CS 189

Introduction to Machine Learning

Notes

Druv Pai

Contents

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 $f\colon \mathbb{R}^d \to \{-1, 1\}$, 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 continous 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.

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 $\{x \in \mathcal{X} \colon f(x) = 0\}$. 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 $\{x \in \mathcal{X} \colon f(x)\}$ is called an **isosurface** of f for the **isovalue** 0. In particular, f has other isosurfaces for other isovalues, e.g. $x \colon f(x) = 1$.

Sometimes there are needs for explicit nonzero isovalues.

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

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

Proof. By computation,

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

as desired. □

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, $\operatorname{sign}(w \cdot x + \alpha) > 0$ 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 $f(x)$.

$\mu_{C_1} \leftarrow \frac{1}{|C_1|} \sum_{Y_i = C_1} X_i$
$\mu_{C_2} \leftarrow \frac{1}{|C_2|} \sum_{Y_i = C_2} X_i$
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 an 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 $X_1, \ldots, X_n$. 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 $\{x \colon w \cdot x = 0\} \mapsto w$. Correspondingly, there is a natural injection from hyperplanes $H \in W$ and vectors $x \in V$ given by $\{w \colon x \cdot w = 0\} \mapsto x$. If vector $x \in \{z \colon w \cdot z = 0\} \subseteq V$, 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 size of $\{z \colon w \cdot z = 0\}$ as w, and
- in W, w should be on the same side of $\{z \colon x \cdot z = 0\}$ 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 $\operatorname{sign}(z) = \operatorname{sign}(Y_i)$, 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 $V = \{i \colon Y_i X_i \cdot w < 0\}$ is the set of indices of misclassified sample points.

If w classifies all $X_1, \ldots, X_n$ 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 $J(w)\colon \mathbb{R}^d \to \mathbb{R}$ 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 $j \in \{1, \ldots, n\}$.
  $V \leftarrow \{(X_i, Y_i)\colon Y_i X_i \cdot w < 0\}$
  while $|V| > 0$ **do**
    $(X_i, Y_i) \leftarrow$ entry of V
    $w \leftarrow w + \varepsilon \min(0, Y_i X_i)$
    $V \leftarrow \{(X_i, Y_i)\colon Y_i X_i \cdot w < 0\}$
  return $f(x) \stackrel{\text{def}}{=} \operatorname{sign}(w \cdot x)$

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 $i \in \{1, \ldots, n\}$. Since this constraint doesn't change when $(w, \alpha, c) \mapsto \left(\frac{w}{c}, \frac{\alpha}{c}, 1\right)$ (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 $H = \{x \colon w \cdot x = 0\}$ 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

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

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 $\nabla^2_{w,\xi} \left(\|w\|_2^2 + C \sum_{i=1}^n \xi_i \right) = \begin{bmatrix} 2I_d & 0_n \\ 0_n & 0_d \end{bmatrix}$ 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 $\Phi\colon \mathbb{R}^d \to \mathbb{R}^{d+1}$, 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

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

where $\begin{bmatrix} -2c^\mathsf{T} & 1 \end{bmatrix}$ 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 $\mathbb{R}^d$. □

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 $\Phi\colon \mathbb{R}^d \to \mathbb{R}^{2d}$ where $\Phi(X) = \begin{bmatrix} X_1^2 & \cdots & X_d^2 & X_1 & \cdots & X_d \end{bmatrix}^\mathsf{T}$. 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 $\Phi\colon \mathbb{R}^d \to \mathbb{R}^{(d^2+3d)/2}$ by $\Phi(x) = \begin{bmatrix} X_1^2 & \cdots & X_d^2 & X_1X_2 & \cdots & X_{d-1}X_d & \cdots & X_1 & \cdots & X_d \end{bmatrix}^\mathsf{T}$. 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 $\Phi(X) = \begin{bmatrix} X_1^p & \cdots & X_d^p & \cdots \prod_{i=1}^d X_i^{j_i} & \cdots & X_1 & \cdots & X_d \end{bmatrix}^\mathsf{T}$. This mapping is $\Phi\colon \mathbb{R}^d \to \mathbb{R}^{\mathcal{O}(d^p)}$. 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 $f\colon \mathbb{R}^d \to \mathbb{R}$ Shewchuk-smooth if $f \in C^1$ (its first derivative is continuous). With respect to a function $f \in \mathbb{R}^d \to \mathbb{R}$, call x a global minimum if $f(x) \leq f(y)$ for each $y \in \mathbb{R}^d$, and call x a global 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 $f\colon \mathbb{R}^d \to \mathbb{R}$ 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: algorithm ??
  - 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: algorithm ??
- 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: algorithm ??
  - 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 $f(w) = c^\mathsf{T} w$, 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

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

Each linear inequality $a_{:,i}^\mathsf{T} w \leq b_i$ 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 $\mathscr{F}$. The optimal $w \in \mathscr{F}$ is the $x \in \mathscr{F}$ 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:

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

In this example, every feasible point (w, α) gives a linear classifier. The data are linearly separable if and only if $\mathscr{F} \neq \emptyset$. 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

$$\text{minimize } x^\mathsf{T} Q x + c^\mathsf{T} x$$

$$\text{subject to } Ax \leq b$$

where Q is a symmetric positive definite matrix (a positive definite matrix has $x^\mathsf{T} Q x > 0$ 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:

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

and so is the soft-margin support vector machine:

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

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 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

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

where p_X is the density of X, and so on. The density $p_{Y|X}(y \mid x)$ 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 $L(z, y) = \mathbb{1}(z \neq y)$.

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

$$R(r) = \operatorname{Ex}[L(r(X), Y)]$$

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

$$R(r) = \operatorname*{Ex}_{\mathcal{D}}[L(r(X), X)]$$

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

$$\hat{R}(r) = \mathop{\text{Ex}}_{\mathcal{D}_{\text{data}}}[L(r(X), X)]$$

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

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

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

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

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

and

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

If L is the zero-one loss, then

$$R(r) = \Pr_{\mathcal{D}}[r(x) \neq y]$$

and the Bayes optimal decision boundary is

$$\left\{x \colon \Pr_{\mathcal{D}}[Y = 1 \mid X = x] = \frac{1}{2}\right\}.$$

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

The advantage of generative and discriminative models is that $p_{Y|X}(C \mid x)$ 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 $X \mid Y \sim \text{Normal}(\mu_Y, \Sigma_Y)$, then

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

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

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

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

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

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

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

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

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

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

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

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

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 $p_{Y|X}(C \mid x) = \sigma(\langle w|x\rangle + \alpha)$.

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 (algorithm ??).

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 $\operatorname{MLE}(Y \mid X) = \operatorname{argmax}_y p_{X|Y}(X \mid y)$.

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

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

$$\begin{aligned}
\operatorname{MLE}(\mu, \sigma \mid X_1, \ldots, X_n) &= \operatorname*{argmax}_{\mu,\sigma} p_{X_1,\ldots,X_n|\mu,\sigma^2}\left(X_1, \ldots, X_n \,\middle|\, \mu, \sigma^2\right) \\
&= \operatorname*{argmax}_{\mu,\sigma} \prod_{i=1}^{n} p_{X_i|\mu,\sigma^2}\left(X_i \,\middle|\, \mu, \sigma^2\right) \\
&= \operatorname*{argmax}_{\mu,\sigma} \sum_{i=1}^{n} \log\left(p_{X_i|\mu,\sigma^2}\left(X_i \,\middle|\, \mu, \sigma^2\right)\right) \\
&= \sum_{i=1}^{n} \left(-\frac{\|X_i - \mu\|^2}{2\sigma^2} - d\log\left(\sqrt{2\pi}\right) - d\log(\sigma) \right)
\end{aligned}$$

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

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

and from the second equation we have

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

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 $f(z) = \|z\|_2^2 = z^\mathsf{T} z$; this is quadratic and isotropic, where the isosurfaces are hyperspheres. More to the point, the function $f(x) = x^\mathsf{T} A^{-2} x = \left\|A^{-1}x\right\|_2^2$ 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 $\left\|A^{-1}x\right\|_2^2 = 1$. 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, $\left\|A^{-1}v_i\right\|_2^2 = \left\|\frac{1}{\lambda_i}v_i\right\|_2^2 = 1$, 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 $x^\mathsf{T} M x > 0$ for all $x \neq 0$. If A has non-negative eigenvalues then A is **positive semi-definite**, and $x^\mathsf{T} A x \geq 0$ 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 $f(x) = x^\mathsf{T} A x$ 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 than the quadratic form has no minimum value (and can grow arbitrarily negative).

We've been dealing with the quadratic form $f(x) = x^\mathsf{T} M x$ (with M some PD/PSD/ID matrix) instead of $f(x) = x^\mathsf{T} A^{-2} x$, which generates the ellipse as an isocontour. It should be easy to see that if the original quadratic from is $x^\mathsf{T} M x$ 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 $\Sigma = U \Lambda U^\mathsf{T}$, we can find the symmetric square root $\Sigma^{1/2} = U \Lambda^{1/2} U^\mathsf{T}$.

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

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

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 $q(x) = (x-\mu)^\mathsf{T} \Sigma^{-1} (x-\mu)$; 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 $\mathbb{R} \to \mathbb{R}$.

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 $\Sigma = U \Lambda U^\mathsf{T}$, $\Sigma^{-1} = U \Lambda^{-1} U$.

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

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

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

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

and

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

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

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

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**

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

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

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

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

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: $h_w(x) = w^\mathsf{T} x$ (x is usually augmented with a 1 vector to find an affine regressor).

- Polynomial regression, which is linear regression with polynomial features.
- Logistic regression: $h_w = \sigma\left(w^\mathsf{T} x\right)$, where $\sigma(x) = \frac{1}{1+\mathrm{e}^{-x}}$, 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: $J(h_w) = \max_{1 \le i \le n} L(h_w(X_i), y_i)$.
- 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

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

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

$$\begin{aligned} w^* &= \operatorname*{argmin}_w \operatorname{RSS}(w) \\ \nabla_w \operatorname{RSS}(w) &= 2X^\mathsf{T} X w - 2X^\mathsf{T} Y \\ &\overset{\text{set}}{=} 0 \\ X^\mathsf{T} X w^* &= X^\mathsf{T} Y \end{aligned}$$

If $X^\mathsf{T} X$ 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 $w^* = \left(X^\mathsf{T} X\right)^{-1} X^\mathsf{T} Y$. If X is very large, we can solve the implied linear system instead of worrying about the inverse. The term $X^+ = \left(X^\mathsf{T} X\right)^{-1} X^\mathsf{T}$ is the **left pseudoinverse** of X, because $X^+ X = I$ when X is full rank. The matrix $H = XX^+$ is called the **hat matrix**, because $\hat{Y} = XX^+ Y = HY$. If $n > d + 1$ then H is singular.

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

Logistic Regression

Logistic regression has the regression function $h_w(x) = \sigma\left(w^\mathsf{T} x\right)$. 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

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

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

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

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

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 discrimiinant 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 w_i biases

$\left(\hat{Y}_i - Y_i\right)^2$ to be smaller, where $\hat{Y} = Xw$. The optimization problem is

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

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

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:

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

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

Algorithm 5 Newton's method.

Input: A convex twice-differentiable function $f(w) \colon \mathbb{R}^d \to \mathbb{R}$ 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 $f(w) \colon \mathbb{R}^d \to \mathbb{R}$ 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 $s_i = \sigma\left(X_i^\mathsf{T} w\right)$ and $s = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}$. Then $\nabla_w J(w) = -\sum_{i=1}^n (Y_i - s_i) X_i = -X^\mathsf{T}(Y - s)$ and $\nabla_w^2 J(w) = \sum_{i=1}^n s_i (1 - s_i) X_i X_i^\mathsf{T} = X^\mathsf{T}\Omega X$ where $\Omega = \operatorname{diag}(s \otimes (1 - s))$. Clearly Ω is positive definite, so $X^\mathsf{T}\Omega X$ 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 $\text{FP} = \frac{|\{y_i : y_i = -1, f(X_i) = -1\}|}{|\{y_i : y_i = -1\}|}$ on the x-axis, against the true positive rate $\text{TP} = \frac{|\{y_i : y_i = 1, f(X_i) = 1\}|}{|\{y_i : y_i = 1\}|}$. The false negative rate is $\text{FN} = 1 - \text{TP}$ and the true negative rate is $\text{FN} = 1 - \text{FP}$. 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 $g \colon \mathbb{R}^d \to \mathbb{R}$ 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 $h(x) = \text{Ex}_{\mathcal{D}}[Y \mid X = x]$; this estimator is unbiased, since

$$\begin{aligned} h(X) &= \operatorname*{Ex}_{\mathcal{D}}[Y \mid X = x] \\ &= \operatorname*{Ex}_{\mathcal{D}}[g(X_i) + \varepsilon \mid X = x] \\ &= \operatorname*{Ex}_{\mathcal{D}}[g(X_i) \mid X = x] + \operatorname*{Ex}_{\mathcal{D}}[\varepsilon \mid X = x] \\ &= g(X) \end{aligned}$$

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

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

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

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

$$\ell(g \mid (X_i, Y_i)_{i=1}^n) = \log\left(\prod_{i=1}^n p_{Y \mid X}(y_i \mid x_i)\right)$$

$$= \sum_{i=1}^{n} \log\left(p_{Y|X}(y_i \mid x_i)\right)$$
$$= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y_i - g(x_i)\right)^2 - C'$$

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

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

and the log-likelihood is

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

There are two sources of error in a hypothesis h:

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

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

Then

$$\begin{aligned}
R(h) &= \operatorname*{Ex}_{\mathcal{D},\gamma}[L(h(z), \gamma)] \\
&= \operatorname*{Ex}_{\mathcal{D},\gamma}\Big[\left(h(z) - \gamma\right)^2\Big] \\
&= \operatorname*{Ex}_{\mathcal{D}}\Big[h(z)^2\Big] + \operatorname*{Ex}_{\mathcal{D},\gamma}\left[\gamma^2\right] - 2 \operatorname*{Ex}_{\mathcal{D},\gamma}[\gamma h(z)] \\
&= \operatorname*{Var}_{\mathcal{D}}[h(z)] + \operatorname*{Ex}_{\mathcal{D}}[h(z)]^2 + \operatorname*{Var}_{\mathcal{D},\gamma}[\gamma] + \operatorname*{Ex}_{\mathcal{D},\gamma}[\gamma]^2 - 2 \operatorname*{Ex}_{\mathcal{D},\gamma}[\gamma] \operatorname*{Ex}_{\mathcal{D}}[h(z)] \\
&= \left(\operatorname*{Ex}_{\mathcal{D}}[h(z)] - \operatorname*{Ex}_{\mathcal{D},\gamma}[\gamma]\right)^2 + \operatorname*{Var}_{\mathcal{D}}[h(z)] + \operatorname*{Var}_{\mathcal{D},\gamma}[\gamma] \\
&= \left(\operatorname*{Ex}_{\mathcal{D}}[h(z)] - g(z)\right)^2 + \operatorname*{Var}_{\mathcal{D}}[h(z)] + \operatorname{Var}[\varepsilon] \\
&= \underbrace{\operatorname{Bias}(h)^2}_{\text{squared bias}} + \underbrace{\operatorname*{Var}_{\mathcal{D}}[h(z)]}_{\text{variance}} + \underbrace{\operatorname{Var}[\varepsilon]}_{\text{irreducible error}}
\end{aligned}$$

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 $n \to \infty$.
- If h can fit g exactly, for many "nice" distributions the bias goes to 0 as $n \to \infty$.
- 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:

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

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

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

for which the closed form solution always exists and is

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

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

Regularization always provides well-defined solutions to normal equations because the matrix $X^\mathsf{T} X + \lambda I \succ 0$ (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 $\operatorname{Var}[h_w(x)] = \operatorname{Var}\left[x^\mathsf{T} \left(X^\mathsf{T} X + \lambda I \right)^{-1} Y \right]$.

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 $\left\|P^{1/2} w\right\|_2^2$ for P a symmetric matrix indicating differing penalities on each feature.

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

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

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:

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

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 ∃y ∈ 𝒴 such that Y_i = y for all i ∈ S then
        return Leaf(y)
    else
        Choose best splitting feature j and splitting value β by minimizing J(S) among all splits.
        S_l ← {i : X_{i,j} < β}
        S_r ← {i : X_{i,j} ≥ β}
        return InternalNode(j, β, GrowTree(S_l), GrowTree(S_r))
GrowTree({1, ..., 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

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

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 takesd $\mathcal{O}(nd \cdot \text{depth of the tree})$ 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 overfiting, 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 guranteed 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 effcective 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 eveery sample point from scratch. Instead, we keep track of which points int he 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 reaseon 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.

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 $1 - \frac{1}{\mathrm{e}}$ 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 $m \le d$ 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 leads to more randomness, less correlation between trees, but also more bias.

AdaBoost

AdaBoost (adaptive boosting) is one particular modification to the boosting algorithm which works similarly to multiplicative weights. We use different weights for each learner, increasing weights of misclassified sample points, adn giving bigger votes to more accurate learners. We train T classifiers $G_1, \dots, G_T$. The weight for sample point X_i in G_t grows according to how many of $G_1 \dots, G_{t-1}$. Moreover, if X_i is misclassified by very accurate learners, its weight grows even more. And the weight shrinks every time X_i is correctly classified. We train G_t to try harder to correctly classify sample points with larger weights. The metalearner M is a linear combination of learners. For test point z, we have $M(z) = \sum_{t=1}^{T} \beta_t G_t(z)$. Each $G_t \in \{-1, 1\}$, but the final prediction of z returns $\operatorname{sign}(M(z))$.

In iteration T, we choose G_T and β_T that minimize the recursive optimization problem

$$p^* = \inf_{G_t, \beta_t} \underbrace{\frac{1}{n} \sum_{i=1}^{n} L(M(X_i), Y_i)}_{\text{risk}}$$
$$\text{s.t. } M(X_i) = \sum_{t=1}^{T} \beta_t G_t(X_i)$$

We usually use (for the metalearner) the exponential loss function $L(y, y_{\text{true}}) = \mathrm{e}^{-y y_{\text{true}}}$. The individual learners may or may not use their own loss functions. It's important to note that $y_{\text{true}} \in \{-1, 1\}$ but $y = M(X)$ is continuous. The exponential loss function has the advantage that it pushes hard against badly misclassified points.

Assuming that we use this loss, we write

$$\begin{aligned}
n \cdot \text{risk} &= \sum_{i=1}^{n} L(M(X_i), Y_i) \\
&= \sum_{i=1}^{n} \exp(-Y_i M(X_i)) \\
&= \sum_{i=1}^{n} \exp\left(-Y_i \sum_{t=1}^{T} \beta_t G_t(X_i)\right) \\
&= \sum_{i=1}^{n} \prod_{t=1}^{T} \exp(-\beta_t Y_i G_t(X_i)) \\
&= \sum_{i=1}^{n} w_i^{(T)} \exp(-\beta_T Y_i G_T(X_i)) && (w_i^{(T)} = \textstyle\prod_{t=1}^{T-1} \exp(-\beta_t Y_t G_t(X_i)).) \\
&= \mathrm{e}^{-\beta_T} \left(\sum_{Y_i = G_T(X_i)} w_i^{(T)} \right) + \mathrm{e}^{\beta_T} \left(\sum_{Y_i \neq G_T(X_i)} w_i^{(T)} \right) \\
&= \mathrm{e}^{-\beta_T} \left(\sum_{i=1}^{n} w_i^{(T)} \right) + \left(\mathrm{e}^{\beta_T} - \mathrm{e}^{-\beta_T} \right) \left(\sum_{Y_i \neq G_T(X_i)} w_i^{(T)} \right).
\end{aligned}$$

The G_T that minimizes the risk is the learner that minimizes the sum of $w_i^{(T)}$ over all misclassified points X_i.

We have a direct recursive formulation by definition of the weights, namely $w_i^{(T)} = w_i^{(T)} \mathrm{e}^{-\beta_T Y_i G_T(X_i)}$. Notice that a weight shrinks if the point was classified correctly, and grows if the point was misclassified.

It remains to choose β_T; we do this by setting $\nabla_{\beta_T}\text{risk} \overset{\text{set}}{=} 0$. We have

$$\begin{aligned}
\nabla_{\beta_T}(n\cdot\text{risk}) &= \nabla_{\beta_T}\left[\mathrm{e}^{-\beta_T}\left(\sum_{i=1}^n w_i^{(T)}\right)+\left(\mathrm{e}^{\beta_T}-\mathrm{e}^{-\beta_T}\right)\left(\sum_{Y_i\neq G_T(X_i)} w_i^{(T)}\right)\right]\\
&= -\mathrm{e}^{-\beta_T}\left(\sum_{i=1}^n w_i^{(T)}\right)+\left(\mathrm{e}^{\beta_T}+\mathrm{e}^{-\beta_T}\right)\left(\sum_{Y_i\neq G_T(X_i)} w_i^{(T)}\right)\\
&\overset{\text{set}}{=} 0\\
0 &= -1+\left(\mathrm{e}^{2\beta_T^*}+1\right)\frac{\sum_{Y_i=G_T(X_i)} w_i^{(T)}}{\sum_{i=1}^n w_i^{(T)}}\\
\beta_T^* &= \frac{1}{2}\log\left(\frac{\sum_{Y_i=G_T(X_i)} w_i^{(t)}}{\sum_{Y_i\neq G_T(X_i)} w_i^{(T)}}\right)\\
&= \frac{1}{2}\log\left(\frac{1-\text{err}_T}{\text{err}_T}\right)
\end{aligned}$$

Note that the weighted error err_T of G_T is defined by $\frac{\sum_{Y_i\neq G_T(X_i)} w_i^{(T)}}{\sum_{i=1}^n w_i^{(T)}}$. If $\text{err}_T = 0$ then $\beta_T = \infty$; if $\text{err}_T = 1$ then $\beta_T = -\infty$; if $\text{err}_T = \frac{1}{2}$, then $\beta_T = 0$.

This results in the actual AdaBoost algorithm:

Algorithm 8 AdaBoost algorithm.

Input: A sequence of learners (G_t), a dataset X, a termination iteration T.
Output: A metalearner $h(\cdot)$.

```
for i ∈ [1, n] do
    w_i ← 1/n
for t ∈ [1, T] do
    Train G_t with weights w_i.
    err ← (Σ_{Y_i ≠ G_t(X_i)} w_i) / (Σ_{i=1}^n w_i).
    β_t ← (1/2) log((1 − err)/err).
    for i ∈ [1, n] do
        w_i ← w_i e^{−β_t Y_i G_t(X_i)}
return h(z) = sign(Σ_{t=1}^T β_t G_t(z)).
```

The most common G_t are short decision trees. This is because

- Decision trees are fast, and short trees are really fast.
- Very little hyperparameter search.
- It's easy to beat any low $\frac{1}{2}+\varepsilon$ threshold by tuning decision trees.
- Easy bias-variance control. Boosting can overfit, but short trees generally have less overfitting.
- AdaBoost with short trees is a form of subset selection.
- Linear decision boundaries don't boost very well.

One can estimate the posterior probably $p_{Y|X}(1\,|\,x)\approx\sigma(2M(x))$ where σ is the sigmoid function. The most important part of AdaBoost is that if every G_t has error $\frac{1}{2}-\varepsilon_t$ for any $\varepsilon_t>0$ then as $\lim_{T\to\infty}|\{i\,|\,M(X_i)\neq Y_i\}|=0$.

12 Kernels

With d input features, a degree-p polynomial lifting map ϕ has codomain $\mathbb{R}^{\mathcal{O}(d^p)}$. Today, we wish to use these features without computing them.

In many algorithms, the following two ideas hold:

- the weights can be written as a linear combination of sample points X_i.
- the only thing we care about is the inner products of $\phi(X_i)$.

Therefore, we don't need to compute $\phi(x_i)$ at all.

Assume $w = X^\mathsf{T}\alpha = \sum_{i=1}^n \alpha_i X_i$ for some $\alpha \in \mathbb{R}^n$. We substitute this identity into our algorithm, and optimize the **dual parameter** α instead of the $\mathcal{O}(d^p)$ primal weights w.

Example 35. Suppose we have the ridge regression problem with normal solution

$$\left(X^\mathsf{T}X + \lambda I\right) w^* = X^\mathsf{T}y$$

Suppose α is a solution to

$$\left(XX^\mathsf{T} + \lambda I\right) \alpha = y$$

Then

$$X^\mathsf{T}y = X^\mathsf{T}XX^\mathsf{T}y + \lambda X^\mathsf{T}\alpha = \left(X^\mathsf{T}X + \lambda I\right) X^\mathsf{T}\alpha$$

Therefore $w = X^\mathsf{T}\alpha$ is a solution to the normal equations, and w is a linear combination of sample points. α is a dual solution to the dual form of ridge regression, which is

$$\alpha = \inf_a \left(\left\|XX^\mathsf{T}a - y\right\|_2^2 + \lambda\left\|X^\mathsf{T}a\right\|_2^2\right)$$

The training phase solves the equation

$$\left(XX^\mathsf{T} + \lambda I\right)\alpha = y$$

for α, and the regression function is

$$h(z) = w^\mathsf{T}z = \alpha^\mathsf{T}Xz = \sum_{i=1}^n \alpha_i\left(X_i^\mathsf{T}z\right) = \sum_{i=1}^n \alpha_i \left\langle X_i | z\right\rangle.$$

Let the kernel function be $k(x, z) = \langle x|z\rangle$, with the corresponding kernel matrix $K = XX^\mathsf{T}$ (where $K_{i,j} = k(x_i, x_j)$). K is singular if $n > \mathcal{O}(d^p)$; in this fcase, there's most likely no solution if $\lambda = 0$.

The primal training method takes $\mathcal{O}\left(d^3 + d^2n\right)$; the dual training method takes $\mathcal{O}\left(n^3 + n^2d\right)$ time. We prefer dual when $d > n$.

The polynomial kernel of degree p is $k(x, z) = (\langle x|z\rangle + 1)^p$.

Lemma 36. We may write $(\langle x|z\rangle + 1)^p$ as $\phi(x)^\mathsf{T}\phi(z)$ where $\phi(x)$ contains all polynomials in x_i up to degree p and constant factors.

Proof. Write the binomial expansion and take each term as a feature. □

This allows us to compute $\langle\phi(x)|\phi(z)\rangle$ in $\mathcal{O}(d + \log(p))$ time instead of $\mathcal{O}(d^p)$ time.

Now, let's redefine the kernel to be $k(x, z) = \langle\phi(x)|\phi(z)\rangle$, but crucially let's not compute $\phi(x)$ or $\phi(z)$, instead computing $k(x, z) = (\langle x|z\rangle + 1)^p$. This allows us to compute the polynomial feature matrix in linear time in d instead of in d^p.

This idea is of use in the perceptron. Let's say that $\phi(X)$ be an $n \times D$ matrix with rows $\phi(X_i)^\mathsf{T}$, where $D = \dim(\text{codomain}(\phi))$. Then let $K = \phi(X)\phi(X)^\mathsf{T}$ and dualize $w = \phi(X)^\mathsf{T}\alpha$. Then the gradient descent step $\alpha_i \leftarrow \alpha_i + \varepsilon y_i$ has the same effect as $w \leftarrow w + \varepsilon y_i\phi(X_i)$, since $\langle\phi(X_i)|w\rangle = (\phi(X)w)_i = \left(\phi(X)\phi(X)^\mathsf{T}\alpha\right)_i = (K\alpha)_i$. The dual perceptron algorithm thus optimizes α:

Algorithm 9 Kernelized Dual Perceptron Algorithm

Input: Training points $(X_i, Y_i)_{i=1}^n$, kernel function $k(x, z)$, learning rate.
Output: Prediction function $h(z)$.

$\alpha \leftarrow \begin{bmatrix} Y_1 & 0 & \dots & 0 \end{bmatrix}^\mathsf{T}$
$K \leftarrow [k(X_i, X_j)]_{i,j=1}^n$
while $\exists i$ s.t. $Y_i (K\alpha)_i < 0$ **do**
    $\alpha_i \leftarrow \alpha_i + \varepsilon Y_i$
return $h(z) \leftarrow \sum_{j=1}^n \alpha_j k(X_j, z)$ or $h(z) \leftarrow \langle w | z \rangle$ for $w = \phi(X)^\mathsf{T} \alpha$ (only possible when D is finite)

We can modify the logistic regression gradient descent likewise. For example, the stochastic gradient descent step is

$$\alpha_i \leftarrow \alpha_i + \varepsilon \left(Y_i - \sigma((K\alpha)_i) \right)$$

and the batch gradient descent step is

$$\alpha \leftarrow \alpha + \varepsilon \left(Y - \sigma(K\alpha) \right)$$

with regression function

$$h(z) \leftarrow \sigma \left(\sum_{j=1}^n \alpha_j k(X_j, z) \right)$$

Support vector machines can also be kernelized; this was rather popular until neural networks took over.

We can start defining $k(x, z)$ as some similarity measure of x and z that is some inner product of some functions of x and z, with the assumption that we can utilize Taylor series to get polynomials.

The Gaussian kernel is $k(x, y) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{\|x-z\|_2^2}{2} \right)$. Then in the case $d = 1$, $\phi(x) = \exp\left(-\frac{x^2}{2\sigma^2} \right) \left[\frac{x^i}{\sigma^i \sqrt{i!}} \right]_{i=0}^\infty$, and the regression function is $h(z) = \sum_{j=1}^n \alpha_j k(X_j, z)$ is a linear combination of Gaussians centered at sample points X_i.

Infinite-dimensional kernels are popoular because they're smooth and resistent to overfitting, analogous to a weighted continuous version of k-nearest-neighbors.

13 Neural Networks

Neural networks are algorithms that can do classification and regression. The idea is to put the output of some linear classifiers as the input of another linear classifier, and so on. But linear combinations of linear combinations are still linear combinations, so the output is linear in the inputs, and so is a hyperplane in the input. To combat that, we introduce nonlinearity at each stage.

More formally, we will cover a neural network with p hidden layers. The equation system is $h_i = A_{i-1}(W_{i-1} h_{i-1})$ with $h_0 = X$ (input layer) and $h_{p+1} = Z$ (output layer), and activation function (usually nonlinear and vector-valued) $A_i = \left[A_{i,j} \right]_{j=1}^{\dim(\text{image}(A_i))}$. The W_i are learned matrices. Sometimes we impose constraints on W_i; sometimes these matrices are even sparse. For notational convenience write $Z = h_W(X)$, the hypothesis function.

The training is usually stochastic or batch gradient descent. We pick a loss function $L(z, y)$ (i.e., $L(z, y) = \|z - y\|_2^2$) and the cost function is $J(W) = \sum_{i=1}^n L(z_W(X_i), Y_i)$. Usually this cost function has many local minima. The gradient descent update rule is $W \leftarrow W - \varepsilon \nabla_W J(W)$, where $\nabla_W J(W)$ is the tensor derivative, i.e. $[\nabla_W J(W)]_{i,j,k} = \nabla_{W_{i,j,k}} J(W)$.

Each value h_i gives a partial derivative of the form $\frac{\partial z}{\partial h_i} = \frac{\partial z}{\partial h_{i+1}} \frac{\partial h_{i+1}}{\partial h_i}$. The second term is computed in the forward pass; the first term is computed in the backward pass, which happens after the forward pass ("backpropagation").

The backpropagation algorithm, in fact, is to compute all $\frac{\partial h_{i+1}}{\partial h_i}$, and then plug these values into the expanded form of $\frac{\partial z}{\partial h_{i-1}}$. The time is $\mathcal{O}(E)$, the number of edges in the computational graph.

For regression, we use a cost function $L(z, y) = \|z - y\|_2^2$, and $A_p(x) = \sigma(x)$ componentwise, the logistic function; the output layer is one-dimensional, i.e. $Z = h_{p+1} \in \mathbb{R}$. For two-class classification, we use a cost function $L(z, y) = y \log(z) + (1 - y) \log(1 - z)$, the logistic loss, and $A_p(x) = \sigma(x)$ componentwise; the output layer is also one-dimensional, but this time $Z = h_{p+1} \in \{0, 1\}$. For k-class classification where $k \geq 3$, the output layer is a k-dimensional vector which is a one-hot encoding of the classes. The loss function is then a generalized cross-entropy loss $L(z, y) = -\sum_{i=1}^{k} y_i \log(z_i)$, and $A_p(x)$ is defined componentwise by $A_{p,i}(x) = \frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{k} \mathrm{e}^{x_j}}$ (the softmax function).

The issue with unit saturation is when $A_i(x) \approx 0$, then for most A_i (especially when $A_i(x) = \sigma(x)$), then $\frac{\mathrm{d}A_i(x)}{\mathrm{d}x} \approx 0$ as well. This leads to slow training and a bad local minimum. Some mitigating factors for this problem are:

- Initialize the weights randomly. If $\eta_{i,j}$ is the number of edges into $W_{i,j}$ (i.e., the j^{th} unit in the i^{th} layer), then $W_{i,j,k} \sim \text{Normal}\left(0, \frac{1}{\eta_{i,j}}\right)$ for any k.
- For two-class classification, set the target values to $\frac{19}{20}$ and $\frac{1}{20}$ or similar.
- Add noise to the gradients during backpropagation.
- Use regularization on the weight tensor.
- Don't use $A_i(x) = \sigma(x)$; others can work fine, such as $A_i(x) = \max(x, 0)$ (ReLU) and so on.

For faster training, any of these apply. Also,

- Stochastic gradient descent is faster than batch gradient descent on large, redundant data sets.
- Normalize/center/scale the data.
- Center the hidden units: use $A_i(x) = \tanh(x)$. Note that $\frac{\mathrm{d}}{\mathrm{d}x} \tanh(x) = 2\sigma(2x) - 1$.
- Different layers have different learning rates – earlier layers have smaller gradients and need larger learning rates.
- Using schemes to balance out the training data or give higher weight in the training process to misclassified examples
- Second-order optimization

Heuristics for avoiding bad local minima are:

- Avoid unit saturation
- Stochastic gradient descent helps a bit
- Momentum – keep a velocity variable used in your gradient descent iteration, and by accounting for velocity we can skip right over local minima.
  - Initially $\Delta_w \leftarrow -\varepsilon \nabla_w J(w)$, and while convergence hasn't been reached then do $w \leftarrow w + \Delta_w$ then $\Delta_w \leftarrow -\varepsilon \nabla_w J(w) + \beta \Delta_w$.
  - Choose $\beta < 1$.

Heuristics for avoiding overfitting are:

- Use ensembles of neural nets; have random initial weights and bagging.
- Use ℓ^2 regularization (weight decay); add $\lambda \|w\|_2^2$ to the cost/loss function, where w is the vector of all weights. The term $-\varepsilon \frac{\partial J}{\partial w_i}$ has an extra term $-2\varepsilon\lambda w_i$.
- Dropout emulates an ensemble in one network. During any given epoch, take a random subset of neurons and disable them, i.e. temporarily set them to zero.
- Use fewer hidden units.

Convolutional Neural Networks

Neural networks are often overparameterized; there are either too many weights, or too little data to effectively train these weights. Convolutional neural networks combat these by:

- Local connectivity: A hidden unit (in the early layer) connects only to a small patch of units in the previous layer
- Shared weight: Groups of hidden units share the same set of weights, called a filter or mask. The hidden units in the first layer are just the masks and patches. If the net learns to detect edges on one patch, every patch has an edge detector (for example).

14 Unsupervised Learning

In unsupervised learning, there are sample points, but no labels. Our goal is to discover structure in the data.

Examples of unsupervised learning are:

- Clustering: partition data into groups of similar/nearby points
- Dimensionality reduction: data often lies near a low-dimensional subspace (or manifold) in feature space; matrices have low-rank approximations. Whereas clustering is about grouping similar sample points, dimensionality reduction is more about identifying a continuous variation from sample point to sample point.
- Density estimation: fit a continuous distribution to discrete data.

Dimensionality Reduction

Principal components analysis is an unsupervised learning technique. Given sample points $(X_i)_{i=1}^n$ in $\mathbb{R}^d$, we want to find the k directions that capture most of the variation.

We wish to do this to:

- Find a small basis for representing variations in complex things.
- Reducing # of dimensions makes some computations cheaper, e.g. regression.
- Remove irrelevant dimensions to reduce overfitting in learning algorithms.
- Like subset selection, but the "features" aren't axis-aligned; they're linear combos of input features.

Sometimes PCA is used as a preprocess before regression or classification for the last two reasons.

Assume that $X \in \mathbb{R}^{n \times d}$ with $\sum_{i=1}^n X_{i,j} = 0$ for each j (i.e., empirical expectation of $X_{:,i}$ is zero). As usual, we can center the data by computing $\mu_X = \sum_{i=1}^n X_i$ and then doing $X \leftarrow X - 1\mu_X^\mathsf{T}$.

Given some orthonormal axis vectors $(v_i)_{i=1}^k$ and $V_k = \operatorname{span}\left((v_i)_{i=1}^k\right)$, recall that the "best approximation" to x in V_k is $\widetilde{x} = \operatorname{proj}_{V_k}(x) = \sum_{i=1}^k \langle x | v_i \rangle v_i$. In the basis $(v_i)_{i=1}^k$, the i^{th} coordinate of $\widetilde{x}$ is $\langle v_i | x \rangle$ – these coordinates are what's important about x in the new space.

If $X \in \mathbb{R}^{n \times d}$, then $X^\mathsf{T}X \in \mathbb{R}^{d \times d}$ is square, symmetric, and positive semidefinite. It turns out that the k principal components of X are the k eigenvectors of $X^\mathsf{T}X$ with the largest eigenvalues.

Here are three interpretations of this result.

- If we fit a Gaussian to the data with maximum likelihood estimation, then the k Gaussian axes of greatest variance, are the k axes of highest variance in the data. Since MLE gives that the covariance matrix of the fitted Gaussian is $\hat{\Sigma} = \frac{1}{n} X^\mathsf{T}X$ (using the centered version of X), this shows the result.

- If we want to find the directions $(v_i)_{i=1}^k$ that maximize the sample variance of projected data, then we get the recursive optimization problems

$$\begin{aligned} v_i &= \operatorname*{argsup}_{v \perp v_1, \dots, v_{i-1}} \operatorname{Var}\left[\operatorname*{proj}_{\operatorname{span}(v_1, \dots, v_{i-1}, v)}(X)\right] \\ &= \operatorname*{argsup}_{v \perp v_1, \dots, v_{i-1}} \frac{1}{n} \sum_{j=1}^n \left(\left\langle X_j \middle| \frac{v}{\|v\|_2} \right\rangle\right)^2 \\ &= \operatorname*{argsup}_{v \perp v_1, \dots, v_{i-1}} \frac{1}{n} \frac{\|Xv\|_2^2}{\|v\|_2^2} \\ &= \operatorname*{argsup}_{v \perp v_1, \dots, v_{i-1}} \frac{1}{n} \frac{v^\mathsf{T} X^\mathsf{T} X v}{v^\mathsf{T} v} \end{aligned}$$

  This is the Rayleigh quotient and indeed v_i is the i^{th} eigenvalue of $X^\mathsf{T} X$, or the eigenvalue of $X^\mathsf{T} X$ with the i^{th} largest eigenvalue. This gives the result.

- If we want to find the directions $(v_i)_{i=1}^k$ that minimize the projection error $\left\|X - \operatorname{proj}_{\operatorname{span}\left((v_i)_{i=1}^k\right)}(X)\right\|_F$, then we have the recursive optimization problems

$$\begin{aligned} v_i &= \operatorname*{arginf}_{v \perp v_1, \dots, v_{i-1}} \left\|X - \operatorname*{proj}_{\operatorname{span}(v_1, \dots, v_{i-1}, v)}(X)\right\|_F \\ &\operatorname*{arginf}_{v \perp v_1, \dots, v_{i-1}} \sum_{j=1}^n \left\|X_j - \operatorname*{proj}_{\operatorname{span}(v_1, \dots, v_{i-1}, v)}(X_j)\right\|_2^2 \\ &\operatorname*{arginf}_{v \perp v_1, \dots, v_{i-1}} \sum_{j=1}^n \left\|X_j - \sum_{l=1}^{i-1} \langle X_j | v_i \rangle v_i\right\|_2^2 \end{aligned}$$

  at which point expanding and doing the algebra leads to the second point of reasoning and therefore proves the claim.

The **singular value decomposition** is also relevant to principal components analysis. The central problem is that $X^\mathsf{T} X$ takes $\mathcal{O}(nd^2)$ time to compute and is usually poorly conditioned, so the eigenvectors are numerically inaccurate.

One can think of the singular value decomposition as a generalization of the symmetric matrix eigendecomposition. If $X \in \mathbb{R}^{n \times d}$, write $X = U_X D_X V_X^\mathsf{T}$, where $r \le \min(m, n)$ and $U_X \in \mathbb{R}^{n \times r}$, $D_X \in \mathbb{R}^{r \times r}$, and $V_X \in \mathbb{R}^{d \times r}$. The orthonormal rows v_i^T are the right singular vectors of X; $V_X^\mathsf{T} V_X = I$. Similarly the orthonormal columns u_i are the left singular vectors of X. The diagonal entries σ_i of D are nonnegative (in this formulation, positive) singular values of X.

For each i, v_i is an eigenvector of $X^\mathsf{T} X$ with eigenvalue σ_i^2. This can be observed by computing $X^\mathsf{T} X = V_X D_X U_X^\mathsf{T} U_X D_X V_X^\mathsf{T} = V_X D_X^2 V_X^\mathsf{T}$. In particular, $X v_j = \sigma_j u_j$.

We can find the k greatest singular values and corresponding vectors in $\mathcal{O}(ndk)$ time deterministically; there are randomized algorithms that d o better.

Clustering

We want to partition data into clusters os points in a cluster are more similar than across clusters. A method of clustering is ***k*-means clustering** or **Lloyd's algorithm**. It's a heuristic to assign k clusters $(Y_i)_{i=1}^n \in [k]$ to the data points $(X_i)_{i=1}^n$ such that it follows the optimization problem

$$y^* = \operatorname*{arginf}_y \sum_{i=1}^k \sum_{y_j = i} \left\|X_j - \frac{1}{|\{l \mid Y_l = i\}|} \sum_{Y_l = i} X_l\right\|_2^2$$

Algorithm 10 k-Means Clustering

Input: A set of data points $(X_i)_{i=1}^n$ and a positive integer k.
Output: A set of cluster assignments $(Y_i)_{i=1}^n \in [k]$.

```
Give random cluster assignments (Y_i)_{i=1}^n
for i ∈ [k] do
    μ_i ← 1/|{j | Y_j=1}| Σ_{Y_j=1} X_j
t ← 0
f_t ← Σ_{i=1}^k Σ_{y_j=i} ‖X_j − μ_i‖_2^2
while f_t > f_{t−1} do
    for i ∈ [k] do
        μ_i ← 1/|{j | Y_j=1}| Σ_{Y_j=1} X_j
    for i ∈ [n] do
        Y_i ← argmin_j ‖X_i − μ_j‖_2^2
    t ← t + 1
    f_t ← Σ_{i=1}^k Σ_{y_j=i} ‖X_j − μ_i‖_2^2
return (Y_i)_{i=1}^n
```

Both steps decrease the objective, unless the objective stays the same, in which case we terminate the algorithm. Since there are a finite number of assignments $(Y_i)_{i=1}^n$, the algorithm terminates at a local minimum.

The random initialization can be improved; for example, we can choose k random sample points to be the initial μ_i.

An equivalent objective function minimizes the within-cluster variation:

$$y^* = \operatorname*{arginf}_y \sum_{i=1}^k \frac{1}{|\{l \mid Y_l = i\}|} \sum_{y_j=i} \sum_{y_l=i} \|X_j - X_l\|_2^2$$

Sometimes, when the units are different, we want to normalize; sometimes we don't. Features that tend to be larger also tend to be optimized more heavily than others.

We cna generalize k-means beyond Euclidean distance. We specify a dissimilarity measure $d(x, y)$; this does not necessarily satisfy the triangle inequality. We generalize the mean to the medoid $\nu_j = X_{\operatorname{argmin}_{Y_i=j} \sum_{Y_l=j} d(X_i, X_l)}$; the sample point which minimizes total distance to all other points in the same cluster. The rest of the objective function is the same.

Hierarchical clustering creates a tree, where every subtree is a cluster. The bottom-up approach is agglomerative clustering; we start with each point being a cluster and repeatedly union pairs of clusters. The top-down approach is divisive clustering; we start with all points in one cluster and repeatedly split the clusters.

To do this, we figure out which clusters to merge. This requires a distance function on clusters to minimize:

- complete linkage: $d(A, B) = \max\{d(x, y) \mid x \in A, y \in B\}$
- single linkage: $d(A, B) = \min\{d(x, y) \mid x \in A, y \in B\}$
- average linkage: $d(A, B) = \frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} d(x, y)$
- centroid linkage: $d(A, B) = d\Big(\frac{1}{|A|} \sum_{X_j \in A} X_j, \frac{1}{|B|} \sum_{X_j \in B} X_j\Big)$

This gives greedy algorithms for hierarchical clustering: keep unioning the two clusters A, B that minimize $d(A, B)$; this naively takes $\mathcal{O}(n^3)$ time.

Centroid linkage can cause inversions, where a parent cluster is unioned at a lower metric than its children.

Spectral Graph Clustering

The input to a spectral graph clustering problem is a weighted undirected simple graph $G(V, E)$. Weight $w_{i,j}$ is the weight of edge $(i, j) = (j, i)$; in particular $w_{i,j} = 0$ if $(i, j) \notin E$.

The goal is to cut G into multiple pieces V_i, which minimize some objective in the cuts. For example, we commonly want to minimize the sparsity

$$\sigma = \frac{\operatorname{Cut}((V_i)_i)}{\prod_i \operatorname{Mass}(V_i)}$$

where $\operatorname{Cut}((V_i)_i) = \sum_{(i,j):\, i \in V_k, j \in V_l \neq V k} w_{i,j}$ is the total weight of edges in the cut and $\operatorname{Mass}(V_i) = \sum_{v \in V_i} \operatorname{Mass}(v)$ is the sum of masses of vertices in G_i (most of the time, the mass of each vertex is 1).

The sparsest cut, the minimum weight bisection, and the maximum weight cut are all NP-hard.

Consider the case of a single cut, so we have $V(G) = V_1 \cup V_2$. Let y be the indicator vector defined by $y_i = 2\mathbb{1}(i \in V_1) - 1$. Then

$$w_{i,j} \frac{(y_i - y_j)^2}{4} = \begin{cases} w_{i,j}, & i \in V_1, j \in V_2 \vee i \in V_2, j \in V_1 \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$\begin{aligned} \operatorname{Cut}(V_1, V_2) &= \sum_{(i,j) \in E} w_{i,j} \frac{(y_i - y_j)^2}{4} \\ &= \frac{1}{4} \sum_{(i,j) \in E} \left(w_{i,j} y_i^2 - 2 w_{i,j} y_i y_j + w_{i,j} y_j^2 \right) \\ &= \frac{1}{4} \left[\left(\sum_{(i,j) \in E} -2 w_{i,j} y_i y_j \right) + \left(\sum_{i=1}^{n} y_i^2 \sum_{k \neq i} w_{i,k} \right) \right] \\ &= \frac{1}{4} y^\mathsf{T} L y \end{aligned}$$

where L is a matrix defined by $L_{i,j} = \begin{cases} -w_{i,j}, & i \neq j \\ \sum_{k \neq i} w_{i,k}, & i = j \end{cases}$. Then L is the symmetric $|V| \times |V|$ **Laplacian matrix** for G. Note that $L = D - A$ where D is the diagonal degree matrix for G and A is the off-diagonal adjacency matrix for G. From either definition it is easy to see that L is positive semidefinite. Also, $L = D - A$ so $\sum_i L_{i,j} = 0$ for all j, so $1 \in \operatorname{kernel}(L)$, so 1 is an eigenvector of L with eigenvalue 0. In general the Laplacian matrix L has as many zero eigenvalues as there are connected components of G.

With this framework, the graph bisection problem (want to find V_1, V_2 with exactly $\frac{n}{2}$ vertices each, or $1^\mathsf{T} y = 0$) can be written as

$$\begin{aligned} y^* = \operatorname*{arginf}_y \; & y^\mathsf{T} L y \\ \text{s.t. } & y_i \in \{-1, 1\}, \quad \forall i \in [|V|] && \text{(Binary constraint.)} \\ & 1^\mathsf{T} y = 0 && \text{(Balance constraint.)} \end{aligned}$$

This is also NP-hard. We relax the binary constraints $y_i \in \{-1, 1\}$. These constraints imply that y is at a vertex of an origin-centered hypercube in $\mathbb{R}^{|V|}$; we relax so that y is on the surface of an origin-centered ball in $\mathbb{R}^{|V|}$. Formally, the relaxed problem is

$$\begin{aligned} y^* = \operatorname*{arginf}_y \; & y^\mathsf{T} L y \\ \text{s.t. } & y^\mathsf{T} y = n \\ & 1^\mathsf{T} y = 0 \end{aligned}$$

or simplifying obtains

$$\begin{aligned} y^* = \operatorname*{arginf}_y \ & \frac{y^\mathsf{T} L y}{y^\mathsf{T} y} \\ \text{s.t. } & 1^\mathsf{T} y = 0 \end{aligned}$$

The solution finds the minimum eigenvalue of L whose eigenvalue is orthogonal to 1. This is the second-smallest eigenvalue $\lambda_{|V|-1}$ with associated eigenvector $\lambda_{|V|-1}$ called the Fiedler vector.

Once we find $v_{|V|-1}$, we sort the components of $v_{|V|-1}$ and try the $n-1$ cuts given by $G = \left\{v_{|V|-1,1}, \cdots, v_{|V|-1,i}\right\} \cup \left\{v_{|V|-1,i+1}, \ldots, v_{|V|-1,|V|}\right\}$ and choose the minimum sparsity cut. This is called the sweep cut method.

This provides a minimum-sparsity cut approximation algorithm.

We now want to incorporate vertex masses. Let M be diagonal matrix defined by $M_{i,i} = \text{Mass}(v_i)$. Then the optimization problem becomes

$$\begin{aligned} y^* = \operatorname*{arginf}_y \ & y^\mathsf{T} L y \\ \text{s.t. } & y^\mathsf{T} M y = \operatorname{trace}(M) \\ & 1^\mathsf{T} M y = 0 \end{aligned}$$

This gives an ellipsoid constraint. The solution is the Fiedler vector (v corresponding to the second-smallest λ) of the system $Lv = \lambda Mv$. By contrast, the system $Lv = \lambda v$ is the regular eigensystem, without weights.

Theorem 37 (Cheeger's Inequality). The sweep cut method finds a cut with sparsity

$$\sigma_{\text{sweep}} \leq \sqrt{2\lambda_{|V|-1} \sup_i \left(\frac{L_{i,i}}{M_{i,i}}\right)}.$$

The optimal cut has sparsity

$$\sigma_{\mathsf{OPT}} \geq \frac{\lambda_{|V|-1}}{2}.$$

To partition G into more than two subgraphs, we use greedy divisive clustering, with the splitting method being minimum-sparsity partitioning.

A normalized cut is where we set $M_{i,i} = L_{i,i}$.

To obtain k clusters, we compute the k solutions to the system $Lv = \lambda Mv$ with smallest λ, calling these not-quite-eigenvectors $v_{|V|}, \cdots v_{|V|-k+1}$. Then we scale the v_i such that $v_i^\mathsf{T} M v_i = 1$. Then if $V = \begin{bmatrix} v_{|V|} & \cdots & v_{|V|-k+1} \end{bmatrix}$ then $V^\mathsf{T} M V = 1$. Then the i^{th} row of V, V_i, is the spectral vector for vertex i. We normalize each row V_i to unit length. A theorem is that all vertices in the same connected component have spectral vector pointing in the same direction; each component vector is orthogonal to all other component vectors. Then intuitively, adding low-importance edges only perturbs the spectral vectors of each vertex a little. Then we can do k-means clustering on the perturbed spectral vectors. The summary of this method is that we can use k-means clustering on the spectral vectors of each vertex.

Random Projection

A cheap alternative to principal component analysis as a preprocessing step for clustering, classification, and regression. It's a dimensionality reduction method that approximately preserves distances between points.

We pick a small ε and δ. We pick a random subspace $S \subset \mathbb{R}^d$ of dimension k, where $k = \frac{2\log\left(\frac{1}{\delta}\right)}{\frac{\varepsilon^2}{2} - \frac{\varepsilon^3}{3}}$.

For any point q, let $\hat{q}$ be an orthogonal projection of q onto S, multiplied by $\sqrt{\frac{d}{k}}$.

Lemma 38 (Johnson-Lindenstrauss Lemma). For any two points $x_1, x_2 \in \mathbb{R}^d$, with probability at least $1 - 2\delta$,

$$(1-\varepsilon)\left\|x_1 - x_2\right\|_2^2 \le \left\|\hat{q} - \hat{w}\right\|_2^2 \le (1+\varepsilon)\left\|q - w\right\|_2^2.$$

We pick ε according to the problem and $\delta = \mathcal{O}(n^{-3})$.

High-Dimensional Space

High-dimensional space is weird, which can impact the performance of k-means clustering. For example, take some origin-centered ball of radius r in $\mathbb{R}^d$, and another origin-centered ball of radius $r - \varepsilon$. Then the ratio of inner ball volume to outer ball volume is

$$\frac{(r-\varepsilon)^d}{r^d} = \left(1 - \frac{\varepsilon}{r}\right)^d \ll 1$$

if $d \gg 1$. As such, random points from the uniform distribution over this ball are nearly all in the outer shell. Random points from the Gaussian distribution over the ball are also nearly all in some shell. In particular, if $\rho \sim \text{Normal}(0, I_d)$, then

$$\begin{aligned}
\left\|\rho\right\|_2^2 &= \sum_{i=1}^d \rho_i^2 \\
\text{Ex}\left[\left\|\rho\right\|_2^2\right] &= d\,\text{Ex}\left[\rho_i^2\right] \\
\text{Var}\left[\left\|\rho\right\|_2^2\right] &= d\,\text{Var}\left[\rho_i^2\right]
\end{aligned}$$

As such, if we consider points at distance r and $r - \varepsilon$ from 0 drawn from $\text{Normal}(0, I_d)$, we expect that the fraction of points is $\varepsilon \sqrt[4]{d\,\text{Var}[\rho_i^2]}$.

In high dimensions, Euclidean distances become less meaningful. Then nearest-neighbor heuristics lose their effectiveness.

15 Nearest Neighbors

In the k-nearest neighbors algorithm, given a query point X, we find the k sample points $\left(X_{(i)}\right)_{i=1}^k$ nearest X, with a distance metric of our choice. In a regression task, the relevant property is the mean label of the k points, in a classification task, the relevant property is the mode label of the k points (or the histogram of the class probabilities).

There are two relevant theorems:

Theorem 39 (Cover and Hart, 1967). If we work in a separable metric space, with the training and test distributions being the same, as $n \to \infty$, the error rate of the k-nearest neighbor algorithm with $k = 1$ is at most $2B - B^2$ where B is the Bayes risk. If there are only two classes, the error rate of the k-nearest neighbor algorithm with $k = 1$ is at most $2B - 2B^2$.

Theorem 40 (Fix and Hodges, 1951). If we work in a separable metric space, with the training and test distributions being the same, as $k, n \to \infty$ with $k/n \to 0$, then the error rate of the k-nearest neighbor algorithm converges to B, so in the long run it is the Bayes optimal classifier.

The exhaustive k-nearest neighbors algorithm naively scans through all n sample points X_i, computing distances to X; one quick optimization is that we maintain a max-heap with the k shortest distances seen so far. Then each query takes $\mathcal{O}(nd + n\log(k))$, but if we process the points in random order it takes expected $\mathcal{O}(nd + k\log(n)\log(k))$ which is useful if $n \gg k$. We can use Voronoi diagrams when $d \le 5$, k-d trees when $d \le 30$, and principal components analysis or random projections with $d \gg 30$ (creating an approximate nearest-neighbor solution). We discuss Voronoi diagrams and k-d trees now.

If V is a set of points, then the Voronoi cell of $X \in V$ is

$$\text{Vor}(X) = \{Z \mid \|Z - X\|_2 \leq \|Z - W\|_2 \forall W \in V\}$$

the set of points that are closest to X out of all elements in V. The total number of elements included in any cell is $\mathcal{O}\left(n^{\lceil \frac{d}{2} \rceil}\right)$ vertices, but often in practice it is $\mathcal{O}(n)$.

A basic task is point location; given a query point $X \in \mathbb{R}^d$, we find the point $Z \in V$ for which $X \in \text{Vor}(Z)$. If $d = 2$ then it requires $\mathcal{O}(n \log(n))$ time to compute the Voronoi diagram, then it takes $\mathcal{O}(\log(n))$ for each query. In $d \gg 2$ dimensions, we use a binary space partition tree (k-d tree, quad-tree, etc.) Voronoi diagrams work when d is small and only for $k = 1$ (any generalization is very computationally expensive).

k-d trees are "decision trees" i for nearest neighbor search. We choose "splitting features" with greatest width: $i = \text{argsup}_{i'} \sup_{j,k} (X_{j,i'} - X_{k,i'})$. Then the "splitting value" v is the median point for feature i, $v = \text{median}\left((X_{j,i})_j\right)$, or alternatively the half-width, $v = \frac{\sup_j X_{j,i} + \inf_j X_{j,i}}{2}$. Tree building through finding the median takes $\lfloor \log_2(n) \rfloor$ tree depth and $\mathcal{O}(nd \log(n))$ tree-building time. We can mix and match these two strategies, obtaining these run-times even with both the splitting strategies in play. Each internal node stores a sample point, and if we choose a median point for the splitting value then the point is stored in the node.

The goal is, given a query point X, we want to find $Y \in V$ such that $\|X - Y\|_2 \leq (1 + \varepsilon) \|X - Z\|_2$, where Z is the closest sample point. Setting $\varepsilon = 0$ results in the exact nearest neighbors problem, but $\varepsilon > 0$ results in the approximate nearest neighbor problem. The query algorithm stores the k nearest neighbors found so far perhaps in a max-heap, and the binary min-heap of unexplored sub-trees, valued by their distance of any point in the box represented by the subtree from X. We continually pop the first subtree from the queue, take the distance of X and this subtree, and update the minimum distance min-heap as appropriate. Then we examine the two subtrees, if they exist; if either subtree fulfills the distance condition, we add it to the queue. When the queue empties, we return the associated points to the minimum k distances.

Distance is defined in terms of any metric, not just the ℓ^2-norm.