i) : Y_i X_i \cdot w < 0\}$

return $f(x) \stackrel{\text{def}}{=} \text{sign}(w \cdot x)$

Support Vector Machines

The support vector machine is like a perceptron, except there is a notion of a “best” separating hyperplane, whereas a perceptron just tries to get any separating hyperplane. In particular, we wish to find the best weight vector for a separating hyperplane.

Definition 18 (Margin). Assume that $(X_i, Y_i)_{i=1}^n$ is a set of linearly separable sample points. Also let the decision boundary be L . Then the *margin* is the distance from the decision boundary to the nearest sample point, that is, $m = \min_{X_i} \min_{\ell \in L} d(X_i, \ell)$.

For a successful classifier, we want to find a decision boundary with a maximized margin. We wish to have at least a margin of $c > 0$, so we enforce that $Y_i (w \cdot X_i + \alpha) \geq c$ for all $i \in \{1, \dots, n\}$. Since this constraint doesn’t change when $(w, \alpha, c) \mapsto (\frac{w}{c}, \frac{\alpha}{c}, 1)$ (dividing both sides by c), it suffices to have the constraint $Y_i (w \cdot X_i + \alpha) \geq 1$. If $\|w\| = 1$, then the signed distance from the hyperplane $H = \{x : w \cdot x = 0\}$ to X_i is $d(H, X_i) = w \cdot X_i + \alpha$. Otherwise,

$$d(H, X_i) = \frac{w}{\|w\|_2} \cdot X_i + \frac{\alpha}{\|w\|_2}$$

Hence the margin is

$$\min_i \frac{1}{\|w\|_2} |w \cdot X_i + \alpha| \geq \min_i \frac{1}{\|w\|_2} = \frac{1}{\|w\|_2}$$

In this manner there exists a *slab* of width $\frac{2}{\|w\|_2}$ which has no sample points. To maximize the margin, we have

$$\max_w \frac{1}{\|w\|_2} = \min_w \|w\|_2 = \min_w \|w\|_2^2$$

with the constraints $Y_i (X_i \cdot w + \alpha) \geq 1$ and is called a quadratic program in $d + 1$ dimensions. Since $\|w\|_2^2$ is convex, it has one unique solution, which gives us a maximum margin classifier, also known as a hard-margin support vector machine.

We now discuss soft-margin support vector machines, which are a modification of maximum-margin classifiers (hard-margin support vector machines). They solve two problems of hard-margin support vector machines:

- Hard-margin support vector machines fail if the data is not linearly separable; soft-margin support vector machines do not fail.
- Hard-margin support vector machines are very sensitive to outliers; soft margin support vector machines do not.

What we will do today is develop a support vector machine that violates the margin. We introduce the slack of a variable, ξ_i , as conceptually the amount that the data point (X_i, Y_i) can violate the margin by; in particular, $d(X_i, H) = \frac{\xi_i}{\|w\|_2}$. We minimize the total amount of slack. The constraints now take the form $Y_i(X_i \cdot w + \alpha) \geq 1 - \xi_i$. We also have to impose the constraint that the slack variables are nonnegative, so to avoid some data points having lots of slack and others having none, i.e. $\xi_i \geq 0$. When the signed distance $d(X_i, H) \leq \frac{1}{\|w\|_2}$, then (X_i, Y_i) is said to violate the margin. In the soft-margin SVM, the margin is actually just defined as $\frac{1}{\|w\|_2}$. To prevent overfitting in the slack terms, we add a loss function of the slack to the objective function. The optimization problem becomes

$$\begin{aligned} & \text{minimize } \|w\|_2^2 + C \sum_{i=1}^n \xi_i \\ & \text{subject to } Y_i(X_i \cdot w + \alpha) \geq 1 - \xi_i \quad \forall i \in [1, n] \\ & \quad \xi_i \geq 0 \quad \forall i \in [1, n] \end{aligned}$$

The value of C is chosen by validation and is not trained.

This optimization problem is called a *quadratic program* in $d + n + 1$ dimensions, where $w \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$, and $\xi \in \mathbb{R}^n$, where $(\xi)_i = \xi_i$. There are $2n$ constraints. A quadratic program has a quadratic objective function and linear constraints. Most quadratic programs the Hessian is also positive semidefinite, which basically means that the program is convex in all dimensions. In this case the Hessian is $\nabla_{w, \xi}^2 (\|w\|_2^2 + C \sum_{i=1}^n \xi_i) = \begin{bmatrix} 2I_d & 0_n \\ 0_n & 0_d \end{bmatrix}$ which is clearly positive semidefinite. The value $C > 0$ is a scalar regularization hyperparameter that trades off:

- the desire to maximize the margin against the desire to keep all the slack variables small
- the tendency to underfit (via misclassifying training data) against the tendency to overfit (via converging to a hard margin SVM).
- the small effect of outliers against a large effect of outliers
- more flat, linear boundaries against more sinuous boundaries

The last item really only applies for the special cases of nonlinear decision boundaries.

We now consider how to turn linear classifiers (such as hard-margin SVMs) into nonlinear classifiers (like neural networks). The main idea is to make nonlinear features that lift points into a higher dimensional space. Then in this higher dimensional space, we can apply a standard linear classifier to obtain the classification function. This has the same effect as a low-dimensional nonlinear classifier.

Example 19 (Parabolic Lifting Map). We're given some sample points $(X_i, Y_i)_{i=1}^n$. We define the parabolic lifting map as $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{d+1}$, characterized as $\Phi(X) = \begin{bmatrix} X \\ \|X\|_2^2 \end{bmatrix}$. This lifts X onto the paraboloid $X_{d+1} = \|X\|_2^2$. A linear classifier in \mathbb{R}^{d+1} , or Φ space, then it induces a spherical classifier in \mathbb{R}^d , or X space, or feature space.

Claim 20. The lifted points $\Phi(X_1), \dots, \Phi(X_n)$ are linearly separable if and only if X_1, \dots, X_n are separable by a hypersphere (including degenerate hyperspheres, which are hyperplanes).

Proof. Consider a hypersphere in \mathbb{R}^d with center c and radius ρ . Then the points x inside the hypersphere

take the form

$$\begin{aligned}\rho^2 &> \|x - c\|_2^2 \\ &> \|x\|_2^2 - 2c \cdot x + \|c\|_2^2 \\ \rho^2 - \|c\|_2^2 &> \begin{bmatrix} -2c^\top & 1 \end{bmatrix} \begin{bmatrix} x \\ \|x\|_2^2 \end{bmatrix}\end{aligned}$$

where $\begin{bmatrix} -2c^\top & 1 \end{bmatrix}$ is the normal vector in \mathbb{R}^{d+1} to the paraboloid $x_{d+1} = \|x\|_2^2$, and $\begin{bmatrix} x \\ \|x\|_2^2 \end{bmatrix} = \Phi(x)$. Thus all points separable by a hypersphere in \mathbb{R}^d are linearly separable under the map Φ in \mathbb{R}^{d+1} .

All of these steps are reversible, so all points linearly separable in \mathbb{R}^{d+1} under the map Φ are separable by a hypersphere in \mathbb{R}^d . \square

Example 21 (Axis-Aligned Ellipsoid/Hyperboloid Decision Boundaries). Axis-aligned ellipsoids in d dimensions have the equation

$$\sum_{i=1}^d \sum_{j=0}^2 w_{3(i-1)+j} x_i^j + \alpha = 0$$

We obtain the map $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{2d}$ where $\Phi(X) = [X_1^2 \ \cdots \ X_d^2 \ X_1 \ \cdots \ X_d]^\top$. The hyperplane is $w \cdot \Phi(x) + \alpha = 0$.

Example 22 (Ellipsoid/Hyperboloid Decision Boundaries). Ellipsoids in d dimensions have the equation

$$\sum_{j_1, \dots, j_d=0}^2 w_{j_1, \dots, j_d} \prod_{i=1}^d x_i^{j_i} + \alpha = 0$$

We obtain the map $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{(d^2+3d)/2}$ by $\Phi(x) = [X_1^2 \ \cdots \ X_d^2 \ X_1 X_2 \ \cdots \ X_{d-1} X_d \ \cdots \ X_1 \ \cdots \ X_d]^\top$. The hyperplane is $w \cdot \Phi(x) + \alpha = 0$.

The isosurface defined by this equation is called a quadric; when $d = 2$ the isosurfaces are conic sections.

Example 23 (Polynomial Decision Function). Let the degree of the polynomial we want be p . Then the mapping is $\Phi(X) = [X_1^p \ \cdots \ X_d^p \ \cdots \ \prod_{i=1}^d X_i^{j_i} \ \cdots \ X_1 \ \cdots \ X_d]^\top$. This mapping is $\Phi: \mathbb{R}^d \rightarrow \mathbb{R}^{\mathcal{O}(d^p)}$. This blows up, but we will use the “kernel trick” to compute these values later.

Example 24 (Edge Detection). Define an edge detector to be an algorithm for approximating grayscale/color gradients in an image, i.e. top filter, Sibel filter, oriented Gaussian derivative filter. We collect the line orientations in local histograms (each having 12 orientation bins per region); then the histograms can be used as features, instead of raw features.

4 Abstractions

We can think of several different abstractions of machine learning. Most of the things we do resolve into four different layers of abstractions.

The top level is the application (i.e. the problem that we’re trying to solve) and the data that we’re given. The question we want to ask is is our data labeled (with a class or numerical score), or not?

- If the answer is yes, then the labels might be categorical (indicating a classification application) or quantitative (indicating a regression application).
- If the answer is no, then the problem is either identifying similarities within the data (indicating a clustering application) or relative positioning within the data (indicating a dimensionality reduction algorithm).

The second level is the model. A model is either parametric (a model can be characterized by a set of parameters, like weights) or non-parametric. A model consists of either:

- In a parametric model, there are decision functions (linear, polynomial, logistic, neural networks, etc.).
- In a non-parametric model, we make decisions based on data (nearest neighbors, decision trees, random forests, etc.).
- Features.
- Low vs. high capacity (which affects overfitting, underfitting, and inference).

The third level is the optimization problem. An optimization problem consists of variables, objective function, and constraints (for example, unconstrained optimization, a convex program, least squares regression, principal component analysis).

The fourth level is the optimization algorithm that solves the optimization problem. This can be gradient descent for differentiable problems, or the simplex method for programs, or the singular value decomposition.

We will discuss what optimization algorithms we have and which problems are prototypical in the space.

5 Optimization Algorithms

Unconstrained Optimization

We have a continuous objective function $f(w)$. Our goal is to find $w^* = \operatorname{argmin}_w f(w)$.

We have some definitions here for posterity. Call a function $f: \mathbb{R}^d \rightarrow \mathbb{R}$ Shewchuk-smooth if $f \in C^1$ (its first derivative is continuous). With respect to a function $f: \mathbb{R}^d \rightarrow \mathbb{R}$, call x a global minimum if $f(x) \leq f(y)$ for each $y \in \mathbb{R}^d$, and call x a local minimum if there exists $\varepsilon > 0$ such that for every Δ with $\|\Delta\|_2 = \varepsilon$ we have $f(x) \leq f(x + \Delta)$.

Definition 25 (Convexity). A set $S \in V$, where V is a vector space, is *convex* if $\forall x, y \in S$,

$$\alpha x + (1 - \alpha) y \in S.$$

A function $f: \mathbb{R}^d \rightarrow \mathbb{R}$ is convex if for every $x, y \in \mathbb{R}^d$,

$$f(\alpha x + (1 - \alpha) y) \geq \alpha f(x) + (1 - \alpha) f(y).$$

A convex function has either

- no minimum (for every $\varepsilon > 0$ there exists x such that $f(x) < -\varepsilon$), or
- just one local minimum, in which case it is also the global minimum, or
- a connected (and path connected) set S of local minima that are all global minima.

If f is smooth, we can use the optimization methods:

- Gradient descent:
 - The blind algorithm found here: Section 3
 - With line search:
 - * Secant method
 - * Newton-Raphson method (which may need the Hessian matrix $\nabla_x^2 f(x)$)
 - Stochastic (blind) gradient descent, found here: Section 3
- Newton's method (needing the Hessian matrix $\nabla_x^2 f(x)$)
- Nonlinear conjugate gradient

If f is non-smooth, we can use the optimization methods:

- Gradient descent:
 - The blind algorithm found here: Section 3
 - With direct line search (e.g. golden section search)
- BFGS

We discussed line search a lot; line search picks the direction of steepest descent, then along this one-dimensional manifold we solve the optimization problem using one-dimensional methods.

Constrained Optimization

Like before, we have a continuous objective function $f(w)$. We also have a continuous constraint function $g(w) = 0$, which describes an isosurface.

One of the methods we can use is Lagrange multiplication, where we have $\nabla_w f(w) = \lambda \nabla_w g(w)$; solving for the optimal λ^* and then w^* finds the constrained optimum.

Another algorithm we can use is that of a linear program. A linear program has a linear objective function $f(w) = c^T w$, and linear inequality constraints $Aw \leq b$, where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$. Formally, we write the linear program as

$$\begin{aligned} & \text{minimize } c^T x \\ & \text{subject to } Ax \leq b \end{aligned}$$

Each linear inequality $a_{:,i}^T w \leq b_i$ separates the space into two half-spaces. The set of points w that satisfy all constraints is a convex polytope called the feasible region. We call the feasible region \mathcal{F} . The optimal $w \in \mathcal{F}$ is the $x \in \mathcal{F}$ furthest in the direction of c . The optimum w achieves equality for some constraints (called the active constraints) but not most.

Example 26. The support vector machine, which is a quadratic program, can be turned into a (crappy) linear program:

$$\begin{aligned} & \text{maximize}_{w, \alpha} 0 \\ & \text{subject to } Y_i (w \cdot X_i + \alpha) \geq 1 \quad \forall i \in [1, n] \end{aligned}$$

In this example, every feasible point (w, α) gives a linear classifier. The data are linearly separable if and only if $\mathcal{F} \neq \emptyset$. This is also true for the regular (quadratic) support vector machine.

Linear programs can be solved via simplex method, or interior point methods.

A quadratic program is the same as a linear program, except the objective function can be quadratic and convex. Formally, we write that

$$\begin{aligned} & \text{minimize } x^T Q x + c^T x \\ & \text{subject to } Ax \leq b \end{aligned}$$

where Q is a symmetric positive definite matrix (a positive definite matrix has $x^T Q x > 0$ for all $x \neq 0$). This program has only one local minimum, and therefore a global minimum.

If Q is indefinite (has negative eigenvalues), then the problem is **NP-hard**. If Q is positive semidefinite, then the objective function has multiple local minima, but the problem is not too hard to solve.

Example 27. The hard-margin support vector machine is a quadratic program:

$$\begin{aligned} & \text{maximize } \|w\|_2^2 \\ & \text{subject to } Y_i (w \cdot X_i + \alpha) \geq 1 \quad \forall i \in [1, n] \end{aligned}$$

and so is the soft-margin support vector machine:

$$\begin{aligned} & \text{maximize } \|w\|_2^2 + C \sum_{i=1}^n \zeta_i \\ & \text{subject to } Y_i (w \cdot X_i + \alpha) \geq 1 - \zeta_i \quad \forall i \in [1, n] \\ & \quad \zeta_i \geq 0 \quad \forall i \in [1, n] \end{aligned}$$

There are simplex-like algorithms to solve quadratic programs, but there are special algorithms to solve support vector machines.

Also, the feasible regions in the linear program are equivalent to the support vectors in the dual linear program.

6 Decision Theory

Our linear algorithms work well enough for “nice” data, but fails on real-world data. For example, we could have multiple sample points at the same Euclidean coordinate, but in different classes. To this end, we assume that the data $X \in \mathcal{X}$ comes from different but overlapping distributions, one per label $y \in \mathcal{Y}$, and attempt to determine the distributions, which gives us a probabilistic classifier.

Theorem 28

Let X and Y be two random variables. Then

$$p_{Y|X}(y | x) = \frac{p_{X|Y}(x | y)p_Y(y)}{p_X(x)}$$

where p_X is the density of X , and so on. The density $p_{Y|X}(y | x)$ is the *posterior probability* of y given x ; the density $p_Y(y)$ is the *prior probability* of y .

Definition 29 (Loss Function). A *loss function* $L(z, y)$ specifies the penalty to the model if the model predicts z and the true class is y .

Usually $L(x, x) = 0$.

Definition 30 (Symmetric Loss Function). A *symmetric loss function* L has that there exists ℓ such that $L(z, y) = \ell$ for every z and every $y \neq z$ (that is, the penalty for a wrong guess is constant).

The most popular classification function is the *zero-one loss function*, which has $L(z, y) = \mathbf{1}(z \neq y)$.

Definition 31 (Risk). Let $r: \mathcal{X} \rightarrow \mathcal{Y}$ be a decision rule. Then the *risk* for r is the expected loss over all values of X and Y :

$$R(r) = \mathbb{E}_X[L(r(X), Y)]$$

The *frequentist risk* is given by taking the expectation over the support of $p_{X,Y}(x, y)$, or \mathcal{D} :

$$R(r) = \mathbb{E}_X^{\mathcal{D}}[L(r(X), X)]$$

The *Bayesian risk* or *empirical risk* is given by taking the expectation over the data distribution, or $\mathcal{D}_{\text{data}}$:

$$\hat{R}(r) = \mathbb{E}_X^{\mathcal{D}_{\text{data}}}[L(r(X), X)]$$

Definition 32 (Bayes Decision Rule). The Bayes decision rule, or the Bayes classifier, is the function r^* such that $r^* = \operatorname{argmin}_r R(r)$. Indeed,

$$\begin{aligned} r^*(x) &= \operatorname{argmin}_y R(r(x) = y | x) \\ &= \operatorname{argmin}_y \int_z L(y, z) p_{Y|X}(z | x) dz \end{aligned}$$

When L is a symmetric loss function, we pick the class y with the biggest posterior probability, $p_{Y|X}(y | x)$.

The *Bayes classifier* has the optimal risk of any classifier.

Deriving and using r^* is called *risk minimization*. In the case where there are two classes $\{-1, 1\}$ and $L(y, y) = 0$, the Bayes classifier is

$$r^*(x) = 2 \operatorname{argmax} \left(L(1, -1) p_{Y|X}(-1 | x), L(-1, 1) p_{Y|X}(1 | x) \right) - 1$$

and

$$R(r^*) = \int_x \min_{y=\pm 1} L(-y, y) p_{X|Y}(x | y) p_Y(y) dx$$

If L is the zero-one loss, then

$$R(r) = \Pr[r(x) \neq y]$$

and the Bayes optimal decision boundary is

$$\left\{ x: \Pr[Y = 1 | X = x] = \frac{1}{2} \right\}.$$

Note that this is an isocontour.

There are three ways to build classifiers:

- *Generative models* (e.g. linear discriminant analysis)
 - We assume that the sample points come from some probability distribution, and each class has its own probability distribution.
 - Guess the form of distribution (i.e. put a prior distribution on each class).
 - For each class C , we fit the distribution parameters to class C points, giving $p_{X|Y}(x | C)$.

- We also estimate the prior distribution $p_Y(C)$.
- Bayes' Theorem gives $p_{Y|X}(C | x)$.
- If the loss is zero-one loss, we pick $C = \operatorname{argmax}_c p_{Y|X}(c | x) = \operatorname{argmax}_c p_{Y|X}(c | x)p_Y(c)$
- *Discriminative models* (e.g. logistic regression)
 - Model $p_{Y|X}(C | x)$ directly.
- Find the decision boundary (e.g. SVM)
 - Model $r(x)$ directly (no posterior distribution).

The advantage of generative and discriminative models is that $p_{Y|X}(C | x)$ tells us the probability that we are correct, which can be useful. Generative models give us a great way to diagnose outliers, with $p_X(x) \ll 1$ being an indicator of the outlier. However, it's often really hard to estimate distributions accurately, and real distributions rarely match the standard distributions we impose as priors.

7 Discriminant Analysis

Gaussian Discriminant Analysis

In this section, we assume that each class is distributed normally, and the distribution for each class C is different.

The density for the multivariate Gaussian is as follows. If $X | Y \sim \text{Normal}(\mu_Y, \Sigma_Y)$, then

$$p_{X|Y}(x | C) = \frac{1}{(\sqrt{2\pi} \det(\Sigma_C))^d} \exp\left(-\frac{1}{2} (x - \mu_C)^\top \Sigma_C^{-1} (x - \mu_C)\right)$$

If $\Sigma_C = \sigma_C^2 I$ (an *isotropic* Gaussian), then

$$p_{X|Y}(x | C) = \left(\sqrt{2\pi\sigma_C^2}\right)^{-d} \exp\left(-\frac{\|x - \mu_C\|^2}{2\sigma_C^2}\right)$$

For each class $C \in \mathcal{Y}$, suppose we estimate the mean $\hat{\mu}_C = \operatorname{Ex}_{\mathcal{D}_{\text{data}}}[X]$, variance $\hat{\sigma}_C^2 = \operatorname{Ex}_{\mathcal{D}_{\text{data}}}[x^2]$, and prior $\hat{\pi}_C = p_Y(C)$. Also assume that the risk function is symmetric. Then the Bayes decision rule gives

$$\begin{aligned} r^*(x) &= \operatorname{argmax}_{C \in \mathcal{Y}} \frac{p_{X|Y}(x | C) \hat{\pi}_C}{p_Y(C)} \\ &= \operatorname{argmax}_{C \in \mathcal{Y}} p_{X|Y}(x | C) \hat{\pi}_C \\ &= \operatorname{argmax}_{C \in \mathcal{Y}} \log(p_{X|Y}(x | C)) + \log(\hat{\pi}_C) \\ &= \operatorname{argmax}_{C \in \mathcal{Y}} -\frac{\|x - \mu_C\|_2^2}{2\sigma_C^2} - d \log(\sigma_C) + \log(\hat{\pi}_C) \end{aligned}$$

Suppose there are only two classes $\mathcal{Y} = \{C, D\}$. Then if we define

$$Q_A(x) = -\frac{\|x - \mu_A\|_2^2}{2\sigma_A^2} - d \log(\sigma_A) + \log(\hat{\pi}_A)$$

the Bayes classifier is

$$r^*(x) = \begin{cases} C, & Q_C(x) > Q_D(x) \\ D, & Q_C(x) < Q_D(x) \end{cases}$$

To recover the posterior probabilities, we use Bayes theorem. In particular, we have

$$p_{Y|X}(C | x) = \frac{p_{X|Y}(x | C)p_Y(C)}{\sum_{A \in \mathcal{Y}} p_{X|Y}(x | A)p_Y(A)} = \frac{e^{Q_C(x)}/\sqrt{2\pi^d}}{\sum_{A \in \mathcal{Y}} e^{Q_A(x)}/\sqrt{2\pi^d}} = \frac{e^{Q_C(x)}}{\sum_{A \in \mathcal{Y}} e^{Q_A(x)}}$$

Again let $\mathcal{Y} = \{C, D\}$. Then

$$p_{Y|X}(C | x) = \frac{e^{Q_C(x)}}{e^{Q_C(x)} + e^{Q_D(x)}} = \frac{1}{1 + e^{Q_D(x) - Q_C(x)}} = \sigma(Q_C(x) - Q_D(x))$$

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

Linear Discriminant Analysis

The linear discriminant analysis is a variant of Gaussian discriminant analysis. The additional assumption is that all the class Gaussians have the same variance σ^2 . In this case it's harder to overfit because there are fewer parameters. Indeed, in the two-class case,

$$Q_C(x) - Q_D(x) = \underbrace{\frac{\langle \mu_C - \mu_D | x \rangle}{\sigma^2}}_{\langle w | x \rangle} - \underbrace{\frac{\|\mu_C\|^2 - \|\mu_D\|^2}{2\sigma^2} + \log(\hat{\pi}_C) - \log(\hat{\pi}_D)}_{\alpha}$$

Now it's a linear classifier. The linear discriminant function is

$$L_C(x) = \frac{\langle \mu_C | \sigma^2 \rangle}{\sigma^2} - \frac{\|\mu_C\|^2}{2\sigma^2} + \log(\hat{\pi}_C)$$

Then the classifier is $r(x) = \operatorname{argmax}_{C \in \mathcal{Y}} L_C(x)$. In the two-class case, the decision boundary is $\langle w | x \rangle + \alpha = 0$ and the posterior distribution is $p_{Y|X}(C | x) = \sigma(\langle w | x \rangle + \alpha)$.

The linear discriminant analysis has a geometric description in the form of Voronoi diagrams; a point is classified to the closest μ_C for $C \in \mathcal{Y}$. Also, in the two-class case, if $\hat{\pi}_C = \hat{\pi}_D = \frac{1}{2}$, then by simple algebraic manipulation the linear discriminant rule degenerates to the centroid classifier (Section 3).

It's important to know that $e^{Q_C(x)} \propto p_{X,Y}(x, C)$, and in a sense we're picking the most likely class for the data x .

8 Parameter Estimation

We use maximum likelihood estimation for parameters.

Definition 33 (Maximum Likelihood Estimation). Given a process, we want to estimate a parameter Y given an outcome $X = x$. Then $\text{MLE}(Y | X) = \operatorname{argmax}_y p_{X|Y}(X | y)$.

Since the density is smooth, it's usually easy to optimize $p_{X|Y}$ via calculus methods.

Example 34 (Gaussian Fitting). Given sample points X_1, \dots, X_n , we want to find the best fit normal distribution. Then

$$\begin{aligned} \text{MLE}(\mu, \sigma | X_1, \dots, X_n) &= \operatorname{argmax}_{\mu, \sigma} p_{X_1, \dots, X_n | \mu, \sigma^2}(X_1, \dots, X_n | \mu, \sigma^2) \\ &= \operatorname{argmax}_{\mu, \sigma} \prod_{i=1}^n p_{X_i | \mu, \sigma^2}(X_i | \mu, \sigma^2) \end{aligned}$$

$$\begin{aligned}
&= \operatorname{argmax}_{\mu, \sigma} \sum_{i=1}^n \log(p_{X_i|\mu, \sigma^2}(X_i | \mu, \sigma^2)) \\
&= \sum_{i=1}^n \left(-\frac{\|X_i - \mu\|_2^2}{2\sigma^2} - d \log(\sqrt{2\pi}) - d \log(\sigma) \right)
\end{aligned}$$

We want to set $\nabla_{\mu} \text{MLE}(\mu, \sigma^2 | X_1, \dots, X_n) = 0$ and $\nabla_{\sigma} \frac{\partial \text{MLE}(\mu, \sigma^2 | X_1, \dots, X_n)}{\partial \sigma^2} = 0$. From the first equation we have

$$\begin{aligned}
\nabla_{\mu} \text{MLE}(\mu, \sigma^2 | X_1, \dots, X_n) &= \sum_{i=1}^n \frac{X_i - \mu}{\sigma^2} \\
&\stackrel{\text{set}}{=} 0 \\
\hat{\mu} &= \frac{1}{n} \sum_{i=1}^n X_i
\end{aligned}$$

and from the second equation we have

$$\begin{aligned}
\frac{\partial \text{MLE}(\mu, \sigma^2 | X_1, \dots, X_n)}{\partial \sigma} &= \sum_{i=1}^n \frac{\|X_i - \mu\|_2^2 - d\sigma^2}{\sigma^3} \\
&\stackrel{\text{set}}{=} 0 \\
\hat{\sigma}^2 &= \frac{1}{dn} \sum_{i=1}^n \|X_i - \mu\|_2^2
\end{aligned}$$

We don't know μ exactly, so in practice we substitute $\hat{\mu}$.

For QDA, we estimate the conditional mean $\hat{\mu}_C$ and conditional variance $\hat{\sigma}_C^2$ for each class C separately, and estimate the prior $\hat{\pi}_C = \frac{|\{Y_i: Y_i=C\}|}{n}$.

For LDA, we use the same means and priors as QDA, but we estimate the one variance for all classes:

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{C \in \mathcal{Y}} \sum_{\{i: Y_i=C\}} \|X_i - \hat{\mu}_C\|_2^2$$

when the inner sum is a constant multiple of the pooled within-class variance.

9 Anisotropic Multivariate Gaussians

Some linear algebraic facts to know:

- If v is an eigenvector of A with eigenvalue λ , then v is an eigenvector of A^k with eigenvalue λ^k .
- If A is invertible and v is an eigenvector of A with eigenvalue λ , then v is an eigenvector of A^{-1} with eigenvalue λ^{-1} .
- Every real, symmetric $n \times n$ matrix has real eigenvalues and n eigenvectors that are mutually orthogonal (this is the Spectral Theorem). In general a matrix can have more than n eigenvectors if its eigenspaces are not orthogonal and there exists a plane of eigenvectors in their intersection.

One of the more useful ways to visualize a symmetric matrix is via quadratic forms. Take the function $f(z) = \|z\|_2^2 = z^T z$; this is quadratic and isotropic, where the isosurfaces are hyperspheres. More to the point, the function $f(x) = x^T A^{-2} x = \|A^{-1}x\|_2^2$ is the quadratic form of the matrix A^{-2} (assuming that A is symmetric). This quadratic form is anisotropic, and the isosurfaces are ellipsoids. In particular, the

canonical isosurface representation of the quadratic form is $\|A^{-1}x\|_2^2 = 1$. The axes of this quadratic form are the eigenvectors of A , and the lengths of the axes are the corresponding eigenvalues of A (because if v_i axis has length λ_i , $\|A^{-1}v_i\|_2^2 = \left\|\frac{1}{\lambda_i}v_i\right\|_2^2 = 1$, so v_i lies on the ellipsoid). The sign of the product of the eigenvalues determines the handedness of the coordinate system. Correspondingly, if A is diagonal, then the eigenvectors are the coordinate axes (and the isosurface ellipsoids are axis aligned).

Recall that if A is symmetric and has all positive eigenvalues then A is *positive definite*, and $x^\top Mx > 0$ for all $x \neq 0$. If A has non-negative eigenvalues then A is *positive semi-definite*, and $x^\top Ax \geq 0$ for all x . If A has at least one positive and at least one negative eigenvalue, then A is *indefinite*. If A has no zero eigenvalue then it is invertible in any case.

Quadratic forms $f(x) = x^\top Ax$ with A positive definite have one minimum x^* ; if A is positive semidefinite then there are infinitely many minima which lie on an affine hyperplane with number of parameters equal to the number of indices i such that $\lambda_i = 0$; if A is indefinite then the quadratic form has no minimum value (and can grow arbitrarily negative).

We've been dealing with the quadratic form $f(x) = x^\top Mx$ (with M some PD/PSD/ID matrix) instead of $f(x) = x^\top A^{-2}x$, which generates the ellipse as an isocontour. It should be easy to see that if the original quadratic form is $x^\top Mx$ then $M = A^{-2}$ and $A = M^{-1/2}$, which means that M has the same eigenvectors as A so $v_i(A) = v_i(M)$ but $\lambda_i(A) = \frac{1}{\sqrt{\lambda_i(M)}}$. Therefore M is PD/PSD/ID if and only if A is PD/PSD/ID respectively.

We now cover eigendecomposition of symmetric A , which is trivial so I won't write it here.

Given a symmetric PSD matrix $\Sigma = U\Lambda U^\top$, we can find the symmetric square root $\Sigma^{1/2} = U\Lambda^{1/2}U^\top$.

Finally, we begin discussing anisotropic Gaussians. Let $X \sim \text{Normal}(\mu, \Sigma)$ have an anisotropic distribution over \mathbb{R}^d . Then

$$p_X(x) = \frac{1}{(\sqrt{2\pi})^d \sqrt{\det(\Sigma)}} \exp\left(-\frac{(x - \mu)^\top \Sigma^{-1} (x - \mu)}{2}\right)$$

where Σ is the $d \times d$ symmetric positive definite covariance matrix. Correspondingly, Σ^{-1} is the $d \times d$ symmetric positive definite *precision* matrix. Let $q(x) = (x - \mu)^\top \Sigma^{-1} (x - \mu)$; we understand this function, since it is a quadratic form. The isosurfaces of $q(x)$ are the same as $p_X(x)$ since the rest of the function is monotonic $\mathbb{R} \rightarrow \mathbb{R}$.

The covariance matrix Σ has eigenvalues that are the variances along the eigenvectors. The square root $\Sigma^{1/2}$ maps spheres to ellipsoids; the eigenvalues of $\Sigma^{1/2}$ are Gaussian widths/ellipsoid axis radii/standard deviations along the axes.

Since $\Sigma = U\Lambda U^\top$, $\Sigma^{-1} = U\Lambda^{-1}U$.

We're dealing with anisotropic Gaussians, so the parameter estimation might be different.

For quadratic discriminant analysis, we have that the estimated covariance for class C is

$$\hat{\Sigma}_C = \frac{1}{|\{i: Y_i = C\}|} \sum_{i: Y_i = C} (X_i - \hat{\mu}_C)(X_i - \hat{\mu}_C)^\top = \frac{(X_C - 1\mu_C^\top)(X_C - 1\mu_C^\top)^\top}{|\{i: Y_i = C\}|}$$

where X_C is the matrix of data points that are class C . For completeness,

$$\hat{\mu}_C = \frac{1}{|\{i: Y_i = C\}|} \sum_{i: Y_i = C} X_i$$

and

$$\hat{\pi}_C = \frac{|\{i: Y_i = C\}|}{n}$$

For linear discriminant analysis, we have that the estimated total covariance is just

$$\hat{\Sigma} = \frac{1}{n} \sum_{C \in \mathcal{Y}} \sum_{i: y_i = C} (X_i - \hat{\mu}_C) (X_i - \hat{\mu}_C)^\top = \frac{1}{n} \sum_{C \in \mathcal{Y}} (X_C - 1\hat{\mu}_C^\top) (X_C - 1\hat{\mu}_C^\top)^\top$$

which is the *pooled-within-class* covariance matrix.

Choosing the C that maximizes the estimated $p_{X,Y}(x, C)$ is equivalent to maximizing the *quadratic discriminant function*

$$Q_C(x) = \log \left(\left(\sqrt{2\pi} \right)^d p_{X|Y}(x|C) \hat{\pi}_C \right) = -\frac{1}{2} (x - \hat{\mu}_C)^\top \hat{\Sigma}_C^{-1} (x - \hat{\mu}_C) - \frac{1}{2} \log(\det(\hat{\Sigma}_C)) + \log(\hat{\pi}_C)$$

In the case of $|\mathcal{Y}| = 2$ the Bayes decision boundary is a quadric.

In the case of linear discriminant analysis where the Gaussians have the same covariance, the *linear discriminant function* is

$$L_C(x) = \hat{\mu}_C^\top \hat{\Sigma}^{-1} x - \frac{1}{2} \hat{\mu}_C^\top \hat{\Sigma}^{-1} \hat{\mu}_C + \log(\pi_C)$$

Sometimes LDA (in a two-class classification scheme) is interpreted as projecting points onto the normal w , where $w^\top = (\hat{\mu}_C - \hat{\mu}_D)^\top \Sigma^{-1}$.

LDA only has $d + 1$ parameters (in the two-class case), whereas QDA has $\frac{d(d+3)}{2} + 1$ parameters. QDA is more likely to overfit.

10 Regression

In classification, given a point X , we want to predict a class (often a binary decision problem). In regression, given a point X , we wish to predict a numerical value. We want to choose a form of the regression function $h_w(x)$, where h is a hypothesis on the regression function form, parameterized by w . This regression function is like a decision function in classification. We choose a cost function (objective function) to optimize; usually this is a sum of loss functions that evaluate the loss at one data point at a time.

Some regression functions are:

- Linear regression: $h_w(x) = w^\top x$ (x is usually augmented with a 1 vector to find an affine regressor).
- Polynomial regression, which is linear regression with polynomial features.
- Logistic regression: $h_w = \sigma(w^\top x)$, where $\sigma(x) = \frac{1}{1+e^{-x}}$, and x is usually augmented.

Let y be the true label for point x . Some loss functions are:

- Squared error: $L(h_w(x), y) = (h_w(x) - y)^2$.
- Absolute error: $L(h_w(x), y) = |h_w(x) - y|$.
- Logistic error, or cross-entropy: $L(h_w(x), y) = -y \log(h_w(x)) - (1 - y) \log(1 - h_w(x))$.

Some cost functionals are

- Mean loss: $J(h_w) = \frac{1}{n} \sum_{i=1}^n L(h_w(X_i), y_i)$.
- Maximum loss: $J(h_w) = \max_{1 \leq i \leq n} L(h_w(X_i), y_i)$.

- Weighted loss: $J(h_w) = \sum_{i=1}^n w_i L(h_w(X_i), y_i)$.
- ℓ^2 -regularized mean loss: $J(h_w) = \lambda \|w\|_2^2 + \frac{1}{n} \sum_{i=1}^n L(h_w(X_i), y_i)$
- ℓ^1 -regularized mean loss: $J(h_w) = \lambda \|w\|_1 + \frac{1}{n} \sum_{i=1}^n L(h_w(X_i), y_i)$

Some famous regression methods are:

- Least-squares linear regression: linear regression function, squared error, mean loss.
- Weighted least-squares linear regression: linear regression function, squared error, weighted sum loss.
- Ridge regression: linear regression function, squared error, ℓ^2 -regularized mean loss.
- LASSO regression: linear regression function, squared error, ℓ^1 -regularized mean loss.
- Logistic regression: logistic regression function, cross-entropy error, mean loss.
- Least absolute deviation regression: linear regression function, absolute error, mean loss.
- Chebyshev criterion regression: linear regression function, absolute error, maximum loss.

The first three regression methods (least-squares linear regression, weighted least-squares linear regression, and ridge regression) give convex quadratic cost functions, and closed form solutions are possible. LASSO regression is often framed as a quadratic program, usually with an exponential number of constraints, and solved using those methods. Logistic regression has a convex cost function, and we minimize it via gradient descent. Least absolute deviation regression and Chebyshev criterion regression are formulated as linear programs and solved that way.

Linear Regression

Least-squares linear regression uses a linear regression function, squared loss function, and a mean loss cost functional. Without augmenting the data matrix X , the optimization problem is

$$\begin{aligned}
 (w_1^*, w_2^*) &= \underset{w_1, w_2}{\operatorname{argmin}} \sum_{i=1}^n L(h_{w_1, w_2}(X_i), Y_i) \\
 &= \underset{w_1, w_2}{\operatorname{argmin}} \sum_{i=1}^n (w_1^\top X_i + w_2 - Y_i)^2 \\
 &= \underset{w_1, w_2}{\operatorname{argmin}} \sum_{i=1}^n \left(\begin{bmatrix} w_1^\top & w_2^\top \end{bmatrix} \begin{bmatrix} X_i \\ 1 \end{bmatrix} \right)^2 \\
 &= \underset{w_1, w_2}{\operatorname{argmin}} \left\| \begin{bmatrix} X & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - Y \right\|_2^2 \\
 &\stackrel{\text{set}}{=} \underset{w_1, w_2}{\operatorname{argmin}} \operatorname{RSS} \left(\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right)
 \end{aligned}$$

Define $w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ and augment X , so $X = \begin{bmatrix} X & 1 \end{bmatrix}$. Then

$$\begin{aligned}
 w^* &= \underset{w}{\operatorname{argmin}} \operatorname{RSS}(w) \\
 \nabla_w \operatorname{RSS}(w) &= 2X^\top X w - 2X^\top Y \\
 &\stackrel{\text{set}}{=} 0 \\
 X^\top X w^* &= X^\top Y
 \end{aligned}$$

If $X^T X$ is singular, then the problem is underconstrained, so we can use Cholesky decomposition, LU decomposition, conjugate gradient method, etc. that solves for w . Otherwise, we can do $w^* = (X^T X)^{-1} X^T Y$. If X is very large, we can solve the implied linear system instead of worrying about the inverse. The term $X^+ = (X^T X)^{-1} X^T$ is the *left pseudoinverse* of X , because $X^+ X = I$ when X is full rank. The matrix $H = X X^+$ is called the *hat matrix*, because $\hat{Y} = X X^+ Y = H Y$. If $n > d + 1$ then H is singular.

The matrix $H = X X^+$ can be interpreted as an orthogonal projection from \mathbb{R}^n onto \mathbb{R}^{d+1} , giving the normal equations $X^T (X w - Y) = 0$.

Logistic Regression

Logistic regression has the regression function $h_w(x) = \sigma(w^T x)$. It fits probabilities in the range $(0, 1)$, so it's used to fit a distribution. Usually it's used for classification; the input Y_i 's can be probabilities, but usually they're either 0 or 1.

Using the augmented X and w , the optimization problem is

$$\begin{aligned} w^* &= \underset{w}{\operatorname{argmin}} J(w) \\ &= - \underset{w}{\operatorname{argmin}} \sum_{i=1}^n \left[Y_i \log(h_w(X_i)) + (1 - Y_i) \log(1 - h_w(X_i)) \right] \end{aligned}$$

This cost function is convex, so we can solve it via gradient descent. First, note that $\nabla_x \sigma(x) = \sigma(x)(1 - \sigma(x))$. Then

$$\begin{aligned} \nabla_w \sum_{i=1}^n Y_i \log(h_w(X_i)) + (1 - Y_i) \log(h_w(X_i)) &= \sum_{i=1}^n y_i \log(\sigma(w^T X_i)) + (1 - Y_i) \log(\sigma(w^T X_i)) \\ &= \sum_{i=1}^n \left(\frac{Y_i}{\sigma(w^T X_i)} - \frac{1 - Y_i}{1 - \sigma(w^T X_i)} \right) \sigma(w^T X_i) (1 - \sigma(w^T X_i)) X_i \\ &= \sum_{i=1}^n (Y_i - \sigma(w^T X_i)) X_i \\ &= X^T (Y - \sigma(Xw)) \quad (\sigma(Xw) \text{ is componentwise.}) \end{aligned}$$

The gradient descent rule is $w \leftarrow w + \varepsilon X^T (Y - \sigma(Xw))$.

If we want to incorporate polynomial features, we can add columns as features that are polynomial functions of the original features. We can also add non-polynomial features in this matter. Otherwise, the process is just like linear or logistic regression. Logistic regression and quadratic features in fact models quadratic discriminant analysis. Higher degree polynomials have very high possibility to overfit (because of the VC-dimension, but of course that won't be mentioned).

Another modification to least-squares regression is to weight each sample point, in weighted least-squares regression. We assign each sample point X_i a weight ω_i and collect them in $\Omega = \operatorname{diag}(\omega) \in \mathbb{R}^{n \times n}$. A greater value of ω_i biases $(\hat{Y}_i - Y_i)^2$ to be smaller, where $\hat{Y} = Xw$. The optimization problem is

$$w^* = \min_w (Xw - Y)^T \Omega (Xw - Y) = \sum_{i=1}^n \omega_i (X_i \cdot w - Y_i)^2$$

and the normal equations are $X^T \Omega X w = X^T \Omega Y$.

Newton's Method

Newton's method is an iterative optimization method for a C^2 (twice differentiable) function $J(w)$ that works well for logistic regression, and in low-dimensional space works much faster than gradient descent. The basic idea is to approximate $J(w)$ near w by a quadratic function, then jump to its unique critical point. We use the Taylor quadratic approximation:

$$\nabla_w J(w) = \nabla_v J(v) + \left(\nabla_v^2 J(v) \right) (w - v) + o(\|w - v\|_2^2)$$

Finding the critical point w sets $\nabla_w J(w) = 0$, so $w = v - \left(\nabla_v^2 J(v) \right)^{-1} (\nabla_v J(v))$

Algorithm 5 Newton's method.

Input: A convex twice-differentiable function $f(w): \mathbb{R}^d \rightarrow \mathbb{R}$ to be minimized.

Output: A minimizer w^* of f .

$w \leftarrow$ randomly initialized weight vector.

while $\nabla_w J(w) \neq 0$ **do**

$w \leftarrow w - \left(\nabla_w^2 J(w) \right)^{-1} (\nabla_w J(w))$

return w

Alternatively, if $\nabla_w^2 J(w)$ is not invertible, then we have

Algorithm 6 Non-invertible Newton's method.

Input: A convex twice-differentiable function $f(w): \mathbb{R}^d \rightarrow \mathbb{R}$ to be minimized.

Output: A minimizer w^* of f .

$w \leftarrow$ randomly initialized weight vector.

while $\nabla_w J(w) \neq 0$ **do**

$e \leftarrow$ a "good" solution to the linear system $\left(\nabla_w^2 J(w) \right) e = -\nabla_w J(w)$

$w \leftarrow w + e$

return w

The Newton's method doesn't know the difference between minima, maxima, and saddle points. The caveat is that the starting point must be "close enough" to the desired critical point.

Newton's method works specifically well for logistic regression. Recall that $\sigma(\gamma) = \sigma(\gamma) (1 - \sigma(\gamma))$ and

define for convenience $s_i = \sigma(X_i^\top w)$ and $s = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$. Then $\nabla_w J(w) = -\sum_{i=1}^n (Y_i - s_i) X_i = -X^\top (Y - s)$

and $\nabla_w^2 J(w) = \sum_{i=1}^n s_i (1 - s_i) X_i X_i^\top = X^\top \Omega X$ where $\Omega = \text{diag}(s \otimes (1 - s))$. Clearly Ω is positive definite, so $X^\top \Omega X$ is positive semidefinite, so $J(w)$ is convex. This problem is weighted least squares, but since Ω changes on every iteration, this problem is iteratively weighted least squares. This penalizes points with $s_i \approx \frac{1}{2}$ and ignores points near 0 or 1. By contrast, even though the posteriors of discriminant analysis (LDA, QDA) are themselves logistic functions, LDA and QDA weight each point equally.

It's worthwhile to compare the two methods.

- Advantages of LDA:

- For well-separated classes, LDA is stable; logistic regression is surprisingly unstable.
- LDA and QDA extends easily and elegantly to multiple classes; logistic regression needs modifying (softmax regression).
- LDA is more accurate when the conditional densities of the data are normal, especially if n is small.
- Advantages of logistic regression:
 - It puts more emphasis on the decision boundary, and creates the decision boundary around the variances in the points near to it. Therefore it's less sensitive to outliers far from the decision boundary.
 - It's more robust on some non-Gaussian distribution (one with skew).
 - Naturally fits labels between 0 and 1.

Normally, our binary classification boundary is where the probability of a point on the boundary being in either class is $\frac{1}{2}$. As we vary that, we can obtain new, possibly better, classifiers. In binary classification problems, the ROC curve plots the false positive rate $FP = \frac{|\{y_i: y_i=-1, f(X_i)=-1\}|}{|\{y_i: y_i=-1\}|}$ on the x -axis, against the true positive rate $TP = \frac{|\{y_i: y_i=1, f(X_i)=1\}|}{|\{y_i: y_i=1\}|}$. The false negative rate is $FN = 1 - TP$ and the true negative rate is $FN = 1 - FP$. The sensitivity is the true positive rate and the false negative rate is the specificity. The points $(0,0)$ and $(1,1)$ are on the curve, representing the classifier that classifies any point as -1 and the classifier that classifies any point as 1 . Any point (x,x) on the straight line line between $(0,0)$ and $(1,1)$ represents a classifier that randomly classes the proportion x of sample points as 1 and the rest -1 . A good metric of the efficacy of your classifier is the area under the ROC curve.

Bias-Variance Decomposition

A typical data distribution model is that

- The sample data pairs are drawn from $X_i \sim \mathcal{D}$, for \mathcal{D} a distribution.
- The Y -values are the sum of an unknown, non-random function, and random noise:

$$Y_i = g(X_i) + \varepsilon_i$$

where $g: \mathbb{R}^d \rightarrow \mathbb{R}$ and $\varepsilon_i \sim \mathcal{N}$, a noise distribution with 0 expectation.

The goal of regression is to find the h that best estimates g . The ideal approach is $h(x) = \text{Ex}_{\mathcal{D}}[Y | X = x]$; this estimator is unbiased, since

$$\begin{aligned} h(X) &= \text{E}_{\mathcal{D}}[Y | X = x] \\ &= \text{E}_{\mathcal{D}}[g(X_i) + \varepsilon | X = x] \\ &= \text{E}_{\mathcal{D}}[g(X_i) | X = x] + \text{E}_{\mathcal{D}}[\varepsilon | X = x] \\ &= g(X) \end{aligned}$$

Of course, we don't have \mathcal{D} to take the expectation over.

Suppose $\mathcal{N} = \text{Normal}(0, \sigma^2)$, i.e. our noises are normally distributed. Then $Y_i | X_i \sim \text{Normal}(g(X_i), \sigma^2)$. Then

$$p_{Y|X}(y | x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - g(x_i))^2}{2\sigma^2}\right)$$

$$\log(p_{Y|X}(y|x)) = -\frac{(y_i - g(x_i))^2}{2\sigma^2} - C \quad (C \text{ is some constant.})$$

$$\begin{aligned} \ell(g | (X_i, Y_i)_{i=1}^n) &= \log\left(\prod_{i=1}^n p_{Y|X}(y_i | x_i)\right) \\ &= \sum_{i=1}^n \log(p_{Y|X}(y_i | x_i)) \\ &= -\frac{1}{2\sigma^2} \sum_{i=1}^n (y_i - g(x_i))^2 - C' \end{aligned}$$

where C and C' are some constants that are irrelevant in the optimization problems that follow. The takeaway is that assuming normal noise, the maximum likelihood estimator h of g is a linear function.

Logistic regression also has a motivation via maximum likelihood. Indeed, imagine that each X_i has β copies, where $Y_i\beta$ copies are in class C and $(1 - Y_i)\beta$ copies are in class D . Then the likelihood is

$$\mathcal{L}(h; (X_i, Y_i)_{i=1}^n) = \prod_{i=1}^n h(X_i)^{Y_i\beta} (1 - h(X_i))^{(1-Y_i)\beta}$$

and the log-likelihood is

$$\begin{aligned} \ell(h) &= \log(\mathcal{L}(h; (X_i, Y_i)_{i=1}^n)) \\ &= \beta \sum_{i=1}^n \left[Y_i \log(h(X_i)) + (1 - Y_i) \log(1 - h(X_i)) \right] \\ &= -\beta \sum_{i=1}^n L(h(X_i), Y_i) \quad (\text{Logistic loss function.}) \\ \max_h \ell(h) &= \max_h \sum_{i=1}^n L(h(X_i), Y_i) \end{aligned}$$

There are two sources of error in a hypothesis h :

- *bias*: the error due to inability of hypothesis h to fit g perfectly, $\mathbb{E}_{\mathcal{D}}[h(X)] - g(X)$.
- *variance*: the error due to fitting random noise in the data, $\mathbb{E}_{\mathcal{D}}[(g(X) - h(X))^2]$.

Indeed, take some point $z \in \mathbb{R}^d$ and draw ε from \mathcal{N} at random. Define $\gamma = h(z) + \varepsilon$. Then $\mathbb{E}_{\mathcal{D}}[\gamma] = \mathbb{E}_{\mathcal{D}}[h(z)]$ and $\text{Var}_{\mathcal{D}}[\gamma] = \text{Var}_{\mathcal{D}}[\varepsilon]$.

Then

$$\begin{aligned} R(h) &= \mathbb{E}_{\mathcal{D}, \gamma}[L(h(z), \gamma)] \\ &= \mathbb{E}_{\mathcal{D}, \gamma}[(h(z) - \gamma)^2] \\ &= \mathbb{E}_{\mathcal{D}}[h(z)^2] + \mathbb{E}_{\mathcal{D}, \gamma}[\gamma^2] - 2 \mathbb{E}_{\mathcal{D}, \gamma}[\gamma h(z)] \\ &= \text{Var}_{\mathcal{D}}[h(z)] + \mathbb{E}_{\mathcal{D}}[h(z)]^2 + \text{Var}_{\mathcal{D}, \gamma}[\gamma] + \mathbb{E}_{\mathcal{D}, \gamma}[\gamma]^2 - 2 \mathbb{E}_{\mathcal{D}, \gamma}[\gamma] \mathbb{E}_{\mathcal{D}}[h(z)] \\ &= \left(\mathbb{E}_{\mathcal{D}}[h(z)] - \mathbb{E}_{\mathcal{D}, \gamma}[\gamma] \right)^2 + \text{Var}_{\mathcal{D}}[h(z)] + \text{Var}_{\mathcal{D}, \gamma}[\gamma] \end{aligned}$$

$$\begin{aligned}
&= \left(\mathbb{E}_{\mathcal{D}}[h(z)] - g(z) \right)^2 + \text{Var}_{\mathcal{D}}[h(z)] + \text{Var}[\varepsilon] \\
&= \underbrace{\text{Bias}(h)^2}_{\text{squared bias}} + \underbrace{\text{Var}_{\mathcal{D}}[h(z)]}_{\text{variance}} + \underbrace{\text{Var}[\varepsilon]}_{\text{irreducible error}}
\end{aligned}$$

This is the pointwise version of the bias-variance decomposition. The general version takes $z \sim \mathcal{D}$ and takes the expectation over z , obtaining the expected squared bias and the expected variance. Some implications are:

- Underfitting indicates a lot of bias.
- Overfitting is caused by high variance.
- Training error reflects the bias but not the variance; the test error reflects both.
- For many distributions, variance goes to 0 as $n \rightarrow \infty$.
- If h can fit g exactly, for many “nice” distributions the bias goes to 0 as $n \rightarrow \infty$.
- If h cannot fit g well, the bias is large at “most” points.
- Adding a good feature reduces bias; adding a bad feature rarely increases it.
- Adding a feature usually increases variance.
- It’s not possible to reduce the irreducible error.
- Noise in test sets affects only the irreducible error; noise in the training set affects only the bias and variance.
- For real-world data, g is rarely knowable.
- But we can test learning algorithms by choosing g and making synthetic data.

Ridge Regression

We want to cover ℓ^2 -regularization, which has the cost function:

$$w^* = \underset{w}{\operatorname{argmin}} \left(\|Xw - Y\|_2^2 + \lambda \|\text{debias}(w)\|_2^2 \right) = \underset{w}{\operatorname{argmin}} J(w)$$

where the “bias” of w is the coefficient of the “fictitious” dimension. In theory, the bias of w can be included in the cost function, so it becomes the more clean

$$w^* = \underset{w}{\operatorname{argmin}} \left(\|Xw - Y\|_2^2 + \lambda \|w\|_2^2 \right)$$

for which the closed form solution always exists and is

$$w^* = \left(X^T X + \lambda I \right)^{-1} X^T Y$$

Of course, the prediction function is $h_w(x) = x^T w$, so $h_{w^*}(x) = x^T \left(X^T X + \lambda I \right)^{-1} X^T Y$.

Regularization always provides well-defined solutions to normal equations because the matrix $X^T X + \lambda I \succ 0$ (is positive definite), and therefore always has an inverse, and it reduces variance by overfitting (by penalizing large values of $\|w\|_2^2$).

Another way to visualize the ridge regression problem is to find where the isocontour of $\|Xw - Y\|_2^2$ (an ellipsoid with radii corresponding to Σ_X^{-1}) and $\lambda \|w\|_2^2$ (a hypersphere with radius λ^{-1}) intersect in w -space.

If we assume that our data model is $Y = Xw + \varepsilon$, then $\text{Var}[h_w(x)] = \text{Var} \left[x^T \left(X^T X + \lambda I \right)^{-1} Y \right]$.

Note that in regularized models which have cost functions $J(w) = f(w) + r(w)$, where r is a regularization functional, the training optimizes $J(w)$ while the validation set is evaluated on the cost function $f(w)$.

Ideally, the features should also be “normalized” to have the same variance. A more generalized version is to use an asymmetric penalty by replacing $\|w\|_2^2$ with $\|P^{1/2}w\|_2^2$ for P a symmetric matrix indicating differing penalties on each feature.

If we say that $w \sim \text{Normal}(0, \sigma^2)$ and apply MLE to the posterior, we have

$$\begin{aligned} p_{W|X,Y}(w | X, Y) &= \frac{p_{X,Y|W}(X, Y | w)p_W(w)}{p_{X,Y}(X, Y)} \\ &\propto p_{X,Y|W}(X, Y | w)p_W(w) \\ w^* &= \underset{w}{\operatorname{argmax}} p_{W|X,Y}(w | X, Y)p_W(w) \\ &= \min_w \left(\|Xw - Y\|_2^2 + \lambda \|w\|_2^2 \right) \end{aligned}$$

Subset Selection

All features increase variance, but not all features reduce bias. We want to identify poorly predictive features and drop them. This leads to less overfitting, smaller test errors, and easier inference.

Sometimes this is hard; different features can partially encode some information, so it’s combinatorially hard to choose the best feature subset.

The simplest algorithm is *Best Subset Selection*, which tries all $2^d - 1$ nonempty subsets of features, choosing the best classifier by cross-validation.

Obviously for large d this is completely infeasible, so we use heuristics.

- Heuristic 1: Forward stepsize selection. We start with the null model and greedily pick the feature with the best predictive power (as determined by validation). This takes $\mathcal{O}(d^2)$ number of models.
- Heuristic 2: Backwards stepsize selection. We start with all d features and greedily drop features whose removal gives the best reduction in the validation error. This takes $\mathcal{O}(d^2)$ number of models.

LASSO Regression

LASSO regression is like ridge regression, except with an ℓ^1 -penalty:

$$w^* = \underset{w}{\operatorname{argmin}} \left(\|Xw - Y\|_2^2 + \lambda \|w\|_1 \right)$$

It naturally sets some weights to 0 in the process of optimizing the cost function (see the Homework 4 for details). The isosurfaces of $\|w\|_1$ are *cross-polytopes* (convex hulls of \pm unit vectors).

11 Decision Trees

The decision tree algorithm is a nonlinear method for classification and regression. The basic idea is that we construct a tree with 2 node types:

- Internal nodes test feature values and branch accordingly.
- Leaf nodes specify the label $h(x)$.

The decision tree splits \mathcal{X} into rectangular cells. It works well with both categorical and quantitative features, returns interpretable results, and can develop arbitrarily complicated decision boundaries.

The simplest method to build trees is a greedy, top-down learning heuristic. We present this method.

Algorithm 7 Decision tree training procedure.

Input: A set of sample points $(X_i, Y_i)_{i=1}^n$, a cost function on an index set $J(S)$.

Output: A decision tree trained on the sample points.

```

function GROWTREE( $S$ )
  if  $\exists y \in \mathcal{Y}$  such that  $Y_i = y$  for all  $i \in S$  then
    return LEAF( $y$ )
  else
    Choose best splitting feature  $j$  and splitting value  $\beta$  by minimizing  $J(S)$  among all splits.
     $S_l \leftarrow \{i: X_{i,j} < \beta\}$ 
     $S_r \leftarrow \{i: X_{i,j} \geq \beta\}$ 
    return INTERNALNODE( $j, \beta$ , GROWTREE( $S_l$ ), GROWTREE( $S_r$ ))
GROWTREE( $\{1, \dots, n\}$ )

```

We choose the best split by devising a cost function $J(S)$ for a set of indices S . Then we choose the split that minimizes the weighted average $\frac{|S_l|J(S_l) + |S_r|J(S_r)}{|S_l| + |S_r|}$.

We now find a good cost function $J(S)$. If we define the entropy of a set as

$$H(S) = - \sum_{C \in \mathcal{Y}} p_C \log_2(p_C), \quad p_C = \frac{|\{i \in S: Y_i = C\}|}{|S|}$$

then we seek the split that maximizes the information gain:

$$H_{\text{gain}} = H(S) - H_{\text{after}} = H(S) - \frac{|S_l|H(S_l) + |S_r|H(S_r)}{|S|}$$

which is the same as minimizing the term $\frac{|S_l|H(S_l) + |S_r|H(S_r)}{|S|}$, since $H(S)$ is fixed. So by pattern matching we find that $J(S) = H(S)$; the cost function of the set is the entropy of the set.

The entropy is not the only function that works well; many concave functions work fine, including quadratic polynomials.

There are some more technical details about choosing a split:

- For a binary feature j , the children are cases where $X_{i,j} = 0$ and $X_{i,j} = 1$; for a discrete feature with more than 2 values, the split depends on the application.
- If the feature j is quantitative, we sort X_i values in S by the values $X_{i,j}$, and we attempt to split between each pair of *unequal* consecutive values. By using prefix sums, we can update the entropy (cost) of the split in $\mathcal{O}(1)$ time per point.

Overall, the algorithm takes $\mathcal{O}(nd \cdot \text{depth of the tree})$ time; often the depth of the tree is logarithmic in n , providing a runtime of $\mathcal{O}(nd \log(n))$ which is quite reasonable.

Splits can be functions of multiple variables. We can construct non-axis-aligned splits using decision boundaries of other classification algorithms, or by generating them randomly. This may yield a better classifier at the cost of worse interpretability or speed. Standard decision trees are very fast because they

check only one feature at each treenode. But if there are hundreds of features, and you have to check all of them at every level of the tree to classify a point, it slows down classification a lot. So it sometimes pays to consider methods like forward stepwise selection when you're learning so that when you classify, you only need to check a few features at each tree node.

The regression decision tree specifies a piecewise constant regression function; the cost function is different; instead of using $J(S) = H(S)$, we can use $J(S) = \frac{1}{|S|} \sum_{i \in S} (Y_i - \mu_S)^2$, where $\mu_S = \frac{1}{|S|} \sum_{i \in S} Y_i$. Perhaps we can use other costs that correspond to the costs for more standard regression methods.

The basic version of the decision tree algorithm keeps subdividing treenodes until every leaf is pure. We don't have to do that; sometimes we prefer to stop subdividing tree nodes earlier. Stopping early limits tree depth (making the algorithm faster), limits tree size (making computing a tree on a big data set tractable), limits overfitting, and estimates posterior probabilities in place of binary variables for pure leaves. When we have overlapping class distributions, refining the tree down to one sample point per leaf is guaranteed to overfit. It's better to stop early, then classify each leaf node by taking a vote of its sample points. Alternatively, you can use the points to estimate a posterior probability for each leaf; if there are many points in each leaf, the posterior probabilities might be reasonably accurate.

Some ways to stop early are:

- Find if the next split doesn't reduce entropy/error enough (this is dangerous; pruning is better).
- Most of the node's points have the same class already; this deals well with outliers.
- The node contains very few sample points.
- The cell's edges are all tiny.
- The depth of the tree is too high.
- Use validation to compare whether we should stop or not.

The last is the slowest but most effective way to know when to stop: use validation to decide whether splitting the node is a win on the validation data. But if your goal is to avoid overfitting, it's generally even more effective to grow the tree a little too large and then use validation to prune it back. We'll talk about that next.

Leaves with multiple points return:

- for classification, a majority vote or posterior probabilities, or
- for regression, an average or posterior probabilities.

The idea of pruning is to intentionally grow the tree too large, then greedily remove each split whose removal improves validation performance. This is more reliable than stopping early.

We have to do validation once for each split that we're considering removing. But we can do that pretty cheaply. What we don't do is reclassify every sample point from scratch. Instead, we keep track of which points in the validation set end up at which leaf. When we decide whether to remove a split, we just look at the validation points in the two leaves we're thinking of removing, and see how they will be reclassified and how that will change the error rate. We can do this very quickly.

The reason why pruning often works better than stopping early is because often a split that doesn't seem to make much progress is followed by a split that makes a lot of progress. If we stop early, we'll never find out. Pruning is a simple idea, but it's highly recommended when you have enough time to build and prune the tree.

Decision trees are fast, simple, interpretable, easy to explain, invariant under scaling and transformation,

and robust to irrelevant features, but they aren't the best at prediction, and have high variance. We fix this by averaging the answer for a particular prediction over many decision trees, in a process called ensemble learning.

The main idea is that sometimes the average output of a bunch of crappy algorithms is better than the opinion of one strong algorithm. We call a learning algorithm a *weak learner* if there exists some $\delta > 0$ such that it is $\frac{1}{2} + \delta$ of the way to perfect performance (i.e. perfect regression or classification), given unlimited data. We call a learning algorithm a *strong learner* if for every $\varepsilon > 0$ the algorithm can get $1 - \varepsilon$ of the way to perfect performance, given unlimited training data. The goal is to combine many weak learners into a strong learner. We can take the average of the output of

- different learning algorithms;
- the same learning algorithm on many training sets;
- the same learning algorithm on many random subsamples of one training set (bagging);
- randomized decision trees on random subsamples (random forests).

The last two are the most common ways to use averaging, because usually we don't have enough training data to use fresh data for every learner. Averaging is not specific to decision trees; it can work with many different learning algorithms. But it works particularly well with decision trees.

To average effectively, we use learners with low bias (e.g., deep decision trees), but potentially high variance and some overfitting are okay, since different learners overfit in different ways. Averaging sometimes reduces the bias and increases the flexibility.

Bagging is a randomized method for creating many different learners from the same data set. It works well with many different learning algorithms. One exception seems to be k -nearest neighbors; bagging mildly degrades it. Given an n -point random subsample of size n' by sampling with replacement. Some points are chosen multiple times; some are not chosen. If $n' = n$ and both quantities are large, then $1 - \frac{1}{e}$ of the training samples are chosen in expectation. We build a learner using the random subsample, and repeat until we have T learners. The metalearner takes a test point, feeds it into all T learners, and returns the average/majority output.

Sometimes, random sampling isn't random enough. If there's one really strong predictor, for example, then the same feature split is at the top of every tree. If the trees are very similar, then taking their average doesn't reduce the variance much. Random forest algorithms fix that. We have many different trees, each with a corresponding subsample generated via bagging. At each tree node, we take the random sample of $m \leq d$ features; we choose the best split from m features ($m = \sqrt{d}$ works well for classification, and $m = \frac{d}{3}$ works well for regression). Each tree node has a different random sample. Smaller m lead to more randomness, less correlation between trees, but also more bias.