CS 189 Lecture Notes

Alec Li

Spring 2022 — Jonathan Shewchuk

Contents

1/19/2022

Lecture 1

Introduction

1.1 Core material

What is machine learning about? In brief, finding patterns in data, and then using them to make predictions; models and statistics help us understand those patterns. Another core part of the class is optimization; optimization algorithms also allow us to "learn" the patterns.

The most important part is still data—data drives everything else; you cannot learn much if you don't have much data or if your data sucks. But with enough data, it can be fascinating what machine learning can do.

1.2 Classification

A lot of the pattern finding we do in this class is called *classification*.

The simplest kind of classification is binary classification, where there are only two possible categories.

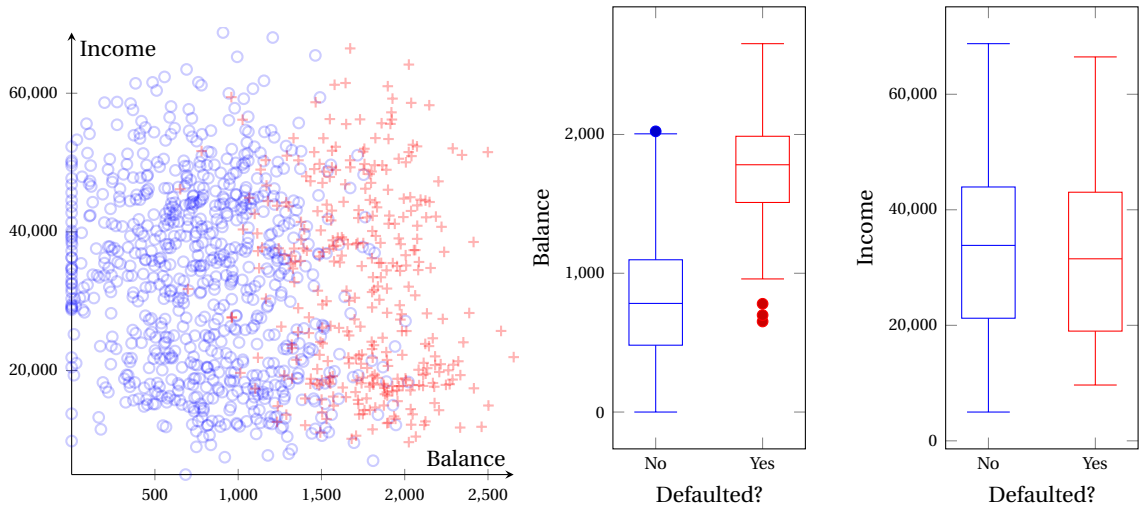


Figure 1.1: Income and balance of credit card debtors

Here, with the data in Fig. 1.1, we want to predict whether individuals default on their credit card payments. Looking at the data, it does seem like there is a pretty clean dividing line between the two categories, though it's not perfect.

What's the process of making classifications?

- Collect training data: in this case, reliable debtors and defaulted debtors.
- Evaluate new applicants (i.e. predictions)

  That is, we have a new data point in our feature space—we know their income and their balance, and we want to predict whether they default or not.

There are many ways to make predictions. One possible way is through a decision boundary; a straight line decision boundary is one of the simplest ways to do that (this is called a linear classifier). As such, making a prediction on a new point boils down to checking which side of the decision boundary the data point is on.

Another way to make predictions is to look at the nearest point in the training data, and use the same classification. This is called a nearest-neighbor classifier.

Which model is better? It depends in the data.

Over the next couple lectures, we'll learn a lot of different decision boundaries, and learn how to decide which method is better.

There are a few pros and cons for both methods:

- The nearest-neighbor classifier is always 100% correct for all training data; this is merely a result of how we constructed the decision boundary.

  In other words, the nearest-neighbor classifier has 100% training accuracy, while the linear classifier is not.

- However, the nearest-neighbor boundary seems very arbitrary and jagged; it doesn't seem realistic that the true structure follows this decision boundary. The linear classifier decision boundary may be more realistic here; there is a lot of randomness in human behavior, so it could be the most reliable we can get. As such, the nearest-neighbor method is *overfitting* to the data.

  One thing to keep in mind is that the quality of a classifier is really determined by how it behaves on *new* test data, not on already provided training data.

Instead of taking the nearest neighbor, we can take a vote from 15 nearest neighbors and take the majority answer. We'd find that the 15-nearest neighbor classifier gives a smoother curve, and as such it's more likely to reflect the real-world structure of behavior.

(Nearest-neighbors usually does well if the decision boundaries are simple and there are no outliers. We'll get back to more comparisons as we learn more classifiers.)

1.2.1 Classifying Digits

In homework 1, we'll be classifying digits with the MNIST database. As input, you'll be given a 2D array of grayscale brightness values, and we want to classify which digit this image represents.

The simplest way of evaluating the image is to treat it as a point in n^2 dimensional space (with images of size $n \times n$). With a linear decision boundary, the decision boundary is going to be a hyperplane in n^2 dimensions (the analog of a line in n^2 dimensions).

1.2.2 Testing and Validation

To *train* a classifier, we take training data and "learn" (by running some optimization algorithm) outputs.

Next, we *test* the classifier on *new* data, checking with known correct outputs.

There are two kinds of errors that we'll talk about here:

- *Training set error*: the fraction of training images not classified correctly
- *Test set error*: the fraction of test images (images not seen during training) not classified correctly

These two errors can be very different from each other, and the 1-nearest neighbors algorithm is an example of this; low training set error does not mean low test set error.

Outliers are points whose labels are atypical. As an example with credit cards, we could have solvent borrowers who defaulted anyway. With too many outliers, the 1-nearest-neighbors classifier tend to not work very well.

Overfitting is when the test error deteriorates because the classifier becomes too sensitive to outliers and other spurious patterns not important for classification.

Most ML algorithms have a few *hyperparameters* that control over/underfitting, and the k in k-nearest neighbors is an example of such a hyperparameter.

The U shape here is very common in ML; there's usually some sweet spot such that we don't underfit or overfit the data.

In *validation*, we hold back a subset of the labeled training data, called the *validation set*. Here, we train the classifier (or multiple classifiers, etc.) multiple times with different hyperparameters.

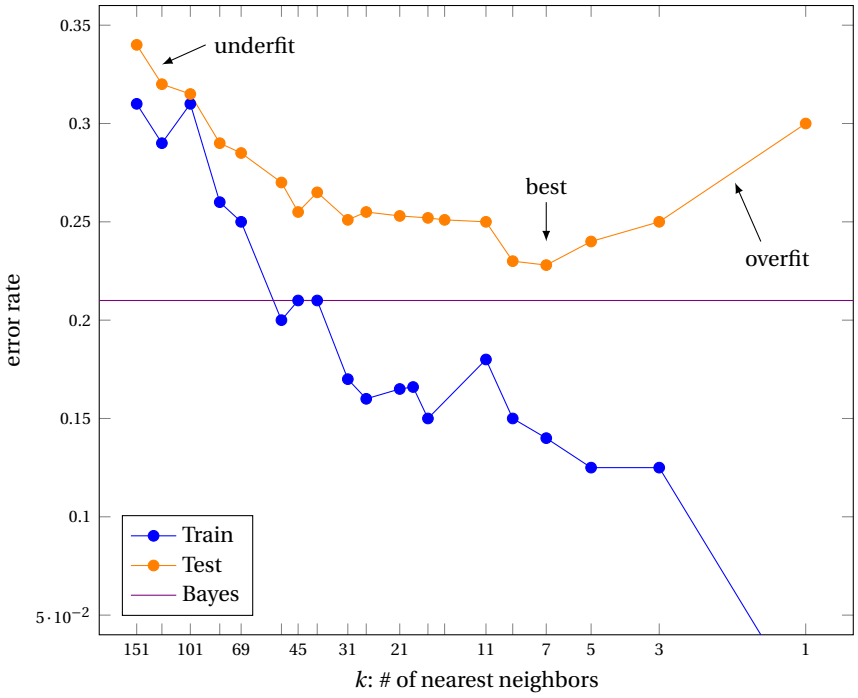


Figure 1.2: Hyperparameter tuning for k-nearest neighbors

We then evaluate the classifiers on the validation st, treating it as the test set, and we choose the setting that works best on the validation set.

As such, we have three datasets: training (to learn the model weights), validation (to tune hyperparameters/choose among different models), and test (as a *final* evaluation of the model; you don't have the data during training).

1/24/2022

Lecture 2

Classifiers

2.1 Classifiers

Today we'll talk about classifiers and the basic setup for a classifiers, and particularly linear classifiers.

With classifiers, you are given a *sample* of *n observations*, and each observation has *d features* (*predictors*) that describe the information.

In a classifier, some observations belong to a *class* C; some do not. Later, we'll consider problems with more than 2 classes, but here we'll focus on classifiers with only two classes.

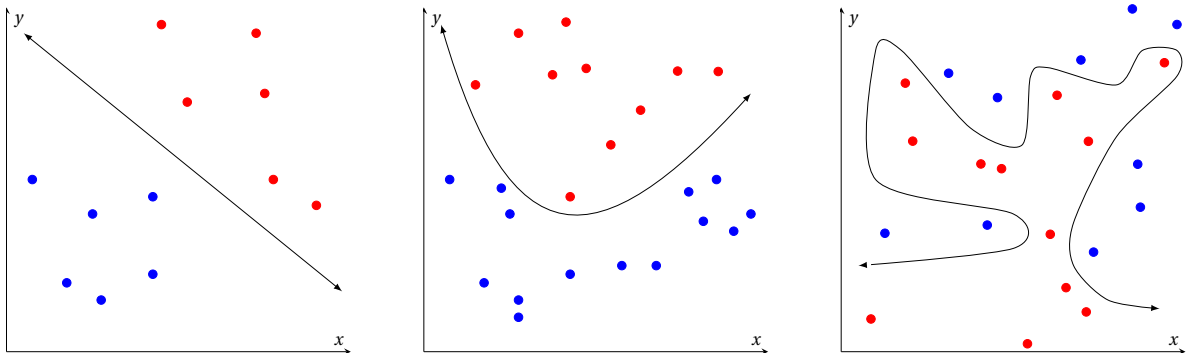
Example 2.1

For example, observations could be bank loans, and the features could be income and age of the person receiving/applying for the loan (here, $d = 2$). Some of the loans will be in the class "defaulted", and some aren't (perhaps in a class "not defaulted").

Our goal is to predict whether future borrowers will default, based on their income and age (the same features as before).

Typically, we represent each observation as a point in a d-dimensional space. There are a lot of names for this point, but we'll call these points *sample points* (other common names are *feature vectors*, and *independent variables*).

We may find that the data points in each class to be so distinct that they're easily distinguished by a linear decision boundary (Fig. 2.1a). We may also find that the data points aren't separated by a line, but can still be separated by a curve (Fig. 2.1b). If the data isn't so simple, we may find that the two classes are interspersed with one another—although we may still draw a decision boundary, it's very unlikely that it describes the structure of reality; we're probably overfitting (Fig. 2.1c).



(a) Linear decision boundary **(b)** Curve decision boundary **(c)** Overfitted decision boundary

Figure 2.1: Various kinds of decision boundaries

Definition 2.2: Decision boundary

A *decision boundary* is the boundary chosen by our classifier to separate items in class from those that are not.

Definition 2.3: Overfitting

There isn't a very precise definition for overfitting, but our intuition is that when decision boundaries become too sinuous, we don't really trust the accuracy of the boundary—it'll classify future points poorly.

(Of course, unless we have so much data that fits the decision boundary that it may actually represent reality. The real test is whether the decision boundary classifies future points well.)

We'll be particularly interested in classifiers that compute a *decision function* (not all classifiers do so). The idea is that we define a function f across the entire feature space, mapping sample points to a scalar (typically real) such that:

$$\begin{cases} f(\vec{\boldsymbol{x}}) > 0 & \text{if } x \in C \\ f(\vec{\boldsymbol{x}}) \leq 0 & \text{if } x \notin C \end{cases}$$

The decision at zero is arbitrary; we've just chosen it to represent the case that $\vec{\boldsymbol{x}}$ is not in class C here.

Decision functions have other names as well—*predictor functions*, or *discriminant functions*.

For these classifiers, the decision boundary is $\{\vec{\boldsymbol{x}} \in \mathbb{R}^d \mid f(\vec{\boldsymbol{x}}) = 0\}$. Usually, this set of points is a $d-1$ dimensional surface embedded in the d dimensional Euclidean space that is our feature space.

Definition 2.4: Isosurfaces and Isovalues

The set of points $\{\vec{\boldsymbol{x}} \mid f(\vec{\boldsymbol{x}}) = 0\}$ is called an *isosurface* of f for the *isovalue* 0.

f has other isosurfaces for other isovalues as well, defined as $\{\vec{\boldsymbol{x}} \mid f(\vec{\boldsymbol{x}}) = \alpha\}$ for the isovalue α.

For example, we could have the decision function $f(x, y) = \sqrt{x^2 + y^2} - 3$; the plot of the function and its isocontours are shown in Fig. 2.2.

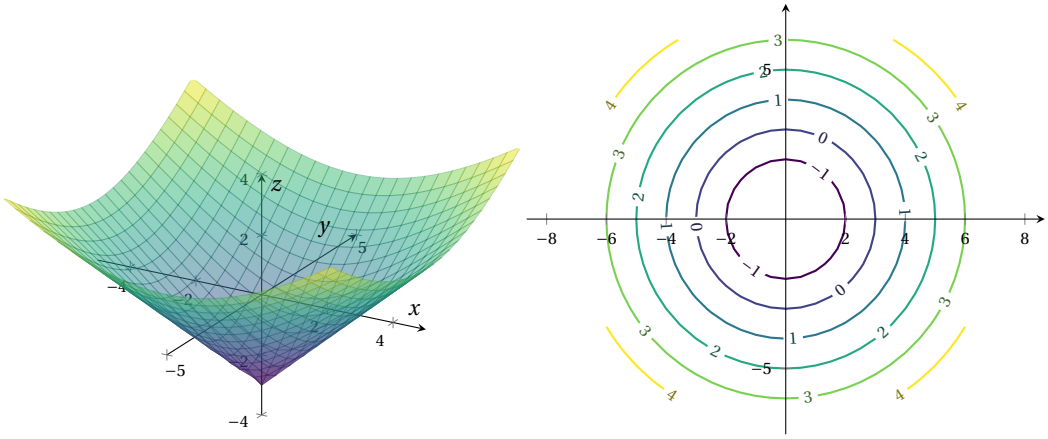


Figure 2.2: Plot and isocontours for $f(x, y) = \sqrt{x^2 + y^2} - 3$

We can also have isosurfaces of a decision function in $\mathbb{R}^d$, each of which is a sphere in $d-1$ dimensions, as shown in Fig. 2.3.

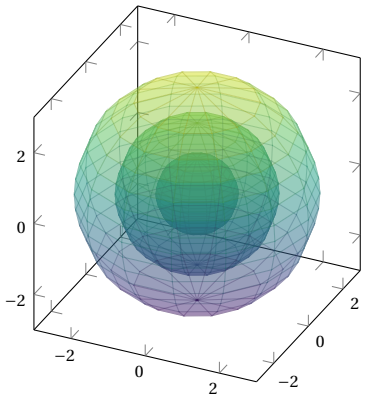


Figure 2.3: Spherical isosurfaces

For a *linear classifier*, the decision boundary is a line/plane/hyperplane.

2.2 Math Review

Briefly, we'll review some math foundations.

Vectors in this class are always assumed to be column vectors unless stated otherwise; we notate this as

$$\vec{\boldsymbol{x}} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{bmatrix}^T.$$

We'll think of $\vec{\boldsymbol{x}}$ with d elements as a point in d dimensional space.

In this class, here are some conventions:

- Uppercase roman letter (X): matrix, random variable, set

  In this set of notes, I'll be using bolded upright uppercase roman letters $\mathbf{X}$ *to represent matrices.*

- Lowercase roman letter (x): vector

  In this set of notes, I'll be using bolded italic roman letters with an arrow $\vec{\boldsymbol{x}}$ *to represent vectors.*

- Greek letter (α): scalar
- Other scalars:
  - n will always be the number of sample points
  - d will always be the number of features (dimension of sample space)
  - i, j, k will always be indices
- Functions (often scalar): $f()$, $s()$, etc.

The *inner product* (i.e. *dot product*) $\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}} = x_1 y_1 + x_2 y_2 + \cdots + x_d y_d$. This is also written $\vec{\boldsymbol{x}}^T \vec{\boldsymbol{y}}$.

Here, we have $f(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha$ as a *linear function* in $\vec{\boldsymbol{x}}$.

With (Euclidean) inner products, we also have the *Euclidean norm*:

$$\|\vec{\boldsymbol{x}}\| = \sqrt{\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{x}}} = \sqrt{x_1^2 + \cdots + x_d^2}.$$

We will refer to $\|\vec{\boldsymbol{x}}\|$ as the *length* (more precisely, the *Euclidean length*) of the vector $\vec{\boldsymbol{x}}$.

Given a vector $\vec{\boldsymbol{x}}$, $\frac{\vec{\boldsymbol{x}}}{\|\vec{\boldsymbol{x}}\|}$ is a *unit* vector (of length 1). To *normalize* a vector $\vec{\boldsymbol{x}}$, we replace $\vec{\boldsymbol{x}}$ with the unit length version $\frac{\vec{\boldsymbol{x}}}{\|\vec{\boldsymbol{x}}\|}$.

Note that the word "normalize" is overused, and there are other definitions (ex. "normalizing an image").

Besides length, we can also compute angles between vectors with dot products. We have

$$\cos\theta = \frac{\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}}}{\|\vec{\boldsymbol{x}}\| \|\vec{\boldsymbol{y}}\|} = \frac{\vec{\boldsymbol{x}}}{\|\vec{\boldsymbol{x}}\|} \cdot \frac{\vec{\boldsymbol{y}}}{\|\vec{\boldsymbol{y}}\|}.$$

The latter definition means that if we already have unit vectors, their dot product will be $\cos\theta$.

If $\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}} > 0$, the angle between them is *acute*, when $\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}} = 0$, the angle between them is a right angle, and when $\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{y}} < 0$, then the angle between them is *obtuse*.

2.3 Linear Decision Functions

Given a linear decision function $f(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha$, the decision boundary is

$$H = \{\vec{\boldsymbol{x}} \mid \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} = -\alpha\}.$$

The set H is a hyperplane (in 2D, it's a line, and in 3D, it's a plane).

Theorem 2.5: $\vec{\boldsymbol{w}}$ orthogonal to all vectors in H

Let $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$ be two points on H. Then, $\vec{\boldsymbol{w}} \cdot (\vec{\boldsymbol{y}} - \vec{\boldsymbol{x}}) = 0$.

That is, $\vec{\boldsymbol{w}}$ and $\vec{\boldsymbol{y}} - \vec{\boldsymbol{x}}$ are always orthogonal vectors—$\vec{\boldsymbol{w}}$ is orthogonal to any vector in H.

Proof. We have

$$\vec{\boldsymbol{w}} \cdot (\vec{\boldsymbol{y}} - \vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{y}} - \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} = (-\alpha) - (-\alpha) = 0.$$

□

Because of this, $\vec{w}$ is called the *normal vector* of H; we can think of $\vec{w}$ as normal/orthogonal/perpendicular to H itself. Further, $\vec{w}$ is orthogonal to the hyperplanes $\vec{w} \cdot \vec{x} = \alpha$ for any α.

Something interesting about this is if $\vec{w}$ is a unit vector, then $\vec{w} \cdot \vec{x} + \alpha$ is the *signed distance* from $\vec{x}$ to the hyperplane H. That is, the signed distance is positive on the $\vec{w}$ side of the H, and negative on the opposite side of H. Moreover, the distance from H to the origin is exactly α; just plug in $\vec{x} = \vec{0}$. As such, $\alpha = 0$ if and only if H passes through the origin.

If $\vec{w}$ is not a unit vector, then we can always rescale the equation: $\frac{\vec{w}}{\|\vec{w}\|} \cdot \vec{x} + \frac{\alpha}{\|\vec{w}\|}$ is the signed distance from $\vec{x}$ to H.

The coefficients in $\vec{w}$ along with α are called the *weights* (or *parameters*, or *regression coefficients*) of the linear classifier. (As such, the $\vec{w}$ here stands for "weights").

We call the input data *linearly separable* if there exists a hyperplane that separates all of the points in class C and not in class C.

2.4 Centroid Method

In the *centroid method*, we compute the mean $\vec{\mu}_C$ of the sample points in class C and a second mean $\vec{\mu}_X$ fo all the sample points not in class C.

We use the decision function

$$f(\vec{x}) = (\vec{\mu}_C - \vec{\mu}_X) \cdot \vec{x} - (\vec{\mu}_C - \vec{\mu}_X) \cdot \frac{\vec{\mu}_C + \vec{\mu}_X}{2}.$$

Specifically, we want the decision boundary to pass through the midpoint of the segment between $\vec{\mu}_C$ and $\vec{\mu}_X$, as shown in Fig. 2.4.

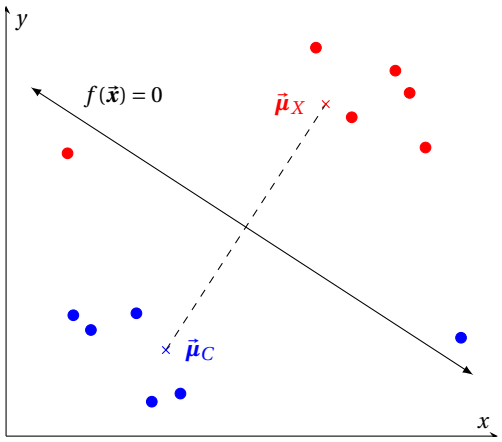


Figure 2.4: Centroid method example

Although this method has its flaws, and will likely not have 100% accuracy, the centroid method can be quite effective in some scenarios.

2.5 Perceptron Algorithm

The perceptron algorithm is slow, but correct for linearly separable points.

Particularly, the perceptron algorithm uses a numerical optimization algorithm, namely *gradient descent*; this is something very central to machine learning that we'll see many times this semester.

Let us consider n training points $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n$. (Here we use $\mathbf{X}_i$ to pick out row i of the matrix $\mathbf{X}$, with each row as a training point.)

For each training point, the *label* y_i is either 1 (if $\mathbf{X}_i \in C$) or -1 (if $\mathbf{X}_i \notin C$).

For simplicity, we will only consider decision boundaries that pass through the origin (that is, $\alpha = 0$). (We will fix this later.)

Our goal is to find a vector of weights $\vec{\boldsymbol{w}}$ such that $\mathbf{X}_i \cdot \vec{\boldsymbol{w}} \geq 0$ whenever $y_i = 1$ and $\mathbf{X}_i \cdot \vec{\boldsymbol{w}} < 0$ if $y_i = -1$. That is, a positive signed distance means we're in class C, and a negative signed distance means that we're not in class C.

With these two conditions, we can simplify this and write the equivalent inequality $y_i \mathbf{X}_i \cdot \vec{\boldsymbol{w}} \geq 0$. This inequality is called a *constraint*; we're trying to force this to be true.

The idea for the algorithm is to define a *risk function* R that is positive if some constraints are violated, and is zero if no constraints are violated. After defining this risk function, we use optimization to find a value of $\vec{\boldsymbol{w}}$ that minimizes R. This is how we train a perceptron classifier.

To define the risk function, we will first define a loss function—this applies to only one training point. The risk function will then be the sum of all loss functions over all training points. The loss function is defined as

$$L(\vec{\boldsymbol{z}}, y_i) = \begin{cases} 0 & \text{if } y_i z \geq 0 \\ -y_i z & \text{otherwise} \end{cases}$$

Here, y_i is the label and z is the classifier's prediction. Here, $y_i z \geq 0$ means that point i is classified correctly (i.e. z has the same sign as y_i), and otherwise $y_i z$ is negative, and we want to give a positive value of loss.

We then define the risk function (also called the *objective function* or the *cost function*) as:

$$R(\vec{\boldsymbol{w}}) = \frac{1}{n} \sum_{i=1}^{n} L(\mathbf{X}_i \cdot \vec{\boldsymbol{w}}, y_i) = \frac{1}{n} \sum_{i \in V} -y_i \mathbf{X}_i \cdot \vec{\boldsymbol{w}}.$$

Here, V is the set of indices for misclassified training points (i.e. where $y_i \mathbf{X}_i \cdot \vec{\boldsymbol{w}} < 0$).

If $\vec{\boldsymbol{w}}$ classifies all $\mathbf{X}_1, \ldots, \mathbf{X}_n$ correctly, then $R(\vec{\boldsymbol{w}}) = 0$; otherwise $R(\vec{\boldsymbol{w}}) > 0$, and we want to find a better $\vec{\boldsymbol{w}}$.

As such, our goal is to solve the optimization problem: $\min_{\vec{\boldsymbol{w}}} R(\vec{\boldsymbol{w}})$.

1/26/2022

Lecture 3

Gradient Descent

3.1 Perceptron Algorithm (continued)

In performing the optimization discussed last time, we're essentially making a geometric transformation from the feature space to a weight space.

The hyperplane $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{z}} = 0\}$ that separates points from the two classes gets transformed into a point $\vec{\boldsymbol{w}}$ representing the normal vector to the hyperplane. At the same time, our training points $\vec{\boldsymbol{x}}$ get transformed into a hyperplane $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{z}} = 0\}$. (Note that this exact reverse transformation isn't always true for all machine learning algorithms; just for this one.)

With this in mind, we can see that if a point $\vec{\boldsymbol{x}}$ lies on the hyperplane $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{z}} = 0\}$, then it is equivalent to say that $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} = 0$. Further, this means that the point $\vec{\boldsymbol{w}}$ lies on the hyperplane $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{z}} = 0\}$ in the weight space.

We want to enforce the inequality $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{w}} \geq 0$—in the feature space, this means that $\vec{\boldsymbol{x}}$ should be on the same side of $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{z}} = 0\}$ as $\vec{\boldsymbol{w}}$ (i.e. it classifies correctly).

Equivalently, in the weight space, we can also interpret this as saying $\vec{\boldsymbol{w}}$ should be on the same side of $\{\vec{\boldsymbol{z}} \mid \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{z}} = 0\}$ as $\vec{\boldsymbol{x}}$. An example of the relationship between the feature space and the weight space is illustrated in Fig. 3.1

All three constraints give us a slice of the plane in which where the weight vector should be in order to classify the training points correctly. Converting the weight vector back, we have our final hyperplane that correctly classifies all of the training points.

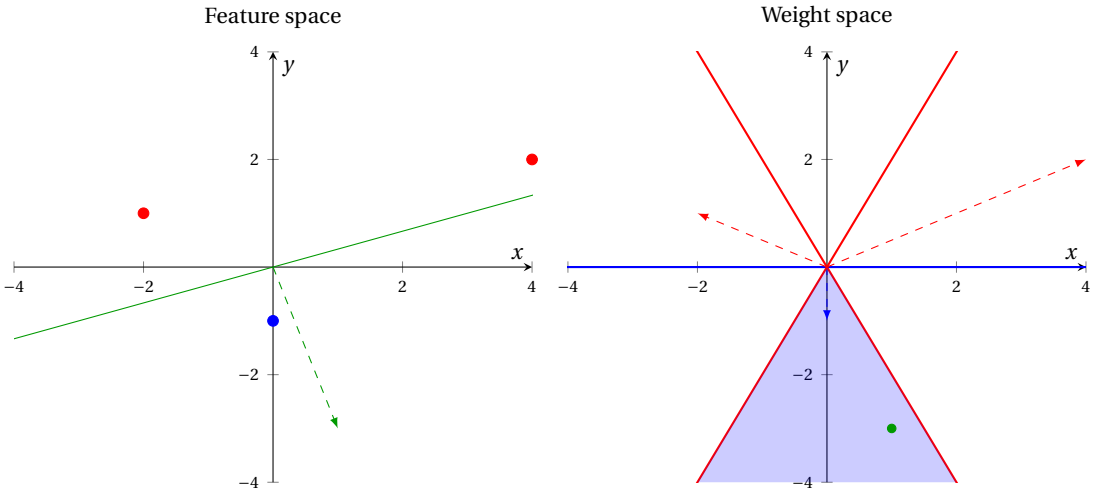


Figure 3.1: Relationship between feature space and weight space

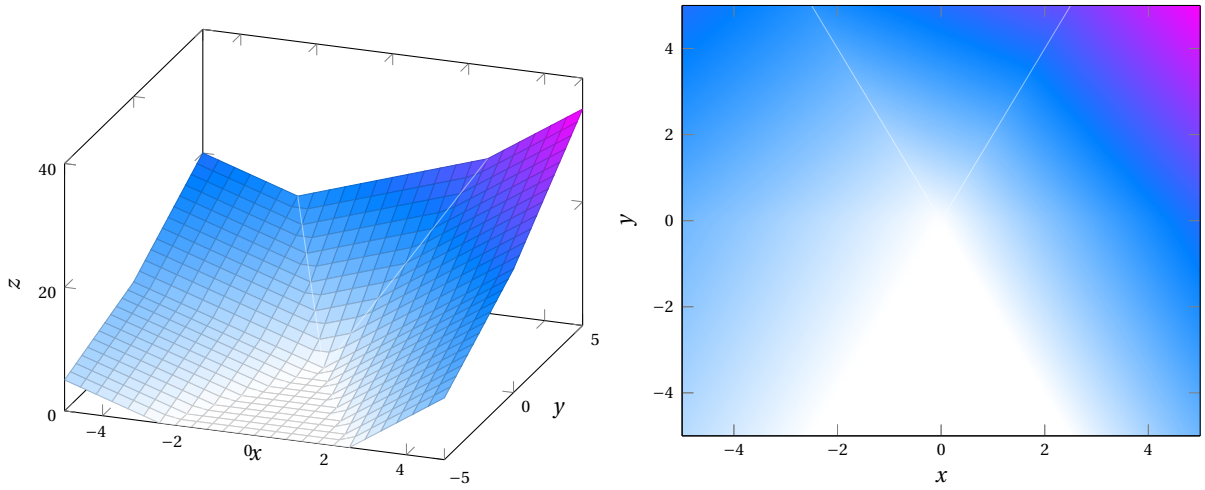


Figure 3.2: Risk function example

Let's look at the risk function that these three sample points create (Fig. 3.2).

The slice that gives zero risk will correctly classify all of the training points. To find this slice, we need an optimization algorithm to solve this problem. The algorithm we'll look at is called *gradient descent*; specifically, gradient descent on R.

3.2 Gradient Descent

Gradient descent is kind of like rolling downhill. We start at an initial point, and find the direction of steepest descent. We then take a step of some length, and repeat the process—we find a new direction of steepest descent, and move some distance in that direction.

How do we formalize this mathematically? Given a starting point $\vec{\boldsymbol{w}}$ (not equal to $\vec{\mathbf{0}}$), we find the gradient of R with respect to $\vec{\boldsymbol{w}}$. This is the direction of steepest *ascent*; to go downhill, we take a step in the opposite direction.

Mathematically, we have

$$\nabla R(\vec{\boldsymbol{w}}) = \begin{bmatrix} \frac{\partial R}{\partial w_1} \\ \frac{\partial R}{\partial w_2} \\ \vdots \\ \frac{\partial R}{\partial w_n} \end{bmatrix}.$$

Recall that the risk function is just a bunch of dot products; we have

$$\nabla_{\vec{\boldsymbol{w}}}(\vec{\boldsymbol{z}} \cdot \vec{\boldsymbol{w}}) = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix} = \vec{\boldsymbol{z}}.$$

Putting this together, we have

$$\nabla R(\vec{\boldsymbol{w}}) = \sum_{i \in V} \nabla(-y_i \mathbf{X}_i \cdot \vec{\boldsymbol{w}}) = -\sum_{i \in V} y_i \mathbf{X}_i.$$

This means that in order to walk downhill in the direction of steepest descent (i.e. $-\nabla R(\vec{\boldsymbol{w}})$)

```
w = arbitrary nonzero starting point (a good choice is any y_i X_i; this correctly classifies sample
    ↪ point i)
while R(w) > 0:
    V = set of indices i for which y_i X_i · w < 0
    w = w + ε Σ_{i∈V} y_i X_i
return w
```

Here, $\varepsilon > 0$ is the *step size*; it is also called the *learning rate*. It's hard to come up with a good ε a priori, so it's usually chosen empirically.

The issue with this algorithm is that it's slow; each step takes $O(nd)$ time.

3.2.1 Stochastic Gradient Descent

An improvement is *stochastic gradient descent*. The idea is to pick just *one* misclassified $\mathbf{X}_i$, and do gradient descent on the loss function $L(\mathbf{X}_i \cdot w, y_i)$.

This is called the *perceptron algorithm*. Each step takes $O(d)$ time, since we're only looking at one sample point (though this does not count the time it takes to find a misclassified point; it could take quite a while if there are very few misclassified points).

The modified algorithm looks like this:

```
while some y_i X_i · w < 0:
    w = w + ε y_i X_i
return w
```

While the perceptron algorithm is out of date, the gradient descent algorithm is very much used today.

3.3 Affine decision function modifications

What if the separating hyperplane does not pass through the origin?

We can add a fictitious dimension. Recall the decision function is $f(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha$. This can be cleverly rewritten as

$$f(\vec{\boldsymbol{x}}) = \begin{bmatrix} w_1 & w_2 & \cdots & w_n & \alpha \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix}.$$

With this, our sample points are essentially in $\mathbb{R}^{d+1}$, all lying on a hyperplane $x_{d+1} = 1$. Crucially, this is in the same form of our previously discussed algorithm, so we can just use this trick to solve the problem with α included.

Theorem 3.1: Linear Convergence Theorem

If the data is linearly separable, the perceptron algorithm will always be able to find a separating hyperplane. Further, it will do so in $O\left(\frac{r^2}{\gamma^2}\right)$ iterations, where $r = \max \|X_i\|_2$ and γ is the *maximum margin*.

We won't show a proof of this theorem, because the perceptron algorithm is obsolete.

Note that ε does not appear in the theorem; it does play a role, but it's complex to determine what exactly it does. If ε is too small or too large, then you'll make the perceptron algorithm very slow. If we take excessively small steps, then we'll take a very long time to get anywhere. If we take excessively large steps, then we'll keep skipping over the flat region at $R(\vec{\boldsymbol{w}}) = 0$, and it takes many iterations to land there.

3.4 Maximum Margin Classifiers

Definition 3.2: Margin

The *margin* of a linear classifier is the distance from the decision boundary to the nearest sample point.

A natural question that comes up is: what if we make the margin as wide as possible?

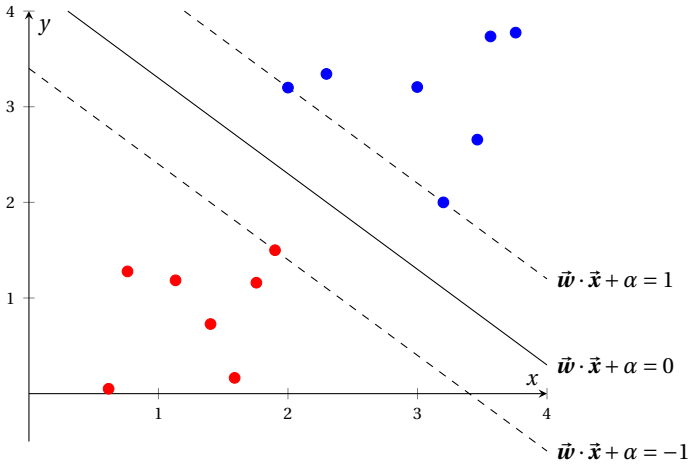


Figure 3.3: Margins in a maximum margin classifier

Looking at Fig. 3.3, we can write these two margins as $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha = 1$ and $\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha = -1$. (We use 1 here because we can always just rescale and capture that scale in $\vec{\boldsymbol{w}}$ and α.)

As such, we can enforce the constraints $y_i(\vec{\boldsymbol{w}} \cdot \mathbf{X}_i + \alpha) \geq 1$ for every sample point i.

Recall that if $\|\vec{\boldsymbol{w}}\| = 1$, then the signed distance from the decision boundary (a hyperplane) to $\mathbf{X}_i$ is $w \cdot \mathbf{X}_i + \alpha$. Otherwise, it's $\frac{w}{\|\vec{\boldsymbol{w}}\|} \cdot \mathbf{X}_i + \frac{\alpha}{\|\vec{\boldsymbol{w}}\|}$.

As such, the margin is

$$\min_i \frac{1}{\|\vec{\boldsymbol{w}}\|} |\vec{\boldsymbol{w}} \cdot \mathbf{X}_i + \alpha|.$$

We use absolute value here, because we want an absolute distance, not a signed distance. Since we've enforced the constraint that $y_i(\vec{\boldsymbol{w}} \cdot \mathbf{X}_i + \alpha) \geq 1$, then the quantity we're minimizing is always going to be $\geq \frac{1}{\|\vec{\boldsymbol{w}}\|}$

As such, we just need to minimize $\|\vec{w}\|$ in order to maximize the margin. Hence, we can write this as a formal optimization problem:

$$\begin{aligned} \min_{\vec{w},\alpha} \quad & \|\vec{w}\|^2 \\ \text{s.t.} \quad & y_i(\mathbf{X}_i \cdot \vec{w} + \alpha) \geq 1, \quad i \in [1,n] \end{aligned}$$

Here we use $\|\vec{w}\|^2$ because we want a smooth objective function. This optimization problem is called a quadratic program (QP) in $d+1$ dimensions with n constraints. If the points are linearly separable, there is one unique solution; otherwise, there are no solutions.

The solution to this QP gives us a maximum margin classifier—there is no linear classifier with a bigger margin that correctly classifies all of the sample points.

This is also called a *hard margin support vector machine* (SVM). (In contrast to the *soft margin support vector machine* that we worked with in HW 1.) Technically, this isn't quite a support vector machine just yet; we need to add features and kernels, which we'll get to in a future lecture.

It turns out that the optimal solution will always give a margin of exactly $\frac{1}{\|\vec{w}\|}$ (since we've aggressively minimized $\|\vec{w}\|$). There is a slab of width $\frac{2}{\|\vec{w}\|}$ that surrounds the decision boundary which contains no sample points.

1/31/2022

Lecture 4

Soft Margin SVMs, Features

4.1 Soft Margin SVMs

Recall from last time, with a hard-margin SVM, we specify $y_i(\mathbf{X}_i \cdot \vec{w} + \alpha) \geq 1$ in order to correctly classify all sample points.

If instead of forcing all sample points to be correctly classified, we allow for a little bit of error, we get a new set of constraints

$$y_i(\mathbf{X}_i \cdot \vec{w} + \alpha) \geq 1 - \varepsilon_i.$$

We also need $\varepsilon_i \geq 0$ for all i in order to give a little bit more leeway.

One note here is that there is no longer a "margin" here, since we're allowing leeway. As such, we're just going to define $\frac{1}{\|\vec{w}\|}$ as the margin.

Additionally, we would like to minimize the amount of slack we give, i.e. we want to minimize ε_i. Remember in hard-margin SVM, we ended up with $\min_{\vec{w},\alpha} \|\vec{w}\|^2$. To additionally minimize the slack variables, we could change this to $\min_{\vec{w},\alpha} \|\vec{w}\| + c\sum_{i=1}^n \varepsilon_i$.

We also introduce a scalar c to determine how much were weighting the slack variables vs. $\|\vec{w}\|$.

For a small c, we give less weight on the slack, so we allow points to cross the margin more; we keep more emphasis on the margin width (we focus more on having a larger margin). Since we're focusing less on trying to correctly classify all sample points, the SVM is less sensitive to outliers, and perhaps more likely to underfit to the data.

For a large c, we give more weight on the slack, so we keep ε_i's small, and put more emphasis on respecting the margin. This in turn comes with the drawback of being more sensitive to outliers, similar to hard-margin SVMs (since we're still wanting to correctly classify outliers). As such, it's more likely to overfit to the data.

The full optimization problem for a soft-margin SVM is thus

$$\begin{aligned} \min_{\vec{w},\alpha,\varepsilon_i} \quad & \|\vec{w}\|^2 + c\sum_{i=1}^n \varepsilon_i \\ \text{s.t.} \quad & y_i(\mathbf{X}_i \cdot \vec{w} + \alpha) \geq 1 - \varepsilon_i, \quad \forall i, \\ & \varepsilon_i \geq 0, \qquad \forall i \end{aligned}$$

How do we choose the c? In practice, we just search the space of possible c values and validate.

4.2 Features

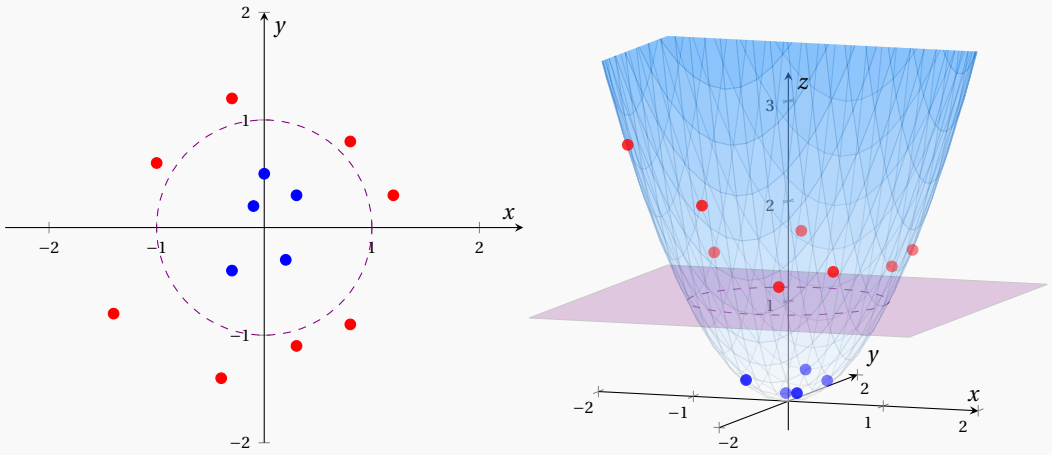
One kind of dataset that our current SVM formulation cannot deal with is if we have one class of points in a disk near the origin, and points farther away are in another class. With features, we can cover these kinds of datasets.

Here are some examples of features:

Example 4.1: Parabolic Lifting Map

A *lifting map* is a function that takes our data from $\mathbb{R}^d$ to a space of higher dimension.

In our case, we have $\Phi : \mathbb{R}^d \to \mathbb{R}^{d+1}$ such that $\Phi(x) = \begin{bmatrix} \vec{x} \\ \|\vec{x}\|^2 \end{bmatrix}$.



Theorem 4.2

Our lifted set of points $\Phi(\vec{x}_1), \ldots, \Phi(\vec{x}_n)$ is linearly separable if and only if $\vec{x}_1, \ldots, \vec{x}_n$ are separable by a hypersphere.

Proof. Briefly, we'll show a proof of the theorem. A hypersphere in $\mathbb{R}^d$ with center $\vec{c}$ and radius ρ is defined by $\|\vec{x} - \vec{c}\|^2 < \rho^2$.

We can rewrite this as

$$\begin{aligned} \|\vec{x} - \vec{c}\|^2 &< \rho^2 \\ \|\vec{x}\|^2 + \|\vec{c}\|^2 - 2\vec{c} \cdot \vec{x} &< \rho^2 \\ \begin{bmatrix} -2\vec{c}^T & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ \|\vec{x}\|^2 \end{bmatrix} &< \rho^2 - \|\vec{c}\|^2 \\ \begin{bmatrix} -2\vec{c}^T & 1 \end{bmatrix} \cdot \Phi(\vec{x}) &< \rho^2 - \|\vec{c}\|^2 \end{aligned}$$

In the last step, all we're doing is rewriting the inequality in terms of our lifting map $\Phi(\vec{x})$; we end up with a vector dot product, but in terms of $\Phi(\vec{x})$ instead of $\vec{x}$; this is a plane with respect to $\Phi(\vec{x})$. □

Example 4.3: Axis-aligned Ellipsoid/hyperboloid

In the previous example, all we did was add the Euclidean distance from the point to the origin. To generalize, we could try to use an ellipsoid (it turns out that the formulation also allows us to classify with hyperboloids.

In general, axis-aligned ellipsoids are represented by (here with three features)

$$Ax_1^2 + Bx_2^2 + Cx_3^2 + Dx_1 + Ex_2 + Fx_3 + \alpha = 0.$$

Formally, we have a lifting map $\Phi : \mathbb{R}^d \to \mathbb{R}^{2d}$ defined by

$$\Phi(\vec{x}) = \begin{bmatrix} x_1^2 & \cdots & x_d^2 & x_1 & \cdots & x_d \end{bmatrix}^T.$$

As such, our $\vec{w}$ becomes the coefficients in this general form.

Example 4.4: Ellipsoid/Hyperpoloid (General)

Here, we're going to allow the ellipsoids/hyperboloids to be rotated; they no longer need to be aligned to the axes. As such, we have extra cross terms to allow us determine the rotations. Again in three dimensions, the equation looks like

$$Ax_1^2 + Bx_2^2 + Cx_3^2 + Dx_1x_2 + Ex_2x_3 + Fx_1x_3 + Gx_1 + Hx_2 + Ix_3 + \alpha = 0.$$

Our lifting map $\Phi : \mathbb{R}^d \to \mathbb{R}^{2d+\binom{d}{2}} \sim \mathbb{R}^{d^2}$ is defined by

$$\Phi(\vec{x}) = \begin{bmatrix} x_1^2 & \cdots & x_d^2 & x_1x_2 & x_1x_3 & \cdots & x_{d-1}x_d & x_1 & \cdots & x_d \end{bmatrix}^T.$$

One issue here that may be an issue is that we're now squaring the number of dimensions. As such, even though we may get a lot of data, this can be too much for a computer to handle.

We call these general ellipsoids/hyperboloids in d-dimensions *quadrics*.

Example 4.5: Degree p polynomials

As of now, we've kept to degree 2 monomial terms. We can generalize further by looking at arbitrary degree p polynomials.

As an example, suppose we have our lifting map $\Phi(\vec{x}) : \mathbb{R}^2 \to \mathbb{R}^7$; this is defined as

$$\Phi(\vec{x}) = \begin{bmatrix} x_1^3 & x_2^3 & x_1x_2^2 & x_1^2x_2 & x_1x_2 & x_1 & x_2 \end{bmatrix}.$$

In general, we map from $\mathbb{R}^d \to \mathbb{R}^{O(d^p)}$; this is an exponential blowup in the complexity of our features.

To bypass this issue (to some extent), we'll talk about the kernel trick in a future lecture.

Further, adding an arbitrary p adds another hyperparameter that needs to be searched for, even though arbitrary degree polynomials can approximate any smooth function.

2/2/2022

Lecture 5

Machine Learning Abstractions, Numerical Optimization

Today we'll first talk a little bit about the broad abstractions that ML has. There are four generic levels of abstraction:

- **Application/Data**: This is the problem that is given to us; whether we want to classify, regress, cluster, reduce dimensionality, etc.—this is what we use to determine the algorithms to use.
- **Model**: This is the primary point of the class; these are all the kinds of models we use and decide to use. That is, what features we want, the decision functions we use, over/under-fitting, etc.
- **Optimization Problem**: This is the level at which we study a lot, and is *how* the models are implemented. This is really about transforming our model into something we can train. There are a lot of different optimization problems that we'll be going over in this class.
- **Optimization Algorithms**: This is how the optimization problems are solved; it's things like gradient descent, simplex, etc.

In this course, we'll mostly be focusing on the middle two levels. We'll talk about a few optimization algorithms, but a lot of them are way too complicated for the scope of this class. The model is ultimately what we'll be dealing with the most; we'll be learning a lot about the trade-offs and benefits of choosing one model over another.

However, today, we'll be focusing on optimization problems.

5.1 Optimization Problems

5.1.1 Unconstrained Optimization

The most generic kind of optimization problem is minimization/maximization without any constraints. We're given an objective function $f(\vec{x})$ which is smooth, and our goal is to find some $\vec{x}^*$ that minimizes f. (Maximization is just minimizing the negative, so we'll just look at the minimization problem.)

f has global minima/maxima and local minima/maxima.

Definition 5.1: Convex Function

$f(\vec{x})$ is convex if for every $\vec{x}, \vec{y} \in \mathbb{R}^d$, a line segment connecting $(\vec{x}, f(\vec{x}))$ and $(\vec{y}, f(\vec{y}))$ never goes below $f(\vec{x})$.

A continuous convex function $f(\vec{x})$ has one of:

- No minimum
- One unique global minimum
- A connected set of local minima with equal $f(\cdot)$.

Convex functions are great, because we can do calculus. Further, combinations of convex functions are usually still convex; for example, $\sum_{i=1}^{n} f_i$ with f_i each convex is still convex.

Here are some algorithms for general smooth functions f (to find local minima):

- Gradient Descent:
  - Blind gradient descent (with a learning rate)

    The reason why this is called "blind" gradient descent is because we don't exploit anything about the function; we just compute the gradient and a step size, and we just perform the algorithm.
  - Stochastic blind gradient descent
  - Gradient descent with line search

    Here, we use a search to find the best step size to use at any given point in time; this is generally an improvement over the blind versions.
- Newton's Method

With gradient descent, we're essentially using a first-order approximation of the function to give us a direction to go. We can do better if we use a higher-order approximation. Newton's method uses the Hessian matrix (i.e. second derivative), and gives us a better approximation of the function.

The drawback here is that computing the Hessian matrix can be infeasible if we have too many dimensions.

- Nonlinear conjugate gradient

There are also some algorithms for unconstrained optimization for functions that aren't necessarily smooth:

- Gradient descent

  We can use a *subgradient* if the function isn't differentiable at certain points.

- BFGS (Broyden-Fletchel-Goldfarb-Schanno)

Going back to line search, the idea is to project down to one dimension and use that result to inform your step. This is because in one dimension, it is easy to use the secant method or Newton's method, or a direct search to figure out the best step size to use.

This is a pattern that will appear again in the future—reducing the number of dimensions to simplify the algorithms.

5.2 Constrained Optimization

With constrained optimization, we are given a function $f(\vec{x})$ and some constraints $g(\vec{x}) = 0$.

We can just use Lagrange multipliers to essentially re-express this problem as an unconstrained optimization problem, in which case we can use all of our aforementioned strategies to solve it.

5.3 Linear Programs

In a linear program, we are given an objective $\vec{c} \cdot \vec{x}$, and a constraint $\mathbf{A}\vec{x} \leq \vec{b}$. Geometrically, the constraints specify a polytope in which our our optimal point must lie in. This polytope is also called the feasible region.

The feasible region is also convex; any two points in the region must lie entirely within the region.

It turns out that the optimal value must lie on the boundary of the polytope; specifically on vertices.

We know that a given point is on a vertex of the polytope when some of the constraints are *active*; that is, equality holds for those constraints.

For example, in the hard-margin SVM, the support vectors are points in which equality holds for the boundary conditions.

Here are some algorithms to solve linear programs:

- Simplex: we pick a vertex and walk the edges of the polytope to find the best vertex.
- Interior point methods

Linear programs are the easiest to solve—there are robust solvers that we can use; as such, if you can turn a problem into a linear program, then it becomes very easy and efficient to find a solution.

5.4 Quadratic Programming

In quadratic programming, we are given a function $f(\vec{x}) = \vec{x}^T\mathbf{Q}\vec{x} + \vec{c}^T\vec{x}$ and constraints $\mathbf{A}\vec{x} \leq \vec{b}$, where our goal is to minimize f. Here, we have that $\mathbf{Q} > 0$ (i.e. positive definite); that is, $\vec{x}^T\mathbf{Q}\vec{x} > 0$ for all $\vec{x} \in \mathbb{R}^n \neq \vec{0}$. This constraint on positive definiteness of $\mathbf{Q}$ gives us a nice convex problem.

As an sample, we already talked about hard/soft-margin SVMs; the one $\|\vec{w}\|^2$ term can be rewritten as $\vec{w}^T\mathbf{I}\vec{w}$, and $\mathbf{I}$ is positive definite.

There are also several algorithms to solve quadratic programs:

- Simplex
- Sequential minimal optimization (SMO)
- Coordinate descent

2/7/2022

Lecture 6

Decision theory, Generative and Discriminative Models

Today we'll be talking about a more probabilistic approach to modeling.

Example 6.1

As a motivational example, suppose we did a study trying to find a link between cancer and calorie intake:

Calories (X)	< 1200	1200-1600	> 1600
Cancer ($Y = 1$)	20%	50%	30%
No cancer ($Y = -1$)	1%	10%	89%

Suppose further that we know further that $\mathbb{P}(Y = 1) = 0.1$. That is, 10% of they population has cancer.

One formula (total probability rule) is

$$\mathbb{P}(X) = \mathbb{P}(X \mid Y = 1)\mathbb{P}(Y = 1) + \mathbb{P}(X \mid Y = -1)\mathbb{P}(Y = -1).$$

The table gives $\mathbb{P}(X \mid Y)$, and we know $\mathbb{P}(Y)$; this means that we can calculate $\mathbb{P}(X)$ individually for each category.

As a followup, suppose we want to determine the probability that someone eating 1400 calories per day has cancer? We can use Bayes' Theorem:

$$\mathbb{P}(Y \mid X) = \frac{\mathbb{P}(X \cap Y)}{\mathbb{P}(X)} = \frac{\mathbb{P}(Y)\mathbb{P}(X \mid Y)}{\mathbb{P}(X)}.$$

The actual answer is 36%; this can be interpreted as a higher probability of no cancer.

What should the diagnosis be? There are two situations if we diagnose wrong: a false negative and a false positive. With a false negative, we incorrectly determine that this person has no cancer (i.e. they actually have cancer), and a false positive means we incorrectly determine that this person does has cancer (i.e. they don't actually have cancer).

To quantify these errors (false positive and false negative), we introduce a *loss function* $L(z, y)$. Here, z is the prediction, and y is the true class.

Example 6.2

In the earlier example, it is arguably worse to say that someone that actually has cancer is incorrectly diagnosed to say that they do not have cancer.

We may give a loss function

$$L(z, y) = \begin{cases} 0 & z = y \\ 1 & z = 1,\ y = -1 \quad \text{false positive} \\ 5 & z = -1,\ y = 1 \quad \text{false negative} \end{cases}$$

This is the idea of asymmetrical loss—we penalize differently depending on whether we have a false positive or a false negative.

On the other hand, we have a notion of 0-1 loss (symmetrical loss), where we have

$$L(z,y) = \begin{cases} 1 & z \neq y \\ 0 & z = y \end{cases}$$

Example 6.3

Another example to consider is spam detection; we label spam as 1, and not spam as -1.

In this case, a false positive is worse than a false negative; it's bad to throw away a real email.

This is the opposite of what we had last time—the definition of the loss function is domain-specific, and we need to determine the appropriate loss function depending on the circumstance.

6.1 Risk

Definition 6.4: Risk

Suppose we have a classifier/decision rule $r : \mathbb{R}^d \to \pm 1$.

The *risk* of this classifier is defined as the expected loss over all values of $\vec{x}$ (features) and y (labels). That is, we define

$$R(r) = \mathbb{E}_{\vec{x},y}[L(r(\vec{x}), y)].$$

With total probability rule and the definition of expectation, we can rewrite this in another way, by splitting by X:

$$\begin{aligned} R(r) &= \mathbb{E}\big[L(r(\vec{x}), y)\big] \\ &= \sum_x \left(L(r(\vec{x}), 1)\,\mathbb{P}(Y = 1 \mid X = \vec{x}) + L(r(\vec{x}), -1)\,\mathbb{P}(Y = -1 \mid X = \vec{x})\right)\mathbb{P}(X = \vec{x}) \end{aligned}$$

We could also split by Y:

$$\begin{aligned} R(r) &= \mathbb{E}\big[L(r(\vec{x}), y)\big] \\ &= \mathbb{P}(Y = 1)\sum_{\vec{x}} L(r(\vec{x}), 1)\,\mathbb{P}(X = \vec{x} \mid Y = 1) + \mathbb{P}(Y = -1)\sum_{\vec{x}} L(r(\vec{x}), -1)\,\mathbb{P}(X = \vec{x} \mid Y = -1) \end{aligned}$$

It is important to note that this is an ideal scenario—in reality, we do not know the true probability distributions, and cannot determine these probabilities exactly. Instead, we collect data and approximate these probabilities and distributions.

6.2 Bayes Decision Rule/Bayes Classifier

Definition 6.5: bayes Decision Rule/Bayes Classifier

A Bayes classifier is a function r^* that minimizes $R(r)$.

Put another way, the Bayes classifier is the best that we can ever do.

More specifically, the Bayes classifier can be described as

$$r^*(\vec{x}) = \begin{cases} 1 & \text{if expected loss of classifying as } -1 \text{ is greater} \\ -1 & \text{if expected loss of classifying as } 1 \text{ is greater} \end{cases}$$

More mathematically, we have

$$r^*(\vec{\boldsymbol{x}}) = \begin{cases} 1 & \text{if } L(-1,1)\,\mathbb{P}(Y=1 \mid X=\vec{\boldsymbol{x}}) > L(1,-1)\,\mathbb{P}(Y=-1 \mid X=\vec{\boldsymbol{x}}) \\ -1 & \text{otherwise} \end{cases}$$

If the loss function is symmetric, i.e. $L(-1,1) = L(1,-1)$, then all we're doing is picking the class with the highest *posterior probability* $\mathbb{P}(Y \mid X = \vec{\boldsymbol{x}})$.

In reality, we can only ever estimate the posterior—we will never know the true posterior probability. The process of finding r^* is called *risk minimization*. With empirical data (which is the best we can do), it's called *empirical risk minimization*.

6.3 Continuous Distributions

Everything we've done before now has been with discrete distributions, but similar ideas can be applied to continuous cases.

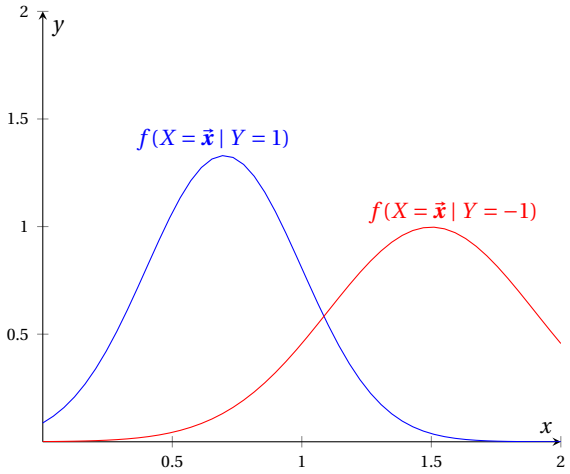


Figure 6.1: Two classes from Gaussian distributions

Suppose we have something like in Fig. 6.1. We can't just take the intersection as the boundary for the decision function; we need to weigh by their individual probabilities $P(Y=1)$ and $P(Y=-1)$.

With this in mind, we have the continuous definition of risk:

$$R(r) = \mathbb{E}\big[L(r(\vec{\boldsymbol{x}}), y)\big] = \mathbb{P}(Y=1)\int L(r(\vec{\boldsymbol{x}}),1) f(X=\vec{\boldsymbol{x}} \mid Y=1)\,\mathrm{d}x + \mathbb{P}(Y=-1)\int L(r(\vec{\boldsymbol{x}}),-1) f(X=\vec{\boldsymbol{x}} \mid Y=-1)\,\mathrm{d}x.$$

If we have 0-1 loss, then $R(r) = \mathbb{P}\big(r(\vec{\boldsymbol{x}}) \text{ is wrong}\big)$, and the Bayes decision boundary is

$$\{x : \mathbb{P}(Y=1 \mid X=\vec{\boldsymbol{x}}) = 0.5\}.$$

That is, the decision boundary is when the posterior probabilities are equal.

6.4 Ways to Build a Classifier

Generative Models A generative model tries to model the distribution of data for each class. That is, we guess the form of distribution for each class.

For each class C, we fit the distribution parameters to get $f(X \mid Y = C)$ (i.e. if we're modeling with a normal distribution, we fit to find mean and variance, etc.). We also estimate $\mathbb{P}(Y = C)$ based on the data we collect. Putting this together, Bayes' theorem gives us $\mathbb{P}(Y \mid X)$.

Using 0-1 loss, we then pick the class with the highest $\mathbb{P}(Y = C \mid X = \vec{\boldsymbol{x}})$, and equivalently $\mathbb{P}(X = \vec{\boldsymbol{x}} \mid Y = C)\,\mathbb{P}(Y = C)$

Discriminative Models Here, we model $\mathbb{P}(Y \mid X)$ directly (ex. logistic regression); these models tend to be simpler.

Specifically, the difference is that in generative models, we split this posterior probability into two parts: $\mathbb{P}(X \mid Y)$ and $\mathbb{P}(Y)$.

Building Decision Boundary In this last category, we can build the decision boundary directly (like an SVM) instead of dealing with any probabilities—that is, we model $r(\vec{x})$ directly.

6.4.1 Pros and Cons

Generative and discriminative models will also tell you the probability that the guess is incorrect/correct (with $\mathbb{P}(Y \mid X)$), since we build estimates with probabilities.

Generative models also diagnose outliers—this is when $\mathbb{P}(X)$ is small; i.e. we know the likelihood that each data point would appear.

A con of generative models is that we rely on guessing the distribution for each class; we make a lot of strong assumptions.

2/9/2022

Lecture 7

Generative Models

7.1 Gaussian Discriminant Analysis

The main idea here is that we're given data points of different classes, we fit a Gaussian distribution for each class, and then we derive the decision boundary/rule.

With QDA, each class has a different variance, and with LDA, the two classes have the same variance.

7.1.1 Gaussian Densities

Recall that if $X \in \mathbb{R}^d$ is an RV, then we say that $X \sim \mathcal{N}(\vec{\mu}, \Sigma)$ for $\vec{\mu} \in \mathbb{R}^d$ is the mean vector, and $\Sigma \in \mathbb{R}^{d \times d}$ is the covariance matrix (symmetric by definition); the covariance matrix is defined as $\mathbb{E}\left[(X - \vec{\mu})(X - \vec{\mu})^T\right]$.

The density of X is

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

If $\Sigma = \sigma^2 \mathbf{I}_d$ (i.e. i.i.d Gaussians), then $(\vec{x} - \vec{\mu})^T \Sigma^{-1} (\vec{x} - \vec{\mu}) = \left\|\vec{x} - \vec{\mu}\right\|_2^2 / \sigma^2$. This means the density is

$$f(\vec{x}) = \frac{1}{(\sigma\sqrt{2\pi})^d} \exp\left(-\frac{\left\|\vec{x} - \vec{\mu}\right\|_2^2}{2\sigma^2}\right).$$

For notational simplicity, we denote $X \sim \mathcal{N}(\vec{\mu}, \sigma^2)$ where σ^2 is a scalar. This means that the covariance matrix is $\Sigma = \sigma^2 \mathbf{I}_d$.

For each class C, we estimate $\vec{\mu}_c$, σ_c^2, and $\pi_c = \mathbb{P}(Y = C)$. We'll discuss how we make this estimate in a little bit.

Recall from last time that the Bayes decision rule/Bayes classifier is defined as

$$r^*(\vec{x}) = \begin{cases} 1 & \text{if } L(-1,1)\,\mathbb{P}(Y = 1 \mid X = \vec{x}) > L(1,-1)\,\mathbb{P}(Y = -1 \mid X = \vec{x}) \\ -1 & \text{otherwise} \end{cases}$$

As such, $r^*(x)$ returns the class C that maximizes the posterior probability $f(X = \vec{x} \mid Y = C)\pi_C$. This comes from

$$\mathbb{P}(Y = C \mid X = \vec{x}) = \frac{\mathbb{P}(X = \vec{x} \mid Y = C)\,\mathbb{P}(Y = C)}{\mathbb{P}(X = \vec{x})}.$$

Notice that we're ignoring $\mathbb{P}(X = \vec{x})$, since it is constant with respect to the maximization problem; we can safely ignore it without affecting the maximization.

This maximization problem is equivalent to maximizing

$$\begin{aligned} Q_C(\vec{x}) &= \ln\left((\sqrt{2\pi})^d f_C(\vec{x})\pi_C\right) \\ &= \ln\left(\frac{1}{\sigma^d}\exp\left(-\frac{\left\|\vec{x}-\vec{\mu}_C\right\|_2^2}{2\sigma_C^2}\right)\pi_C\right) \\ &= -\frac{\left\|\vec{x}-\vec{\mu}_C\right\|_2^2}{2\sigma_C^2} - d\ln\sigma_C + \ln\pi_C \end{aligned}$$

Notice that we've multiplied by a scalar constant in the logarithm, to cancel out the constant in the density. The last expression is proportional to the logarithm of the posterior of $\vec{x}$, which is equivalent to the original maximization problem.

As an addendum, we can write $Q_C(\vec{x})$ in the general form of a multivariate Gaussian PDF, without the assumption that the variance is of the form $\sigma_C^2\mathbf{I}$:

$$\begin{aligned} Q_C(\vec{x}) &= \ln\left(\left(\sqrt{2\pi}\right)^d f_C(\vec{x})\pi_C\right) \\ &= \ln\left(\frac{1}{|\Sigma_C|}\exp\left(-\frac{1}{2}(\vec{x}-\vec{\mu}_C)^T\Sigma_C^{-1}(\vec{x}-\vec{\mu}_C)\right)\pi_C\right) \\ &= -\frac{1}{2}(\vec{x}-\vec{\mu}_C)^T\Sigma_C^{-1}(\vec{x}-\vec{\mu}_C) - \ln|\Sigma_C| + \ln\pi_C \end{aligned}$$

7.2 Quadratic Discriminant Analysis

Suppose there are two classes C and D (the same reasoning applies for more than 2 classes, but we keep to 2 classes for simplicity).

We then have (with symmetric loss)

$$r^*(x) = \begin{cases} C & \text{if } Q_C(\vec{x}) - Q_D(\vec{x}) > 0 \\ D & \text{otherwise} \end{cases}$$

All we're doing here is replacing $\mathbb{P}(Y = C \mid X = \vec{x})$ with $Q_C(\vec{x})$ and $\mathbb{P}(Y = D \mid X = \vec{x})$ with $Q_D(\vec{x})$ respectively—$Q_C(\vec{x})$ and $Q_D(\vec{x})$ are our estimations of the probabilities. (Note that this idea still applies with asymmetric loss functions, but we'd just need to add a scalar multiplier.)

The decision boundary is at $Q_C(\vec{x}) - Q_D(\vec{x}) = 0$.

$Q_C(\vec{x})$ and $Q_D(\vec{x})$ are quadratic functions, hence the *quadratic* discriminant analysis.

As a remark, note that all of this analysis is done under the premise that the distribution of $Y \mid X$ is gaussian; this is a very strong assumption that may not be true. However, due to the central limit theorem, as long as we have enough data points, the overall distribution of the sample data should be approximately Gaussian—as such, it's generally not a bad guess to model with a Gaussian distribution, and it simplifies our calculations a lot.

Recall that an advantage of a generative model is that we know $\mathbb{P}(Y = C \mid X = \vec{x})$ exactly:

$$\begin{aligned} \mathbb{P}(Y = C \mid X = \vec{x}) &= \frac{\mathbb{P}(X = \vec{x} \mid Y = C)\pi_C}{\mathbb{P}(X = \vec{x})} \\ &= \frac{\mathbb{P}(X = \vec{x} \mid Y = C)\pi_C}{\mathbb{P}(X = \vec{x} \mid Y = C)\pi_C + \mathbb{P}(X = \vec{x} \mid X = D)\pi_D} \\ &= \frac{\exp(Q_C(\vec{x}))}{\exp(Q_C(\vec{x})) + \exp(Q_D(\vec{x}))} \\ &= \frac{1}{1 + \exp(Q_D(\vec{x}) - Q_C(\vec{x}))} \end{aligned}$$

Note that in converting to $Q_C(\vec{x})$ and $Q_D(\vec{x})$, the constant we added before gets canceled out, since all terms have the same constant. Further, we'd need to take the exponential, since we took the logarithm before.

This final result is of the form

$$s(\gamma) = \frac{1}{1+e^{-\gamma}},$$

with $\gamma = Q_D(\vec{x}) - Q_C(\vec{x})$. This is called the *sigmoid function*, or the *logistic function*.

One important propriety of the logistic function is that at zero, $s(0) = 0.5$; below zero, $s(\gamma) < 0.5$ and above zero, $s(\gamma) > 0.5$. This matches with the expected behavior.

7.3 Linear Discriminant Analysis

With LDA, we make a simplifying assumption that the Gaussians of all classes have the same variance. This means that we can simplify

$$Q_C(\vec{x}) - Q_D(\vec{x}) = \frac{\vec{\mu}_C - \vec{\mu}_D}{\sigma^2} \cdot \vec{x} - \frac{\|\vec{\mu}_C\|^2 - \|\vec{\mu}_D\|^2}{2\sigma^2} + \ln \pi_C - \ln \pi_D.$$

This is a linear function, since we have a dot product with $\vec{x}$, and a scalar unrelated to $\vec{x}$. (The derivation is left out here.)

This means that the decision boundary is linear, represented as $\vec{w} \cdot \vec{x} + \alpha = 0$. The posterior now is

$$\mathbb{P}(Y = C \mid X = \vec{x}) = s(Q_C(\vec{x}) - Q_D(\vec{x})) = s(\vec{w} \cdot \vec{x} + \alpha).$$

Specifically, we want to choose the class C with the maximal

$$Q_C(\vec{x}) = \frac{\vec{\mu}_C}{\sigma^2} \cdot \vec{x} - \frac{\|\vec{\mu}_C\|^2}{2\sigma^2} + \ln \pi_C.$$

Extending this to a multivariate Gaussian without the assumption that the variance is $\sigma_C^2 \mathbf{I}$, we have

$$Q_C(\vec{x}) = \vec{\mu}_C^T \Sigma^{-1} \vec{x} - \frac{1}{2} \vec{\mu}_C^T \Sigma^{-1} \vec{\mu}_C + \ln \pi_C.$$

7.4 Estimating Distribution Parameters

Recall that we stated before that we need to estimate $\vec{\mu}_C$, σ_C^2, and π_C for each class C; QDA and LDA both rely on the assumption that we know these parameters. How do we determine these parameters in the first place?

In general, we use maximum likelihood estimation (MLE).

7.4.1 Estimation of Priors

We'll start by going over how we estimate the priors π_C and π_D.

As a motivational example, suppose we have a binomial distribution (multinomial for more than 2 classes). We then have

$$\mathbb{P}(X = \vec{x}) = \binom{n}{x} p^x (1-p)^{n-x}.$$

Maximizing this probability with respect to p (for fixed n and x), we'd end up with $\frac{x}{n}$ as our MLE estimate. That is, we want to choose the parameters that best explain our given observations.

A similar idea can be applied with Gaussian distributions—we use the sample proportions as these prior estimates.

7.4.2 Estimation of Mean and Variance

If we have an observation of n points $\vec{x}_1, \ldots, \vec{x}_n$ all from class C, we find that the likelihood of observing all of these points is the product $f(\vec{x}_1) f(\vec{x}_2) \cdots f(\vec{x}_n)$. Again to simplify the calculations, we can convert this into a summation $\sum_{i=1}^{n} \ln f(\vec{x}_i)$.

We want

$$\max_{\vec{\mu}_C, \sigma_C^2} \sum_{i=1}^{n} \ln f(\vec{x}_i),$$

and taking the gradients and solving for when they are equal to zero, this gives us

$$\hat{\vec{\mu}} = \frac{1}{n} \sum_{i=1}^{n} \vec{x}_i$$
$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{i=1}^{n} \left\| \vec{x}_i - \vec{\mu} \right\|_2^2$$

One remark is that we don't actually know $\vec{\mu}$, so the best we can do is use our estimated $\hat{\vec{\mu}}$.

2/14/2022

Lecture 8

Eigenvectors, Anisotropic Gaussian Distributions

8.1 Eigenvectors

Consider a square matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$. $\mathbf{A}$ here defines a linear map

$$f : \mathbb{R}^d \to \mathbb{R}^d.$$

That is, the linear map preserves the following properties:

$$f(\vec{x}_1 + \vec{x}_2) = f(\vec{x}_1) + f(\vec{x}_2)$$
$$f(\alpha \vec{x}) = \alpha f(\vec{x})$$

Specifically, we're interested in the notion of eigenvectors. For a square matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$, if a vector $\vec{v} \in \mathbb{R}^d \setminus \{\vec{0}\}$ satisfies

$$\mathbf{A}\vec{x} = \lambda \vec{x},$$

then $\vec{v}$ is called an eigenvector, and λ is the corresponding eigenvalue. In other words, the direction is preserved when we multiply $\vec{v}$ by $\mathbf{A}$.

Geometrically, we can see how $\mathbf{A}$ affects its eigenvectors in Fig. 8.1. Here, we look at what multiplying by $\mathbf{A}$ does to the eigenvectors of $\mathbf{A}$ corresponding to eigenvalues $\lambda_1 = 2$ (Fig. 8.1a) and $\lambda_2 = \frac{1}{2}$ (Fig. 8.1b).

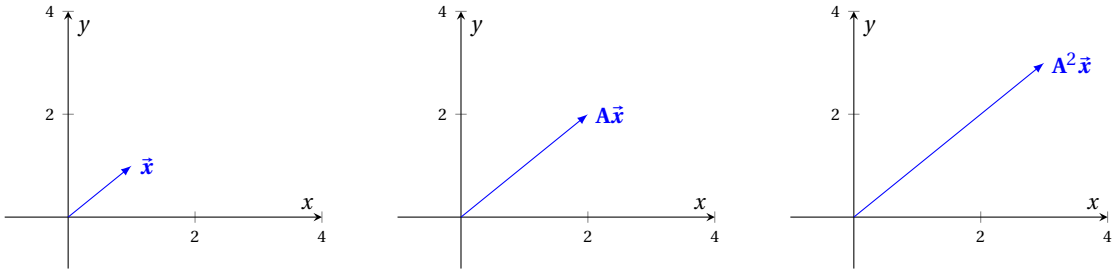
Theorem 8.1

For a matrix $\mathbf{A}$, with eigenvector $\vec{v}$ with eigenvalue λ, $\vec{v}$ is an eigenvector of $\mathbf{A}^k$, with eigenvalue λ^k.

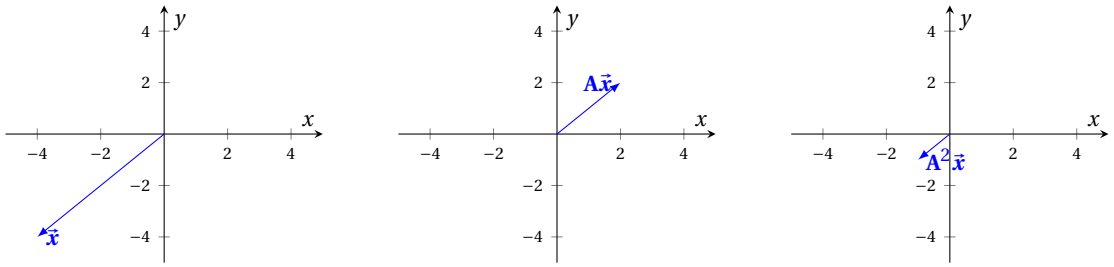
Proof. We have

$$\mathbf{A}\vec{v} = \lambda \vec{v} \implies \mathbf{A}^2 = \mathbf{A}(\mathbf{A}\vec{v}) = \mathbf{A}\lambda\vec{v} = \lambda \mathbf{A}\vec{v} = \lambda^2 \vec{v}.$$

□



(a) Applying $\mathbf{A}$ on eigenvector $\vec{x}$ for eigenvalue $\lambda_1 = 2$



(b) Applying $\mathbf{A}$ on eigenvector $\vec{x}$ for eigenvalue $\lambda_2 = -\frac{1}{2}$

Figure 8.1: Effect of $\mathbf{A}$ on its eigenvectors

Theorem 8.2

Similarly, if $\mathbf{A}$ is invertible with eigenvector $\vec{v}$, then $\vec{v}$ is also an eigenvector of $\mathbf{A}^{-1}$ with eigenvalue $\frac{1}{\lambda}$.

Proof. Since $\mathbf{A}\vec{v} = \lambda\vec{v}$, then $\vec{v} = \frac{1}{\lambda}\mathbf{A}\vec{v}$; we have

$$\mathbf{A}^{-1}\vec{v} = \mathbf{A}^{-1}\left(\frac{\mathbf{A}\vec{v}}{\lambda}\right) = \frac{1}{\lambda}\vec{v}.$$

□

Theorem 8.3: Spectral Theorem

For all real symmetric $d \times d$ matrices $\mathbf{A}$,

- All eigenvalues are real
- All eigenvectors are mutually orthogonal; that is, $\vec{v}_i^T\vec{v}_j = 0$ for all $i \neq j$

One remark: more than d eigendirections are possible. To see this, suppose $\vec{v}_1$, $\vec{v}_2$ are eigenvectors of $\mathbf{A}$ with the same eigenvalue. That is, $\mathbf{A}\vec{v}_1 = \lambda\vec{v}_1$ and $\mathbf{A}\vec{v}_2 = \lambda\vec{v}_2$. If we look at any linear combination of $\vec{v}_1$ and $\vec{v}_2$, we can see that it will also be an eigenvector; specifically,

$$\mathbf{A}(\alpha\vec{v}_1 + (1-\alpha)\vec{v}_2) = \lambda(\alpha\vec{v}_1 + (1-\alpha)\vec{v}_2).$$

That is, the linear subspace $\text{span}\{\vec{v}_1, \vec{v}_2\}$ is a subspace of eigenvectors corresponding to eigenvalue λ.

If it is the case that all eigenvalues are distinct, then we would only have d eigendirections; the eigenspaces would have dimension 1.

8.2 Quadratic Forms

We'll mainly be looking at matrices that are real and symmetric going forward. One way of understanding what real symmetric matrices are doing is by looking at *quadratic forms*.

Suppose we have $\vec{z} \in \mathbb{R}^d$, with $f(\vec{z}) = \|\vec{z}\|_2^2 = \vec{z}^T\vec{z} = \sum_{i=1}^d z_i^2$

This function f is an *isotropic* quadratic form; the scaling of each element is the same.

The *isosurfaces* of the function f are sets of the form $\{\vec{z} : f(\vec{z}) = c\}$. In our case, we have the set $\{\vec{z} : \|\vec{z}\|_2^2 = c\}$.

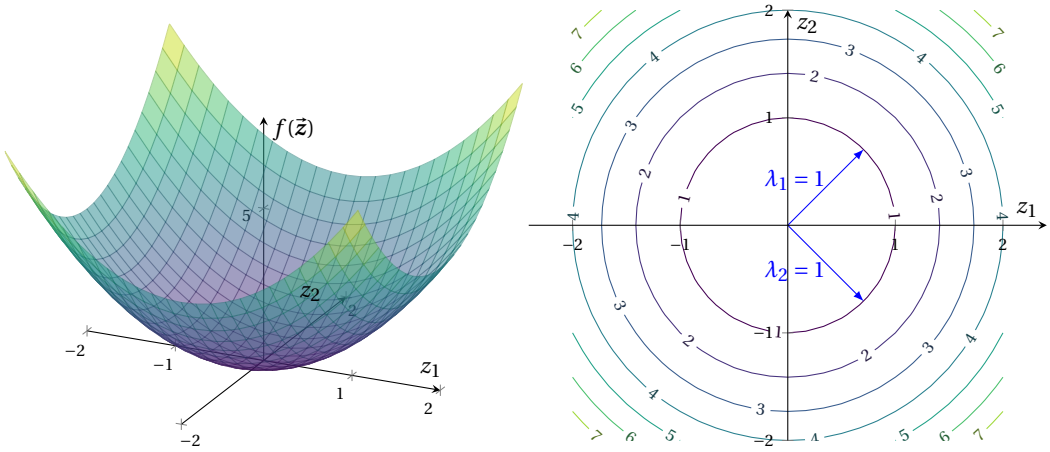


Figure 8.2: Isocontours for $f(\vec{z}) = \|\vec{z}\|_2^2$

Suppose we have $\mathbf{A} = \begin{bmatrix} \frac{3}{4} & \frac{5}{4} \\ \frac{5}{4} & \frac{3}{4} \end{bmatrix}$. We can compute that the eigenvalues of $\mathbf{A}$ are 2 and $-\frac{1}{2}$.

Suppose we define $g(\vec{x}) = \left\|\mathbf{A}^{-1}\vec{x}\right\|_2^2 = \vec{x}^T\mathbf{A}^{-2}\vec{x}$. We call this the quadratic form of $\mathbf{A}^{-2}$.

Note that the only difference between f and g is that we've substituted $\vec{z} = \mathbf{A}^{-1}\vec{x}$, or rearranged, $\vec{x} = \mathbf{A}\vec{z}$. Geometrically, this means that we've essentially mapped from the $\vec{z}$-space to the $\vec{x}$-space through the linear map defined by $\mathbf{A}$.

The diagrams in Fig. 8.3 should confirm that intuition—since $\mathbf{A}$ preserves the directions of its eigenvectors, and scales them with respect to the eigenvalues, we see that the resulting isocontours are also stretched by the eigenvalues in the corresponding directions of its eigenvectors. Here, instead of circles, we have ellipses as the contours; the axes of the ellipse (described by the eigenvectors) are scaled by the eigenvalues.

A special case here is when $\mathbf{A}$ is diagonal, ex. $\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$. Here, the eigenvectors are the coordinate axes, and geometrically, the ellipsoids are axis aligned.

8.2.1 Quadratic Form of Symmetric Matrices

Consider $\mathbf{M} \in S^d$, the set of all symmetric matrices of dimension d. Then, we have the following definitions:

- $\mathbf{M}$ is positive definite (i.e. $\vec{x}^T\mathbf{M}\vec{x} > 0$ for all nonzero $\vec{x} \in \mathbb{R}^d$) when all eigenvalues $\lambda_i > 0$.
- $\mathbf{M}$ is positive semidefinite (i.e. $\vec{x}^T\mathbf{M}\vec{x} \geq 0$ for all nonzero $\vec{x} \in \mathbb{R}^d$) when all eigenvalues $\lambda_i \geq 0$.
- $\mathbf{M}$ is indefinite when there are positive eigenvalues and negative eigenvalues.
- $\mathbf{M}$ is invertible if there are no zero eigenvalues.

Geometrically, with a positive definite matrix, there will be a unique minimum; with a positive semidefinite matrix, there may be a non-unique minimum, falling on a line. This can be seen in Fig. 8.4.

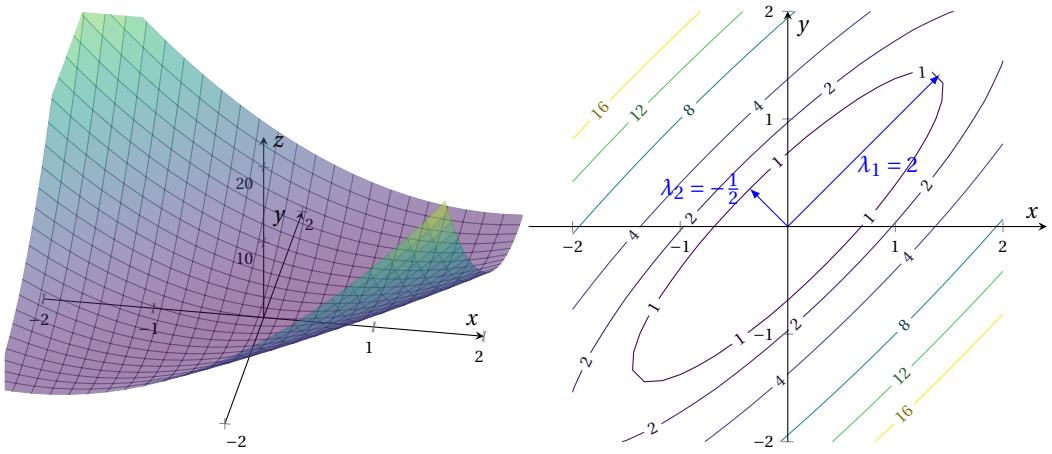


Figure 8.3: Isocontours for $g(\vec{x}) = \left\|\mathbf{A}^{-1}\vec{x}\right\|_2^2$

If we look at $\vec{x}^T\mathbf{A}^{-2}\vec{x}$, where we assume $\mathbf{A}$ is invertible, then we know that all eigenvalues are nonzero; as such, $\mathbf{A}$ is positive definite, with $\mathbf{A}^{-2}$ having eigenvalues $\frac{1}{\lambda_i^2} > 0$.

Let us look at isosurfaces of $\vec{x}^T\mathbf{M}\vec{x}$. If $\mathbf{M}$ is positive definite, then $\vec{x}^T\mathbf{M}\vec{x} > 0$ for all nonzero $\vec{x} \in \mathbb{R}^d$.

The isocontours are ellipsoids, and the radii of ellipsoids are the eigenvalues of $\mathbf{M}^{-\frac{1}{2}}$. This comes from our earlier examples of the isocontours of $\left\|\mathbf{A}^{-1}\vec{x}\right\|_2^2 = \vec{x}^T\mathbf{A}^{-2}\vec{x}$; it turned out that the axes of the ellipsoids were scaled exactly by the eigenvalues of $\mathbf{A}$.

If $\mathbf{M}$ is positive semidefinite, then $\vec{x}^T\mathbf{M}\vec{x} \geq 0$ for all $\vec{x} \in \mathbb{R}^d$. The isocontours would look like cylinders, not ellipses (the cylinders would be perpendicular to the direction of the eigenvector corresponding to $\lambda_i = 0$, as the axis has no length).

8.2.2 Constructing a Quadratic Form

We've looked at matrices $\mathbf{A}$ such that $f : \mathbb{R}^d \to \mathbb{R}^d$ is a linear map defined by $f(\vec{x}) = \mathbf{A}\vec{x}$, and looked at the directions that preserve direction (eigenvectors). That is, we looked at ways in which we define the mapping, then find the eigendirections.

Alternatively, suppose we know the directions we want to preserve (eigenvectors), and the scaling along each of the preserved directions (eigenvalues). How would we construct a matrix $\mathbf{A}$ that gives us these properties?

Here is the procedure:

1. Choose d orthogonal unit vectors, corresponding to each direction we want to preserve.

   That is, $\|\vec{v}_i\|_2^2 = 1$, and $\langle \vec{v}_i, \vec{v}_j \rangle = 0$ for all $i \neq j$.

   We then stack these $\vec{v}_i$'s as columns in a $d \times d$ matrix $\mathbf{V}$.

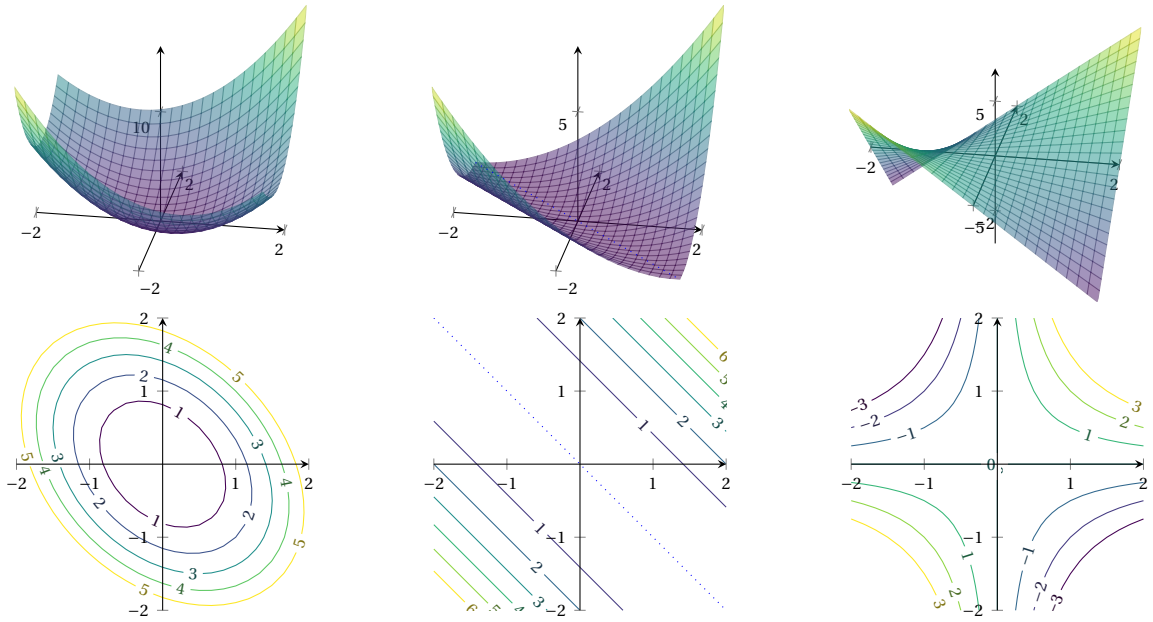
   Observe that $\mathbf{V}^T\mathbf{V} = \mathbf{V}^T\mathbf{V} = \mathbf{I}$. That is, $\mathbf{V}^T = \mathbf{V}^{-1}$.

2. Select the scaling for each of these dimensions $\lambda_i \in \mathbb{R}$, and put them into a diagonal matrix $\Lambda = \text{diag}(\lambda_i)$.

3. Then, by properties of eigenvectors, we have

   $$\mathbf{AV} = \mathbf{V}\Lambda \implies \mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^T.$$

   Notice that $\mathbf{A}$ is symmetric; $\left(\mathbf{V}\Lambda\mathbf{V}^T\right)^T = \mathbf{V}\Lambda\mathbf{V}^T$.



(a) Positive definite **(b)** Positive semidefinite **(c)** Indefinite

Figure 8.4: Various quadratic forms

Theorem 8.4: Eigendecompositions of Symmetric Matrices

For a real symmetric matrix $\mathbf{A}$, we can always write

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^T = \begin{bmatrix} \vec{\boldsymbol{v}}_1 & \cdots & \vec{\boldsymbol{v}}_d \end{bmatrix} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_d \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{v}}_1 \\ \vdots \\ \vec{\boldsymbol{v}}_d \end{bmatrix} = \sum_{i=1}^{d} \lambda_i \vec{\boldsymbol{v}}_i \vec{\boldsymbol{v}}_i^T,$$

where $\mathbf{V}$ is the orthogonal matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues.

The last summation tells us that $\mathbf{A}$ can be written as a weighted sum of rank 1 matrices $\vec{\boldsymbol{v}}_i \vec{\boldsymbol{v}}_i^T$.

This matrix factorization is called the *eigendecomposition* of $\mathbf{A}$.

Observe that $\mathbf{A}^2 = \mathbf{V}\Lambda\mathbf{V}^T\mathbf{V}\Lambda\mathbf{V}^T = \mathbf{V}\Lambda^2\mathbf{V}^T$. A similar idea can be used to compute the square root of a matrix: $\mathbf{A}^{-2} = \mathbf{V}\Lambda^{-\frac{1}{2}}\mathbf{V}^T$.

This motivates us to define the *symmetric square root* of a matrix $\mathbf{A} = \Sigma^{\frac{1}{2}}$.

If Σ is a positive semidefinite matrix, then the symmetric square root of Σ is always defined, since the eigenvalues are all nonnegative; we can take square roots. This means that the square root of Σ is just

$$\Sigma^{\frac{1}{2}} = \mathbf{V}\Lambda^{\frac{1}{2}}\mathbf{V}^T,$$

for the orthogonal matrix of eigenvectors $\mathbf{V}$ and diagonal matrix of eigenvalues Λ.

The most expensive step in this process is computing the eigenvalues and eigenvectors; the time complexity required is $O(d^3)$.

8.3 Anisotropic Gaussians

With all of this setup, we can now look at anisotropic gaussians.

Recall the univariate normal $\mathcal{N}(\mu, \sigma^2)$ with mean $\mu = \mathbb{E}[X]$ and variance $\sigma^2 = \text{Var}(X)$, and PDF

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

In the past, we've studied multivariate Gaussians of the form $X \sim \mathcal{N}(0, \sigma^2 \mathbf{I}_d)$; these are called *isotropic* Gaussians, as the variance in all directions are the same.

A natural extension to this in higher dimensions is defined by a mean vector $\mu \in \mathbb{R}^d$, and a covariance matrix Σ that is symmetric positive semidefinite (notated as $\Sigma \in S^d_{++}$). THis covariance matrix defines the covariance between the elements of the multivariate X.

The covariance matrix Σ has a few properties. It is always square and symmetric, and it is always positive definite.

The inverse of the covariance matrix Σ^{-1} is also a PSD matrix, which is called the *precision matrix*.

The anisotropic Gaussian $X \sim \mathcal{N}(\mu, \Sigma)$ with $X, \mu \in \mathbb{R}^d$ and $\Sigma \in S^d_{++}$ has PDF

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

The part in the exponent $(\vec{\boldsymbol{x}} - \mu)^T \Sigma^{-1} (\vec{\boldsymbol{x}} - \mu)$ is just a quadratic form, but shifted by μ; this expression will result in a scalar after plugging in $\vec{\boldsymbol{x}}$. The constant at front is just a normalization factor to make the PDF integrate to 1.

The negative exponential function here has a few nice properties. It is monotonic ($y > x \implies f(y) < f(x)$), and it is also convex. This means that Jensen's inequality can be applied (which only applies to convex functions):

$$f(\mathbb{E}[X]) \leq \mathbb{E}\big[f(X)\big].$$

To abstract some of this away, we can think of the multivariate Gaussian PDF as the composition of a exponential map $n : \mathbb{R} \to \mathbb{R}$ and a quadratic form $q : \mathbb{R}^d \to \mathbb{R}$. That is, $f(\vec{\boldsymbol{x}}) = n(q(\vec{\boldsymbol{x}}))$.

Here, notice that since n is a monotonic convex map, we really only need to care about the properties of $q(\vec{\boldsymbol{x}})$, as the negative exponential is a mapping that preserves all isocontours of q.

8.4 Covariance

Consider two random variables $X, Y \in \mathbb{R}^d$. We define the covariance of X and Y to be

$$\text{Cov}(X, Y) = \mathbb{E}\big[(R - \mathbb{E}[R])(Y - \mathbb{E}[Y])^T\big].$$

We also define the variance of a random variable $X \in \mathbb{R}^d$ as

$$\text{Var}(X) = \text{Cov}(X, X) = \begin{bmatrix} \text{Var}(X_1) & \text{Cov}(X_1, X_2) & \cdots \\ \text{Cov}(X_2, X_1) & \text{Var}(X_2) & \ddots \\ \vdots & \ddots & \ddots \end{bmatrix}.$$

We can see that $\text{Var}(X)$ is a symmetric positive definite matrix as well.

If X and Y are independent RVs, then $\text{Cov}(X, Y) = 0$, but it is not necessarily true that $\text{Cov}(X, Y) = 0$ implies that X and Y are independent (they can still be dependent).

However, if X and Y are both normally distributed, then $\text{Cov}(X, Y) = 0$ does in fact imply that X and Y are independent.

2/16/2022

Lecture 9

More on Anisotropic Gaussians

Recall that last time we characterized the anisotropic gaussian PDF in two parts; the composition of n and q, where n is a negative exponential, and q is the quadratic form in $\vec{x} - \mu$.

Since Σ is symmetric, we can compute its eigendecomposition $\mathbf{PDP}^T$.

One related matrix is $\Sigma^{\frac{1}{2}} = \mathbf{PD}^{\frac{1}{2}}\mathbf{P}^T$, which maps each eigenvalue to its square root $\lambda_i \mapsto \sqrt{\lambda_i}$.

Another related matrix is $\Sigma^{-1} = \mathbf{PD}^{-1}\mathbf{P}^T$, which is the precision matrix for the negative exponential.

Suppose we have the same matrix $\mathbf{A}$ as in Fig. 8.3; if we apply the negative exponential, we get the following isocontours:

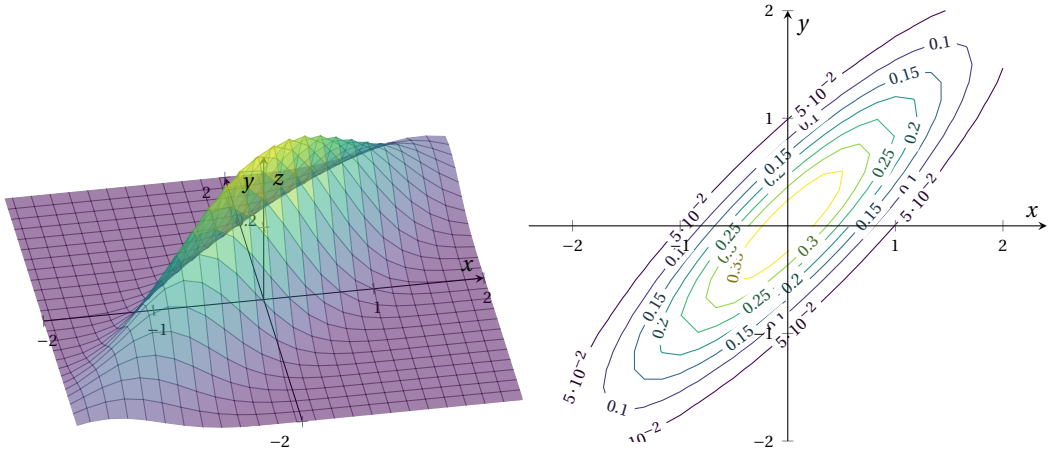


Figure 9.1: Isocontours for $\exp\left(-\left\|\mathbf{A}^{-1}\vec{x}\right\|_2^2\right)$

9.1 MLE of Anisotropic Gaussians

Given sample points $X_1, \ldots, X_n$, and classes $Y_1, \ldots, Y_n$, we want to find the best-fit Gaussians; here, X_i is a column vector.

There really isn't too much difference between the isotropic and anisotropic cases; we just use the covariance matrix Σ instead.

For individual Gaussians, we have

- QDA:
$$\hat{\sigma}_C^2 = \frac{1}{dn_C} \sum_{i:Y_i=C} \left\|X_i - \hat{\mu}\right\|_2^2.$$
- LDA:
$$\hat{\sigma}^2 = \frac{1}{dn} \sum_C \sum_{i:Y_i=c} \left\|X_i - \hat{\mu}\right\|_2^2.$$

For anisotropic Gaussians, we have

- QDA:
$$\hat{\Sigma}_i = \frac{1}{n_C} \sum_{i:Y_i=C} (X_i - \hat{\mu}_C)(X_i - \hat{\mu}_C)^T.$$
- LDA:
$$\hat{\Sigma} = \frac{1}{n} \sum_C \sum_{i:Y_i=C} (X_i - \hat{\mu}_C)(X_i - \hat{\mu}_C)^T.$$

9.1.1 QDA

With QDA, we want to choose the class C which maximizes $f(X = \vec{x} \mid Y = c)\pi_C$ for a given $\vec{x}$. The quadratic discriminant function is

$$\begin{aligned} Q_C(\vec{x}) &= \ln\left(\left(\sqrt{2\pi}\right)^d f_C(\vec{x})\pi_C\right) \\ &= -\frac{1}{2}(\vec{x}-\mu_C)^T\Sigma_C^{-1}(\vec{x}-\mu_C) - \frac{1}{2}\ln|\Sigma| + \ln\pi_C \end{aligned}$$

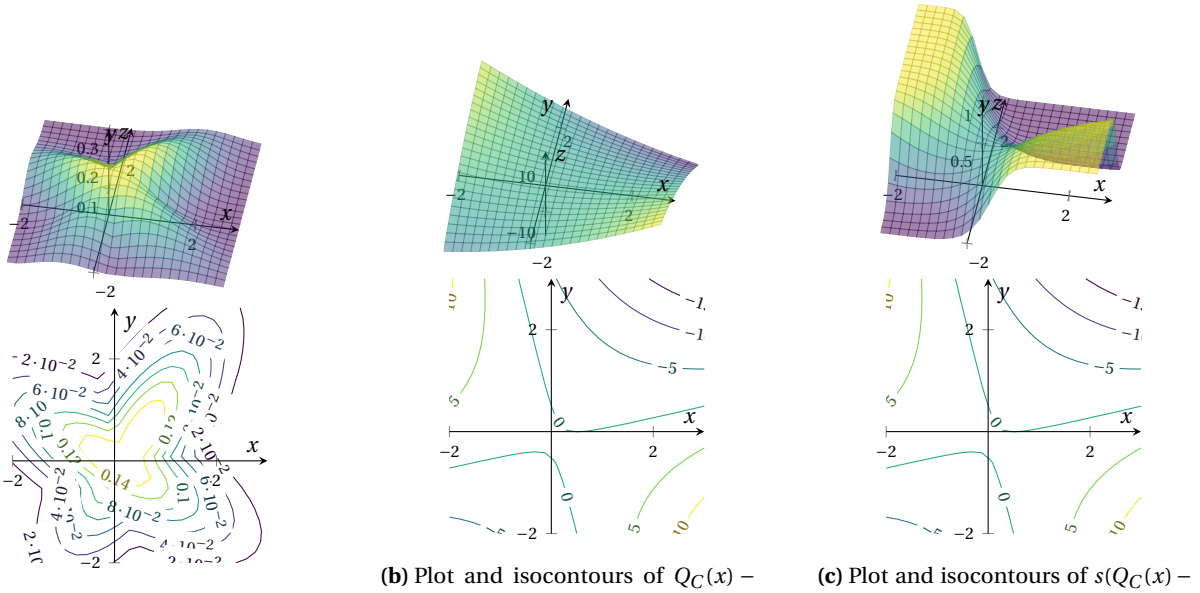
This is a quadratic in $\vec{x}$.

In the 2-class case, the decision function is $Q_C(\vec{x}) - Q_D(\vec{x})$, which is also quadratic. The decision boundary is thus $Q_C(\vec{x}) - Q_D(\vec{x}) = 0$; this boundary is also quadratic.

The posterior $\mathbb{P}(Y = C \mid X = \vec{x}) = s(Q_C(\vec{x}) - Q_D(\vec{x}))$, where s is the logistic function

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

Figure 9.2 illustrates the various functions that come up when executing QDA.



(a) Plot and isocontours of $f_C(x)$ and $f_D(x)$

(b) Plot and isocontours of $Q_C(x) - Q_D(x)$; decision boundary when $Q_C(x) - Q_D(x) = 0$

(c) Plot and isocontours of $s(Q_C(x) - Q_D(x))$; represents posterior probabilities

Figure 9.2: Example of QDA isocontours and decision boundary

9.1.2 LDA

With LDA, the only difference is that we have one common covariance matrix $\hat{\Sigma}$ across all classes.

We want to choose the class C that maximizes the linear discriminant function

$$Q_C(\vec{x}) = \mu_C^T\Sigma^{-1}\vec{x} - \frac{1}{2}\mu_C^T\Sigma^{-1}\mu_C + \ln\pi_C.$$

This means that we have

$$Q_C(\vec{x}) - Q_D(\vec{x}) = (\mu_C - \mu_D)^T\Sigma^{-1}\vec{x} - \frac{\mu_C^T\Sigma^{-1}\mu_C - \mu_D^T\Sigma^{-1}\mu_D}{2} + \ln\pi_C - \ln\pi_D.$$

This is linear in $\vec{x}$ since the quadratics cancel out.

The decision is a hyperplane of the form $\vec{w}^T\vec{x} + \alpha = 0$, and the posterior is $\mathbb{P}(Y = C \mid X = \vec{x}) = s(\vec{w}^T\vec{x} + \alpha)$.

One sidenote is that $\mu_C^T\Sigma^{-1}$ can be computed in advance using the linear system

$$\mu_C^T = \vec{v}\Sigma.$$

This is a linear system in $\vec{v}$, and can be solved for relatively easily.

9.2 LDA vs QDA

When would you want to use LDA over QDA, or QDA over LDA? We'll focus on the 2 class case for now.

For LDA,

- LDA has $d+1$ parameters; we have $\vec{w} \in \mathbb{R}^d$ and $\alpha \in \mathbb{R}$.
- LDA is less likely to overfit

For QDA,

- QDA has $\frac{d(d+3)}{2} + 1$ parameters
- QDA is more likely to overfit

With added features, LDA can give nonlinear boundaries (recall the parabolic lifting map). We can also similarly make QDA nonlinear.

Another note to make is that we don't get the *true* Bayes optimum classifier, since we've estimated distributions from finite data, and because real world data isn't always going to look gaussian.

Further, changing priors and losses don't change much; all we're doing is adding constants to our decision function. It turns out that the isovalue corresponding to probability p is the same as choosing p for false positive and $1-p$ for false negative. That is, $\pi_C = 1 - p$ and $p_D = p$.

9.3 Centering, Decorrelating, Sphering, Whitening

Suppose $\mathbf{X}$ is an $n \times d$ design matrix; each row of $\mathbf{X}$ is a sample point $\vec{x}_i^T$.

We can *center* $\mathbf{X}$ to get $\dot{\mathbf{X}}$ by computing $\vec{\mu}^T = \frac{1}{n}\sum_{i=1}^n \vec{x}_i^T$, and subtracting $\vec{\mu}_i^T$ from $\vec{x}_i^T$.

To compute the sample covariance matrix, let R be the uniform distribution on our sample points. This means that we have

$$\text{Cov}(R) = \frac{1}{n}\dot{\mathbf{X}}^T\dot{\mathbf{X}}.$$

Per class, we have

$$\hat{\Sigma}_C = \frac{1}{n_C}\dot{\mathbf{X}}_C^T\dot{\mathbf{X}}_C.$$

To *decorrelate* $\dot{\mathbf{X}}$ to get $\mathbf{Z}$, we first compute $\text{Var}(R) = \mathbf{V}\Lambda\mathbf{V}^T$, and set $\mathbf{Z} = \dot{\mathbf{X}}\mathbf{V}$. Notice that this makes $\text{Var}(\mathbf{Z}) = \Lambda$.

To *sphere* $\dot{\mathbf{X}}$ to get $\mathbf{W}$, we apply a transformation $\mathbf{W} = \dot{\mathbf{X}}\text{Var}(R)^{-\frac{1}{2}}$.

To *whiten* $\mathbf{X}$, we center and sphere $\mathbf{X} \mapsto \mathbf{W}$. This results in $\text{Cov}(\mathbf{W}) = \mathbf{I}$.

Figure 9.3 illustrates the differences between these terms.

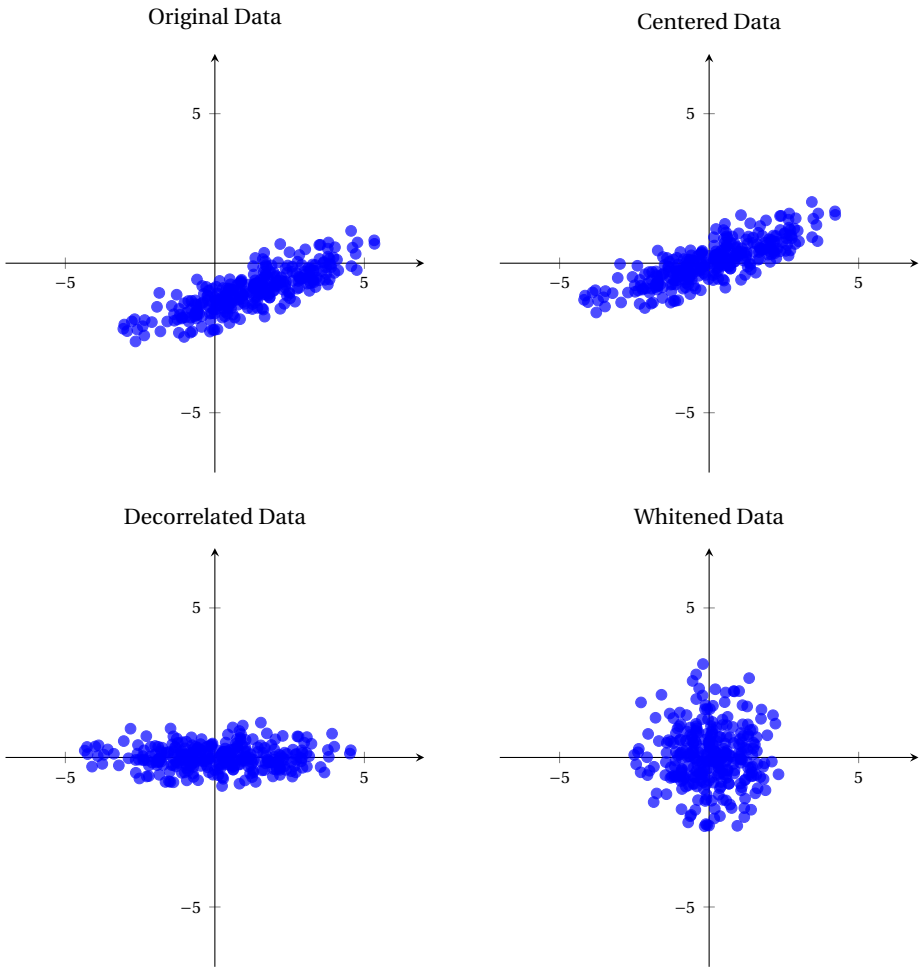


Figure 9.3: Comparisons between centered, decorrelated, and whitened data

2/23/2022

Lecture 10

Regression

10.1 Regression

We talked a lot about classification in prior lectures—we're now going to be talking about regression, or fitting curves to data.

With classification, we are given a point $\vec{\boldsymbol{x}}$, and we predict its class (which is often binary). With regression, on the other hand, we are given a point $\vec{\boldsymbol{x}}$, and we predict a numerical value on a *continuous scale*. Otherwise, regression is much like classification.

We've already seen an example of regression with GDA; LDA and QDA not only give us a classifier, but also a posterior probability—this is the probability in which our chosen class is correct.

A general recipe for regression can be thought of as follows. We can then build different regression algorithms by making different chocies:

- Choose form of regression function $h(\vec{\boldsymbol{x}}; p)$, with samples $\vec{\boldsymbol{x}}$ and parameters p. The h stands for *hypothesis*.

  We can think of the regression function as a lot like the decision function in classification.

- Choose a cost function (objective function) to optimize.

  The cost function is what we optimize, and is usually based on a loss function. A typical loss function is the risk function (i.e. expected loss).

Example 10.1: Various regresssion functions

1. Linear: $h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) = \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha$.
2. Polynomial:

   Here, we just take a linear regression function, but we use polynomial features.
3. Logistic: $h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) = s(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha)$

   This is just the logistic function $s(x) = \frac{1}{1+e^{-x}}$ applied to a linear function; this always gives us a number between 0 and 1. This means that it's useful when making regressions on probabilities.

Once we've chosen a regression function, we want to fit it to data through an optimization algorithm by minimizing some cost function; this will usually be a combination of cost functions, one for each training point.

Example 10.2: Various loss functions

Suppose z is the prediction $h(\vec{\boldsymbol{x}})$ for a sample point $\vec{\boldsymbol{x}}$, and let y be the true label. Here are some ways in which we can score the loss of the prediction z:

1. Squared error: $L(z, y) = (z - y)^2$
2. Absolute error: $L(z, y) = \left|z - y\right|$

   The absolute value function is harder to work with compared to the squared error.
3. Logistic loss, cross-entropy: $-y \ln(z) - (1 - y) \ln(1 - z)$

   Here we assume $y \in [0, 1]$ and $z \in (0, 1)$; this means that the logarithms are always negative, and makes the loss function nonnegative. Note that since the logistic regression function always gives values strictly between 0 and 1, the constraint $z \in (0, 1)$ shouldn't be a problem.

Example 10.3: Various cost functions

Here are several ways in which we can combine loss functions to create a cost function over all sample points:

1. Mean loss: $J(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(\mathbf{X}_i), y_i)$

   Note that the $\frac{1}{n}$ here is a constant and does not change which parameters minimize the cost, and as such can be omitted.
2. Maximum loss: $J(h) = \max_{i=1,\dots,n} L(h(\mathbf{X}_i), y_i)$
3. Weighted sum loss: $J(h) = \sum_{i=1}^{n} \omega_i L(h(\mathbf{X}_i), y_i)$

   Here, ω_i are the weights in the weighted sum. In cases where some data points are more important or trustworthy, we may want to assign a higher weight to them.
4. ℓ_2 penalized/regularized loss: $J(h) = (\cdots) + \lambda \|\vec{\boldsymbol{w}}\|_2^2$

   Here, we can use either one of the first three $J(h)$'s, and add an extra term to *also* minimize the size of the weights.
5. ℓ_1 penalized/regularized loss: $J(h) = (\cdots) + \lambda \|\vec{\boldsymbol{w}}\|_1$

Example 10.4: Various regression methods

Putting everything together, we have a regression method; here is a brief list of the most famous combinations:

1. Least-squares linear regression:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= (z - y)^2 \\ J(h) &= \frac{1}{n} \sum_{i=1}^{n} L(h(\mathbf{X}_i), y_i) \end{aligned}$$

2. Weighted least-squares linear regression

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= (z - y)^2 \\ J(h) &= \sum_{i=1}^{n} \omega_i L(h(\mathbf{X}_i), y_i) \end{aligned}$$

3. Ridge regression:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= (z - y)^2 \\ J(h) &= \sum_{i=1}^{n} \omega_i L(h(\mathbf{X}_i), y_i) + \lambda \|\vec{\boldsymbol{w}}\|_2^2 \end{aligned}$$

4. Lasso regression:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= (z - y)^2 \\ J(h) &= \sum_{i=1}^{n} \omega_i L(h(\mathbf{X}_i), y_i) + \lambda \|\vec{\boldsymbol{w}}\|_1 \end{aligned}$$

5. Logistic regression:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= s(\vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha) \\ L(z, y) &= -y \ln(z) - (1 - y) \ln(1 - z) \\ J(h) &= \frac{1}{n} \sum_{i=1}^{n} L(h(\mathbf{X}_i), y_i) \end{aligned}$$

6. Least absolute deviations:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= \left| z - y \right| \\ J(h) &= \frac{1}{n} \sum_{i=1}^{n} L(h(\mathbf{X}_i), y_i) \end{aligned}$$

7. Chebyshev criterion:

$$\begin{aligned} h(\vec{\boldsymbol{x}}; \vec{\boldsymbol{w}}, \alpha) &= \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} + \alpha \\ L(z, y) &= \left| z - y \right| \\ J(h) &= \max_{i=1,\ldots,n} L(h(\mathbf{X}_i), y_i) \end{aligned}$$

With the least squares regressions and the ridge regression, the cost function is quadratic; this means that we can find a closed-form solution with calculus.

With LASSO, we can express this optimization problem as a quadratic program (with an exponentially large number of constraints), but we'd need a special solver to deal with all of the constraints.

The logistic regression has a convex cost function, and as such it's not too hard to minimize—we can use gradient descent.

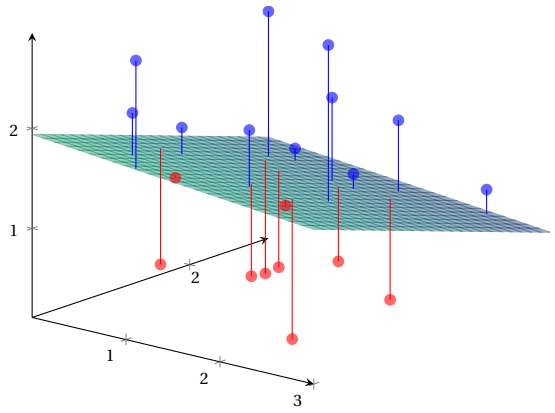
Least absolute deviations and Chebyshev criteria can both be expressed as a linear program.

10.2 Least-Squares Linear Regression

Recall that the linear regression uses a linear regression function, a squared loss function, and a mean loss function. This means that the optimization problem can be expressed as

$$\min_{\vec{\boldsymbol{w}}, \alpha} \quad \sum_{i=1}^{n} \left(\mathbf{X}_i \cdot \vec{\boldsymbol{w}} + \alpha - y_i\right)^2$$

Graphically, we have the following:



Here, we're minimizing the sums of the squares of the lengths of the data points to the fitted plane. Again, we didn't need to square the lengths; we can just use the absolute sum of the lengths, etc., but we can't just solve it with calculus anymore.

Typically, we get training points in the form of a *design matrix* $\mathbf{X}$, of dimension $n \times d$. That is, every row is a sample point $\mathbf{X}_i^T$ (we have $\mathbf{X}_i$ as a column vector, but it is a row in a design matrix), and every column is a feature.

We also have a vector $\vec{\boldsymbol{y}}$ of length N of scalar labels.

Usually, we have $n > d$.

Recall our fictitious dimension trick: we can rewrite $h(\vec{\boldsymbol{x}}) = \vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{w}} + \alpha$ by tacking on a 1 to the end of $\vec{\boldsymbol{x}}$ and α to the end of $\vec{\boldsymbol{w}}$ so that we can remove the constant term.

Now, $\mathbf{X}$ is an $n \times (d+1)$ matrix, and $\vec{\boldsymbol{w}}$ has length $d+1$.

The linear-least squares regression problem is now

$$\min_{\vec{\boldsymbol{w}}} \left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2,$$

where $\text{RSS}(\vec{\boldsymbol{w}}) = \left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2$ is the residual sum of squares.

We can optimize through calculus:

$$\left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2 = \vec{\boldsymbol{w}}^T\mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} - 2\vec{\boldsymbol{y}}^T\mathbf{X}\vec{\boldsymbol{w}} + \vec{\boldsymbol{y}}^T\vec{\boldsymbol{y}}$$

Taking the gradient, we have

$$\nabla\left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2 = 2\mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} - 2\mathbf{X}^T\vec{\boldsymbol{y}}$$

Setting the gradient to zero, we have

$$\begin{aligned} 2\mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} - 2\mathbf{X}^T\vec{\boldsymbol{y}} &= 0 \\ \mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} &= \mathbf{X}^T\vec{\boldsymbol{y}} \end{aligned}$$

This last set of linear equations is called the *normal equations*.

The $\mathbf{X}^T\mathbf{X}$ term is a $(d+1)\times(d+1)$ matrix, and on the RHS we have a vector $\mathbf{X}^T\vec{\boldsymbol{y}}$ of length $d+1$.

We know that $\mathbf{X}^T\mathbf{X}$ is PSD. If $\mathbf{X}^T\mathbf{X}$ is singular, then the problem is underconstrained. This happens when the points in feature space all lie on a common hyperplane in the feature space.

If $\mathbf{X}^T\mathbf{X}$ is invertible, then there is a unique solution

$$\vec{\boldsymbol{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\vec{\boldsymbol{y}}.$$

The term $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ is also called the *pseudoinverse* $\mathbf{X}^\dagger$ of $\mathbf{X}$, of dimension $(d+1)\times n$. This means that $\mathbf{X}^\dagger$ has the same size of $\mathbf{X}^T$.

Observe that

$$\mathbf{X}^\dagger\mathbf{X} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{X} = \mathbf{I}_{d+1}.$$

This means that $\mathbf{X}^\dagger$ is a left-inverse of $\mathbf{X}$.

Another thing to observe: now that we've found a plane that fits our data points, what labels does this function predict for our data points? The predicted values are $\hat{y}_i = \vec{\boldsymbol{w}} \cdot X_i$, or in matrix form,

$$\hat{\vec{\boldsymbol{y}}} = \mathbf{X}\vec{\boldsymbol{w}} = \mathbf{X}\mathbf{X}^\dagger\vec{\boldsymbol{y}} = \mathbf{H}\vec{\boldsymbol{y}}.$$

Here, we define $\mathbf{H} = \mathbf{X}\mathbf{X}^\dagger$, called the *hat matrix*.

Here are some advantages and disadvantages for least-squares linear regression.

Advantages:

- Easy to compute; we just need to solve a linear system.
- Usually a unique, stable solution; there are ways of handling it if the problem is underconstrained.

Disadvantages:

- Very sensitive to outliers, because errors are squared—points with large errors will dominate the loss.
- The basic form here fails if $\mathbf{X}^T\mathbf{X}$ is singular, but there are ways of handling this.

10.3 Logistic Regression

Recall that a logistic regression has a logistic function, a logistic loss function, and a mean loss cost function.

The logistic regression is usually used to fit probabilities in the range $(0,1)$. The logistic regression is also commonly used for classification; the input y_i's *can* be probabilities, but are usually just binary class markers.

Further, recall that LDA and QDA are generative models, while logistic regression is a discriminative model. With QDA/LDA, we have an intermediate step of fitting Gaussians, before finding the posterior probabilities, whereas in logistic regression we just directly fit to find the posterior probabilities.

Suppose we have $\mathbf{X}$ and $\vec{\boldsymbol{w}}$ including the fictitious dimension, with α as $\vec{\boldsymbol{w}}$'s last component. The optimization problem asks to find the $\vec{\boldsymbol{w}}$ that minimizes

$$J = \sum_{i=1}^{n} L(\mathbf{X}_i \cdot \vec{\boldsymbol{w}}, y_i) = -\sum_{i=1}^{n} \big(y_i \ln(s(X_i \cdot \vec{\boldsymbol{w}})) + (1 - y_i) \ln(1 - s(X_i \cdot \vec{\boldsymbol{w}}))\big).$$

The graphs in Fig. 10.1 show how this loss function looks like for various true labels.

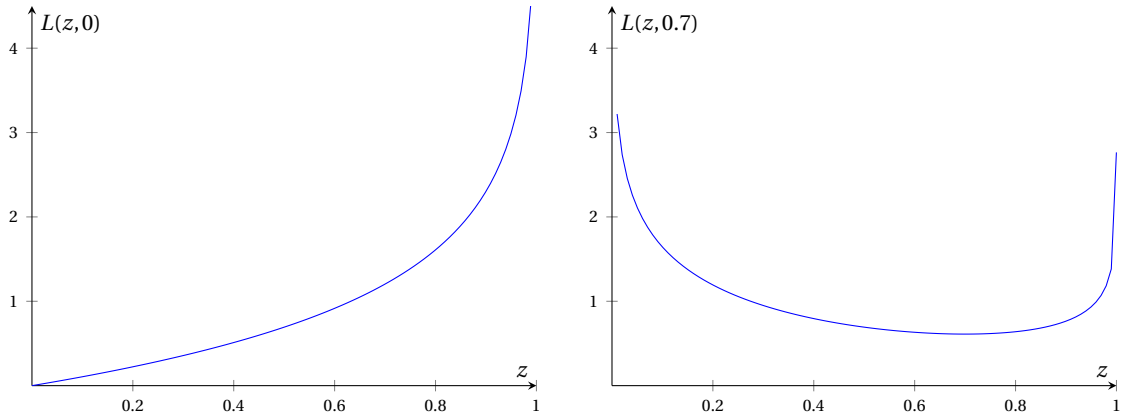


Figure 10.1: Logistic function for various true labels

Recall that $J(\vec{\boldsymbol{w}})$ is convex—this means that we can solve by gradient descent. Firstly, we can calculate the derivative of the logistic function:

$$\begin{aligned} s'(x) &= \frac{\mathrm{d}}{\mathrm{d}x} \frac{1}{1+e^{-x}} \\ &= \frac{e^{-x}}{(1+e^{-x})^2} \\ &= s(x)(1-s(x)) \end{aligned}$$

Figure 10.2 shows what the logistic function and its derivative look like.

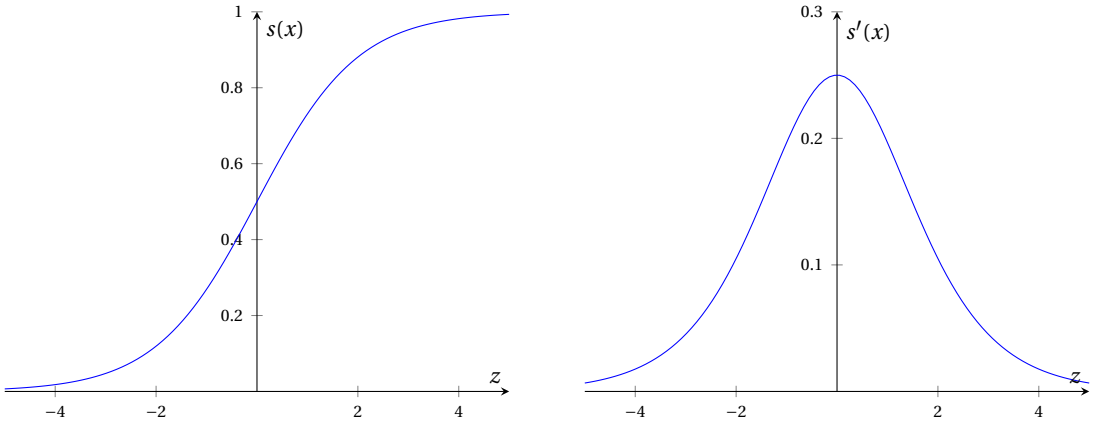


Figure 10.2: Logistic function and its derivative

Continuing with the gradient calculation, we have

$$\begin{aligned}\nabla_{\vec{w}} J &= -\sum\left(\frac{y_i}{s_i}\nabla s_i - \frac{1-y_i}{1-s_i}\nabla s_i\right)\\ &= -\sum\left(\frac{y_i}{s_i} - \frac{1-y_i}{1-s_i}\right)s_i(1-s_i)\mathbf{X}_i\\ &= -\sum(y_i - s_i)\mathbf{X}_i\\ &= -\mathbf{X}^T(\vec{y} - s(\mathbf{X}\vec{w})\end{aligned}$$

Here, we let $s(\mathbf{X}\vec{w})$ as the component-wise application of s onto the vector $\mathbf{X}\vec{w}$.

Now that we know $\nabla_{\vec{w}} J$, we can do gradient descent: $\vec{w} \leftarrow \vec{w} + \varepsilon\mathbf{X}^T(\vec{y} - s(\mathbf{X}\vec{w}))$. (Normally we'd subtract the gradient, but the gradient itself also has a negative sign.)

We can also use stochastic gradient descent, in which case we have $\vec{w} \leftarrow \vec{w} + \varepsilon(y_i - s(\mathbf{X}_i \cdot \vec{w}))\mathbf{X}_i$.

There is one big difference in applying stochastic gradient descent to logistic regression. Recall that in the perceptron algorithm, we only look at the misclassified points, since the correctly classified points have a loss of zero. However, with logistic regression, there are no points with loss of zero, and as such, we need to iterate through all of the points.

It tends to work best if we shuffle the points randomly, and process the points one by one. On large datasets, sometimes the algorithm can even converge before we visit all of the points (though this really only happens with an extremely huge dataset).

One other point is that it usually works quite well to start $\vec{w} = \vec{0}$ (unlike with the perceptron algorithm).

If the sample points turn out to be linearly separable (i.e. there exists some normal vector $\vec{w}$ such that $\vec{w} \cdot \vec{x} = 0$ separates them, with the decision boundary touching no point), logistic regression will always eventually find the separation. However, this will scale $\vec{w}$ to have infinite length, which causes $s(\mathbf{X}_i \cdot \vec{w})$ goes to 1 or 0 depending on the point's class. In this case, the cost function attains its minimum at infinity, but gradient descent will eventually get to a point where J is arbitrarily small.

2/28/2022

Lecture 11

More Regression, Newton's Method, ROC Curves

11.1 Least Squares Polynomial Regression

Here, we replace each $\mathbf{X}_i$ with a feature vector $\Phi(\mathbf{X}_i)$ with all terms of degree $0, \ldots, p$.

Example 11.1

In quadratic regression, in 2 dimensions, we'd have

$$\Phi(\mathbf{X}_i) = \begin{bmatrix} X_{i1}^2 & X_{i1}X_{i2} & X_{i2}^2 & X_{i1} & X_{i2} & 1\end{bmatrix}^T.$$

This is the same as Example 4.5, but we've also added explicitly the degree zero term.

After adding these features, we can use linear or logistic regression.

As a side note, if we use logistic regression and add quadratic features, this is equivalent to trying to fit posterior probabilities like QDA—we're fitting the logistic function applied to a quadratic function.

With polynomial regression, as the degree gets higher, it is very easy to overfit, as shown in Fig. 11.1. Higher degrees oscillate wildly through the data points, and begins to fit to the noise rather than the data.

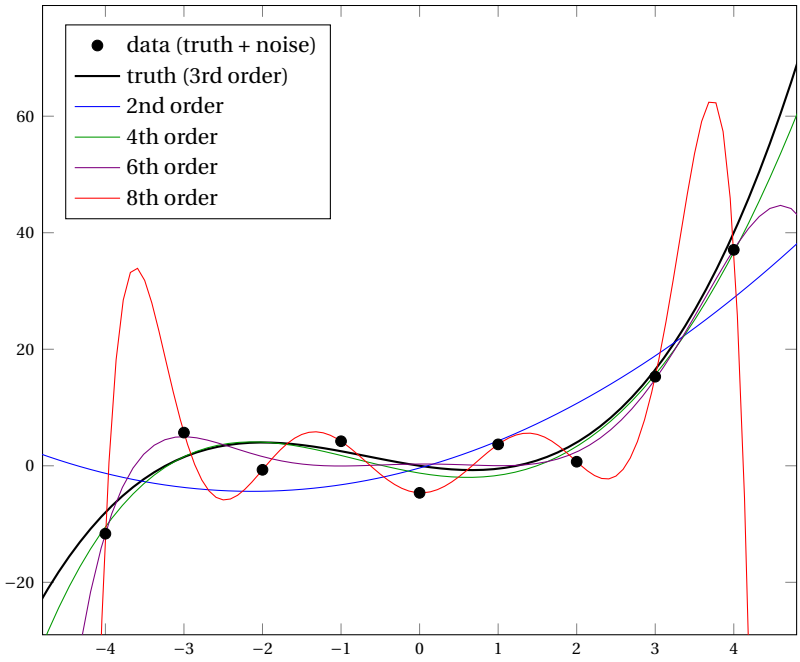


Figure 11.1: Polynomial regression and overfitting

Another note is that polynomials don't tend to capture the behavior of data outside of the domain of the data. That is, extrapolation does not work well with polynomial regression, as seen in Fig. 11.2. A polynomial regression predicts that the US population will hit zero in the year 2118 (with US population data from 1910–2020).

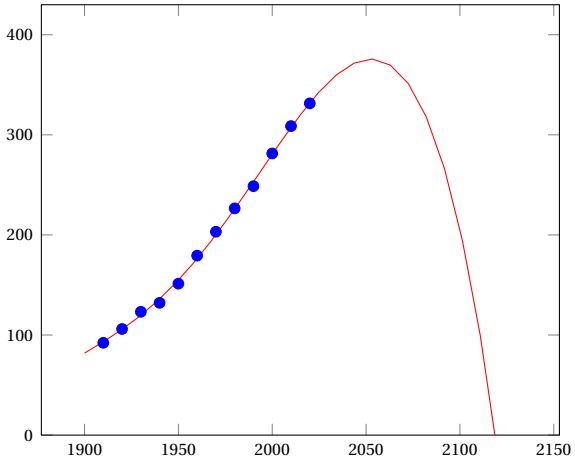


Figure 11.2: Extrapolating US Population Data

Lastly, if we have enough data points, we can tame higher degree polynomials, and they tend to do a little better. However, again, extrapolation is hard, and the polynomial does not capture data outside of the domain of data well.

11.2 Weighted Least Squares Regression

Last time, we talked about how we can weight the sample points in linear least-squares regression to tell us which are more important or more trustworthy.

Specifically, we assign each sample point a weight ω_i, and we collect all of these weights ω_i in a $n \times n$ diagonal matrix Ω.

Intuitively, a greater ω_i means that we're trying harder to minimize $\|\hat{y}_i - y_i\|^2$, i.e. the corresponding predicted $\hat{y}_i$ for the sample point $\mathbf{X}_i$ weighted by ω_i.

The full optimization problem is

$$\min_{\vec{w}} (\mathbf{X}\vec{w} - \vec{y})^T \Omega (\mathbf{X}\vec{w} - \vec{y}) = \sum_{i=1}^{n} \omega_i (\mathbf{X}_i \cdot \vec{w} - y_i)^2.$$

We can solve for $\vec{w}$ just like we did for ordinary least-squares; taking the gradient and setting it to zero, we have

$$\mathbf{X}^T \Omega \mathbf{X} \vec{w} = \mathbf{X}^T \Omega \vec{y}.$$

Notice that $\mathbf{X}^T \Omega \mathbf{X}$ is known, and $\mathbf{X}^T \Omega$ is also known—this means that we're just solving a system of linear equations, just as before.

11.3 Newton's Method

Newton's method is an iterative optimization method for optimizing smooth functions $J(\vec{w})$ (ex. its Hessian is defined everywhere). Newton's method is oftentimes much faster than gradient descent.

The idea here is that we're at a point $\vec{v}$, and we want to get closer to the minimum of J. To do so, we approximate $J(\vec{w})$ locally by a quadratic function. It's easy through calculus to determine the unique critical point of the quadratic function, and we just jump to this minimum. We repeat this process until the points converge. This process is shown in Fig. 11.3

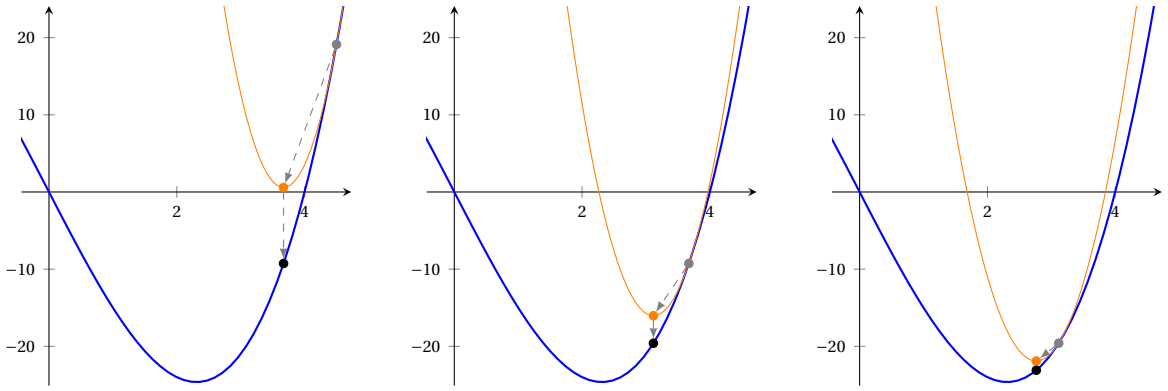


Figure 11.3: Example iterations of Newton's method

Note that some potential flaws with Newton's method is that we may not find the true minimum, or we may not find a minimum at all (ex. the quadratics become concave, and we go the opposite direction to get a local maximum)

Looking at the math of Newton's method, we take the Taylor series of J about $\vec{v}$:

$$\nabla J(\vec{w}) = \nabla J(\vec{v}) + \left(\nabla^2 J(\vec{v})\right)(\vec{w} - \vec{v}) + O\left(\|\vec{w} - \vec{v}\|^2\right).$$

Here, $\nabla^2 J(\vec{v})$ is the Hessian matrix of J; it's the multivariate version of the second derivative—each element is $\frac{\partial^2 J}{\partial v_i \partial v_j}$.

We find the critical point $\vec{w}$ by setting $\nabla J(\vec{w}) = 0$, ignoring higher order terms (which we hope to be small):

$$\vec{w} = \vec{v} - \left(\nabla^2 J(\vec{v})\right)^{-1} \nabla J(\vec{v}).$$

We then repeat this process until it converges. Since in doing so we need to compute the inverse of a matrix multiplied by a vector, we can just solve a system of linear equations to do the computation.

In general, Newton's method is

```
pick starting point w⃗
repeat until convergence:
    solve linear system (∇²J(w⃗))e⃗ = −∇J(w⃗) for e⃗
    w⃗ = w⃗ + e⃗
```

Again, a warning here is that as is, this algorithm does not know the difference between minima, maxima, and saddle points (although we can add additional code to check for this). Further, the starting point must be "close enough" to the desired critical point in order for Newton's method to work well.

Further, one edge case is if J is quadratic, then it takes one iteration to get to the minimum; if it's close to quadratic, it'll take a few iterations but it gets very close very fast.

A few differences between Newton's method and gradient descent:

- Newton's method computes the step size to take depending on the shape of the cost function, while gradient descent has a predefined step size.
- Newton's method also doesn't always go in the direction of steepest descent, whereas gradient descent always does.
- However, sometimes computing Hessians can be difficult or computationally intensive, and sometimes the Hessian may not even exist (ex. a piecewise smooth function will not work).

11.3.1 Logistic Regression

Recall that

$$s'(x) = s(x)(1 - s(x)),$$

and we define $s_i = s(\mathbf{X}_i \cdot \vec{\boldsymbol{w}})$, giving us the vector $\vec{\boldsymbol{s}}$ of s_i's. Further, recall that we have

$$\nabla_{\vec{\boldsymbol{w}}} J(\vec{\boldsymbol{w}}) = -\sum_{i=1}^{n} (y_i - s_i)\mathbf{X}_i = -\mathbf{X}^T(\vec{\boldsymbol{y}} - \vec{\boldsymbol{s}}).$$

We also need the hessian of J for Newton's method, which can be computed to be

$$\nabla^2_{\vec{\boldsymbol{w}}} J(\vec{\boldsymbol{w}}) = \sum_{i=1}^{n} s_i(1 - s_i)\mathbf{X}_i\mathbf{X}_i^T = \mathbf{X}^T\Omega\mathbf{X},$$

where $\Omega = \operatorname{diag}(s_i(1 - s_i))$. Note that Ω is always PD; $\mathbf{X}^T\Omega\mathbf{X}$ will thus always be PSD. This means that $J(\vec{\boldsymbol{w}})$ is convex

Since J is convex, it has a unique minimum, and Newton's method will always find a globally optimal point.

Knowing the gradient and Hessians, we can write Newton's method for logistic regression:

```
w⃗ = 0
repeat until convergence:
    solve (XᵀΩX)e⃗ = Xᵀ(y⃗ − s⃗) for e⃗
    w⃗ = w⃗ + e⃗
```

Note that $\vec{\boldsymbol{s}}$ and Ω are both functions are $\vec{\boldsymbol{w}}$, and as such we have a different linear system on each iteration.

Further, note that this looks very similar to weighted least squares, except there is no Ω on the RHS. Some like to say that this is an example of *iteratively reweighted least squares*, as the weights change on each iteration. Don't read too much into this though—the weights do not have the same meaning as in weighted least squares. (Smaller Ω gives larger weight here.)

Observe that misclassified points far away from the boundary have a lot of influence (because $y_i - s_i$ is large), whereas correctly classified points far away from the boundary don't have much influence (because $y_i - s_i$ is small).

11.4 LDA vs. Logistic Regression

Recall that LDA produces a posterior probability that looks like the logistic function. Logistic regression also produces a prediction for the posterior probability through the logistic function. However, the two methods produce different answers, and the two are used in different circumstances.

Advantages of LDA:

- For well separated classes, LDA tends to be stable, i.e. it gives a reliable decision boundary between the classes. Logistic regression on the other hand tends to be surprisingly unstable.
- For more than 2 classes, it is easy and elegant to use LDA, whereas logistic regression is designed more for 2-class problems. We *can* modify logistic regression (through *softmax regression*) for multiple classes, but it's not as easy or elegant.
- LDA is also slightly more accurate when classes are nearly normal, and is especially true when n is small (i.e. we don't have as much data).

Advantages of logistic regression:

- There is more emphasis on the decision boundary; it always separates linearly separable points.

  That is, correctly classified points far from the decision boundary have a small effect on the logistic regression, whereas misclassified points far from the decision boundary have the largest effect.
- Logistic regression is also more robust on some non-Gaussian distributions, especially when the distributions have a large skew.
- Logistic regression also more naturally fits labels between 0 and 1 rather than binary labels—LDA isn't designed to do this.

11.5 ROC Curves

ROC stands for *receiver operating characteristics*, which is a bad name but is used for historical reasons.

Suppose we've trained a classifier that computes a posterior probability, and we can choose a probability threshold between two classes.

ROC curves (as shown in Fig. 11.4) are used to evaluate test sets, showing the relationship between false positive and false negative rates.

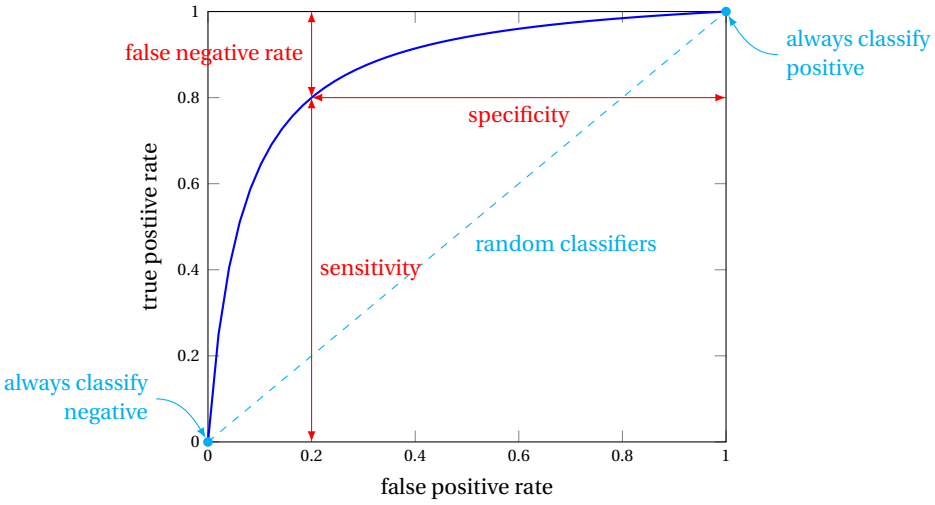


Figure 11.4: The ROC curve

Recall that the false positive rate is the percentage of sample points of negative labels classified as positive, whereas the true positive rate is the percent of positive labels correctly classified as positive. The true positive rate is also called the *sensitivity*.

Along the diagonal from $(0,0)$ to $(1,1)$ are random classifiers, and the two ends of the diagonal are classifiers that always classify negative or always classify positive, respectively.

3/12/2022

Lecture 12

Statistical Justifications, Bias-Variance Decomposition

12.1 Statistical Justifications for Regression

In the past few lectures, we've talked about regression and the specific optimization problems involved. Optimization is at the bottom of the 4 levels of machine learning abstractions, and today we'll go up to the second level: the model. Specifically, we'll talk about some models, how they lead to the aforementioned optimization problems, and how they contribute to underfitting or overfitting.

A typical model of reality that we use is as follows:

- Sample points come from an unknown probability distribution $X_i \sim D$
- y-values are the sum of an unknown non-random function and random noise; that is, for each i,

$$y_i = g(X_i) + \varepsilon_i,$$

  where each $\varepsilon_i \sim D'$, with D' having mean zero.

Here, we are assuming that reality is described by a function g. We don't know g, but g is not random—it's deterministic, representing a consistent relationship between X and y that we can estimate.

Adding g to a random variable ε represents measurement errors and all other kinds of statistical error when measuring real-world phenomena. Here, notice that the noise is independent of X; it's a pretty big assumption here, and in practice it often doesn't apply, but we make the assumption for simplicity.

Further, notice that this model does not include systematic errors, ex. when your measuring device adds one to every measurement, because we typically can't diagnose systematic errors from data alone.

The goal of regression is to find a function h that estimates g; the ideal approach is to choose

$$h(\vec{\boldsymbol{x}}) = \mathbb{E}[Y \mid X = \vec{\boldsymbol{x}}] = \mathbb{E}\big[g(\vec{\boldsymbol{x}}) + \varepsilon \mid X = \vec{\boldsymbol{x}}\big] = g(x) + \mathbb{E}[\varepsilon] = g(\vec{\boldsymbol{x}}).$$

We can retroactively define g to be this expectation; it's the expected output given our data.

12.1.1 Least-Squares Regression from Maximum Likelihood

Suppose the noise $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$; this means that $y_i \sim \mathcal{N}(g(X_i), \sigma^2)$.

Recall that the log of the normal PDF and the log likelihood is (for some constant C)

$$\begin{aligned}
\ln f(y_i) &= -\frac{(y_i - \mu)^2}{2\sigma^2} - C \\
&= -\frac{(y_i - g(X_i)^2}{2\sigma^2} - C \\
\ell(g; X, y) &= \ln\big(f(y_1) f(y_2) \cdots f(y_n)\big) \\
&= \ln f(y_1) + \ln f(y_2) + \cdots + \ln f(y_n) \\
&= -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - g(X_i))^2 - C
\end{aligned}$$

The maximum likelihood estimate on the "parameter" g is essentially the same as estimating g by least-squares regression; the final quantity we're minimizing is exactly the same form as least-squares regression. Specifically, we treat g as a "distribution parameter", choosing a g that minimizes $\sum_{i=1}^{n}(y_i - g(X_i))^2$.

This means that if the noise is normally distributed, MLE justifies using the least-squares cost function.

We've talked about how least-squares cost is very sensitive to outliers; if the error is truly normally distributed, then it's not a big deal, especially if we have a lot of sample points. However, in the real world, the distribution of outliers isn't normal—they'd come from wrong measurements, data entry errors, anomalous events, etc. This means that if we had something like a heavy-tailed distribution, least-squares is not a good choice.

12.1.2 Empirical Risk

Recall that the *risk* for the hypothesis h is the expected loss $R(h) = \mathbb{E}[L]$ over all $\vec{x} \in \mathbb{R}^d$, $y \in \mathbb{R}$.

With a discriminative model, we don't know the distribution D of X, but we want to minimize risk (with a generative model, we can just estimate the distribution and derive the expected loss). In order to approximate the distribution with a discriminative model, we'll pretend that the sample points *are* the distribution.

Definition 12.1: Empirical Distribution

We define the *empirical distribution* to be the discrete uniform distribution over the sample points.

Definition 12.2: Empirical Risk

We then define the *empirical risk* to be the expected loss under the empirical distribution:

$$\hat{R}(h) = \frac{1}{n}\sum_{i=1}^{n} L(h(X_i), y_i).$$

Oftentimes, the empirical risk is the best we can do in approximating the true statistical risk we really want to minimize. For most distributions, the empirical risk converges to the true risk as $n \to \infty$; choosing h that minimizes $\hat{R}$ is called *empirical risk minimization*.

The takeaway here is that this is why we usually minimize the sum of loss functions; it's a good approximate of the true statistical risk.

12.1.3 Logistic Loss from Maximum Likelihood

What cost function should we use for probabilities? The actual probability that the point X_i is in the class is y_i, and our predicted probability is $h(X_i)$.

Suppose we imagine β duplicate copies of X_i; this means that $y_i\beta$ are in the class, and $(1-y_i)\beta$ are not. (Here, we assume that β is large enough that $y_i\beta$ and $(1-y_i)\beta$ are both integers for all i.)

If we use maximum likelihood estimation to choose the weights that are most likely to generate this sequence of sample points and labels, we have

$$L(h; X, y) = \prod_{i=1}^{n} h(X_i)^{y_i\beta}(1 - h(X_i))^{(1-y_i)\beta}.$$

The log likelihood would then be

$$\begin{aligned}\ell(h) &= \ln L(h) \\ &= \beta\sum_{i=1}^{n}\left(y_i \ln h(X_i) + (1-y_i)\ln(1-h(X_i))\right)\end{aligned}$$

This is just the sum over logistic loss functions:

$$= -\beta \sum_{i=1}^{n} L(h(X_i), y_i)$$

This means that the maximum likelihood estimate is equivalent to minimizing the sum of logistic loss functions; this is where the logistic loss is derived from.

12.2 Bias-Variance Decomposition

There are typically 2 sources of error in a hypothesis h:

Definition 12.3: Bias

The *bias* is the error due to inability of the hypothesis h to fit g perfectly, ex. fitting a quadratic g with a linear h.

Definition 12.4: Variance

The *variance* is the error due to fitting random noise in data, ex. fitting a linear g with a linear h, but $h \neq g$.

Recall that our model is that sample points $X_i \sim D$, where the distribution D is unknown, noise $\varepsilon_i \sim D'$ where D" has mean zero, and $y_i = g(X_i) + \varepsilon_i$. We want to fit a hypothesis H to X, y.

Here, h is a random variable, i.e. its weights are random.

Suppose we consider an arbitrary point $\vec{z} \in \mathbb{R}^d$ (which is not necessarily a sample point), and we have $\gamma = g(\vec{z}) + \varepsilon$, for the random noise $\varepsilon \sim D'$. (Note that $\vec{z}$ is *arbitrary*, but γ is *random*.)

Observe that $\mathbb{E}[\gamma] = g(\vec{z})$ since $\mathbb{E}[\varepsilon] = 0$, and $\text{Var}(\gamma) = \text{Var}(\varepsilon)$ since $g(\vec{z})$ is constant.

If we're using the squared error loss, then the risk function is

$$R(h) = \mathbb{E}\big[L(h(\vec{z}), \gamma)\big].$$

Recall that h is a random variable here; we're taking the mean over the probability distribution of hypotheses. It's kind of weird at first, but remember that the training data X and y both come from probability distributions, and as such the weights that define h also come from some probability distribution, since we use the training data to find the weights.

If we rewrite the expectation of the loss, we have

$$\begin{aligned} R(h) &= \mathbb{E}\big[L(h(\vec{z}), \gamma)\big] \\ &= \mathbb{E}\big[(h(\vec{z}) - \gamma)^2\big] \\ &= \mathbb{E}\big[h(\vec{z})^2\big] + \mathbb{E}\big[\gamma^2\big] - 2\mathbb{E}\big[\gamma h(\vec{z})\big] \end{aligned}$$

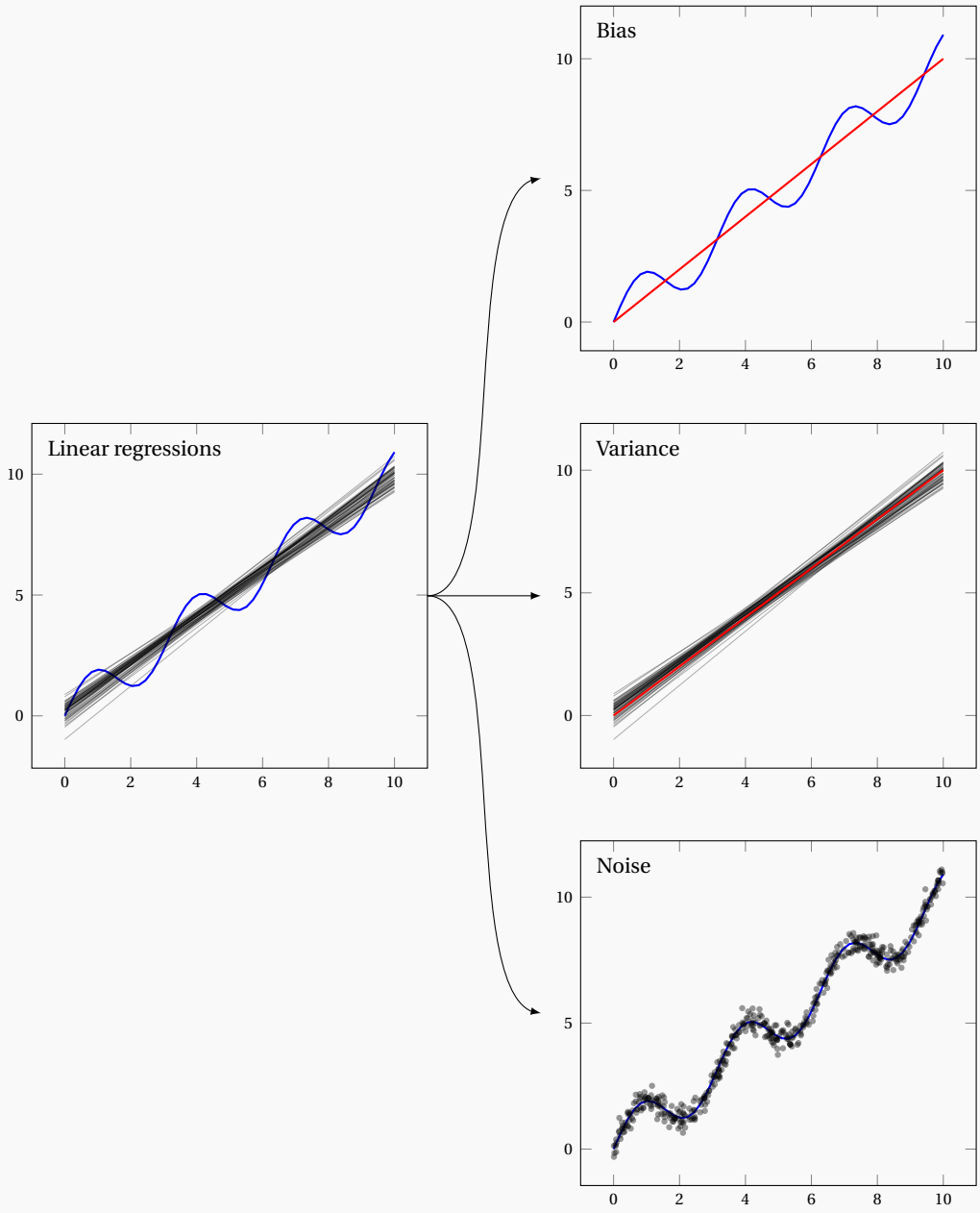
Since γ and $h(\vec{z})$ are independent, we have

$$\begin{aligned} &= \text{Var}(h(\vec{z})) + \mathbb{E}[h(\vec{z})]^2 + \text{Var}(\gamma) + \mathbb{E}\big[\gamma\big]^2 - 2\mathbb{E}\big[\gamma\big]\mathbb{E}[h(\vec{z})] \\ &= \big(\mathbb{E}[h(\vec{z})] - \mathbb{E}\big[\gamma\big]\big)^2 + \text{Var}(h(\vec{z})) + \text{Var}(\gamma) \\ &= \underbrace{\big(\mathbb{E}[h(\vec{z})] - g(\vec{z})\big)^2}_{\text{squared bias of method}} + \underbrace{\text{Var}(h(\vec{z}))}_{\text{variance of method}} + \underbrace{\text{Var}(\varepsilon)}_{\text{irreducible error}} \end{aligned}$$

This is called the *bias-variance decomposition* of the risk function.

Example 12.5

Here, we have 50 linear regressions, each from 20 data points coming from $y_i = g(x_i) + \varepsilon_i$, where the true function is $g(x) = x + \sin(2x)$ and the noise is drawn from $\varepsilon_i \sim \mathcal{N}(0, 0.25)$.



The top graph shows the expected linear estimator (in red), and the true function (in blue); the bias is the vertical distance between the expected estimator and the true function (specifically, squared bias is the squared distance). It should be clear that most points have a large bias here; a line does not fit sine waves very well. However, there are certain points in which we have a small bias; these are points in which the sine wave crosses the line.

The middle graph shows the expected linear estimator (in red), and all of the linear regressions (in black); the variance is the expected squared deviation between a given line and the expected linear estimator (at a point z).

The bottom graph shows the true function (in blue) and a scatter plot of the noise; the irreducible error is the expected squared difference between a random point and the sine wave.

The prior example is the pointwise version of the bias-variance decomposition. The mean version of the bias-variance decomposition looks at the mean squared bias and variance of a random variable z over a distribution D.

Here are some consequences of this:

- Underfitting is usually caused by too much bias
- Overfitting is usually caused by too much variance
- Training error reflects bias but not variance, whereas test error reflects both; this is why low training error can fool you when you've overfitted
- For many distributions, the variance goes to 0 as $n \to \infty$
- If h can fit g exactly, for many distributions the bias goes to 0 as $n \to \infty$
- If h cannot fit g well, 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; this means that you shouldn't add a feature unless it reduces bias more
- Can't reduce irreducible error (hence the name)
- Noise in the test set affects only $\text{Var}(\varepsilon)$, and noise in the training set affects only bias and $\text{Var}(h)$
- We can't precisely measure bias or variance of real-world data, because we cannot know g exactly and our noise model might be wrong
- We can, however, test learning algorithms by choosing g and making synthetic data

Example 12.6: Least-Squares Linear Regression

For simplicity, suppose we have no fictitious dimension; this means that our linear regression function passes through the origin.

In our model, we have $g(\vec{z}) = \vec{v}^T\vec{z}$, i.e. the ground truth is linear, so we can fit g perfectly with a linear h if not for the noise in the training set.

Let $\vec{e}$ be the noise vector, with $e_i \in \mathcal{N}(0, \sigma^2)$; this means our training labels are $\vec{y} = \mathbf{X}\vec{v} + \vec{e}$. Here, $\mathbf{X}$ and $\vec{y}$ are inputs to the linear regression, and we don't know $\vec{v}$ nor $\vec{e}$.

A linear regression computes the weights

$$\vec{w} = \mathbf{X}^\dagger\vec{y} = \mathbf{X}^\dagger(\mathbf{X}\vec{v} + \vec{e}) = \vec{v} + \mathbf{X}^\dagger\vec{e}.$$

We want $\vec{w} = \vec{v}$, but the noise in $\vec{y}$ becomes noise in $\vec{w}$; the $\mathbf{X}^\dagger\vec{e}$ term is the noise in the weights.

The bias is

$$\mathbb{E}[h(\vec{z})] - g(\vec{z}) = \mathbb{E}\left[\vec{w}^T\vec{z}\right] - \vec{v}^T\vec{z} = \mathbb{E}\left[\vec{z}^T\mathbf{X}^\dagger\vec{e}\right] = \vec{z}^T\,\mathbb{E}\left[\mathbf{X}^\dagger\right]\mathbb{E}[\vec{e}] = 0,$$

since $\vec{e}$ is independent from $\mathbf{X}^\dagger$, and $\mathbb{E}[\vec{e}] = 0$.

A warning: this does *not* mean that $h(\vec{z}) - g(\vec{z})$ is *always* 0; the difference is sometimes positive and sometimes negative, but the mean over the training set should be approximately 0. These deviations from the mean are captured in the variance.

When the bias is zero, a perfect fit is possible, but when a perfect fit is possible, not all learning methods give a bias of zero. Here, it's a benefit of squared error loss function; with a different loss function, we might have a nonzero bias even fitting a linear h to a linear g.

The variance is

$$\text{Var}(h(\vec{z})) = \text{Var}(\vec{w}^T\vec{z}) = \text{Var}(\vec{z}^T\vec{v} + \vec{z}^T\mathbf{X}^\dagger\vec{e}) = \text{Var}(\vec{z}^T\mathbf{X}^\dagger\vec{e}).$$

Note that this is the dot product of a vector $\vec{z}^T\mathbf{X}^\dagger$ with an isotropic, normally distributed vector $\vec{e}$. The dot product reduces it to a one-dimensional Gaussian along the direction $\vec{z}^T\mathbf{X}^\dagger$, so this variance is just the variance of the one-dimensional Gaussian times the squared length of the vector $\vec{z}^T\mathbf{X}^\dagger$:

$$\begin{aligned}\text{Var}(\vec{z}^T\mathbf{X}^\dagger\vec{e}) &= \sigma^2\left|\vec{z}^T\mathbf{X}^\dagger\right|^2 \\ &= \sigma^2\vec{z}^T\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{X}\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\vec{z} \\ &= \sigma^2\vec{z}^T\left(\mathbf{X}^T\mathbf{X}\right)^{-1}\vec{z}\end{aligned}$$

If we choose a coordinate system so that $\mathbb{E}[\mathbf{X}] = 0$, then $\mathbf{X}^T\mathbf{X} \to n\,\text{Cov}(D)$ as $n \to \infty$, so one can show that for $\vec{z} \in D$,

$$\text{Var}(h(\vec{z})) \approx \sigma^2\frac{d}{n},$$

where d is the dimension, i.e. the number of features per sample point.

Some takeaways here:

- The bias can be zero when hypothesis function can fit the true function (this is a nice property of the squared error loss function)
- The variance portion of the residual sum of squares (RSS) decreases as $\frac{1}{n}$, increases as d, or $O(d^p)$ if you use degree p polynomials

3/13/2022

Lecture 13

Ridge Regression, Subset Selection, LASSO

13.1 Ridge Regression

In ridge regression, we have a linear regression function, with a squared error loss function, and an ℓ_2-penalized cost function. Specifically, we have the optimization problem

$$\min_{\vec{w}} \left\|\mathbf{X}\vec{w} - \vec{y}\right\|^2 + \lambda\left\|\vec{w}'\right\|^2,$$

where $\vec{w}'$ is $\vec{w}$ with the component α replaced by 0 (i.e. we don't penalize the fictitious dimension in $\mathbf{X}$).

Notice that in ridge regression, we add a *regularization* term, i.e. a *penalty* term, for *shrinkage*. That is, we want to encourage a small $\left\|\vec{w}'\right\|$.

Why do we want to encourage a small $\left\|\vec{w}'\right\|$ with the regularization term? Firstly, because this guarantees positive definite normal equations; there is always a unique solution. Standard least-squares linear regression yields singular normal equations when the sample points lie on a common hyperplane in the feature space.

A comparison of least squares and ridge regression is shown in Fig. 13.1. The cost function for least squares is positive semidefinite, which has many minima, and the regression problem is said to be *ill-posed*. By adding a small penalty term, we obtain a positive definite quadratic form in ridge regression, which has one unique minimum.

A second reason why we want to regularize is because it reduces overfitting by reducing variance. Why is this the case? Suppose that $500x_1 - 500x_2$ is the best fit for well-separated points with $y_i \in [0,1]$. With this fit, a small change

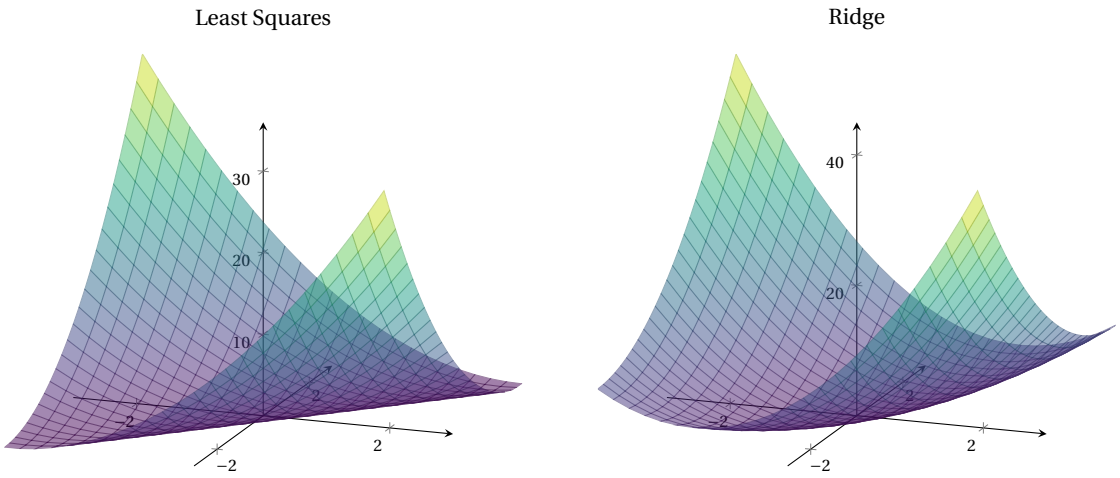


Figure 13.1: Least Squares Linear Regression vs. Ridge Regression Costs

in x corresponds to a big change in y; given that all the y values are small and the x values are not, this is a sure sign that we've overfit if tiny changes in x cause huge changes in y. As such, we penalize large weights such as these.

To actually compute the optimum for ridge regression, we can solve for $\nabla J = 0$; this gives the normal equations

$$\left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}'\right)\vec{\boldsymbol{w}} = \mathbf{X}^T\vec{\boldsymbol{y}},$$

where $\mathbf{I}'$ is the identity matrix with the bottom right entry set to zero; again, this is because we don't want to penalize the bias term α. Here, note that $\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}'$ is still always positive definite for $\lambda > 0$. We'd solve for $\vec{\boldsymbol{w}}$ and return $h(\vec{\boldsymbol{z}}) = \vec{\boldsymbol{w}}^T\vec{\boldsymbol{z}}$.

Observe that increase λ means that there is more regularization, causing a smaller $\left\|\vec{\boldsymbol{w}}'\right\|$.

Recall from the previous lecture that our typical data model is $\vec{\boldsymbol{y}} = \mathbf{X}\vec{\boldsymbol{v}} + \vec{\boldsymbol{e}}$, where $\vec{\boldsymbol{e}}$ is noise. The variance of ridge regression ins $\mathrm{Var}(\vec{\boldsymbol{z}}^T\left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}'\right)^{-1}\mathbf{X}^T\vec{\boldsymbol{e}})$. This means that as $\lambda \to \infty$, the variance tends to zero, but the bias increases; we'd tune for the hyperparameter λ by cross-validation.

Ideally, features should be normalized to have the same variance; an alternative is to use asymmetric penalty by replacing $\mathbf{I}'$ with some other diagonal matrix, giving us different penalties for different features.

13.1.1 Bayesian Justification for Ridge Regression

If we assign a prior probability on $\vec{\boldsymbol{w}}' \sim \mathcal{N}(0, \sigma^2)$, we can apply MLE to the posterior probability:

$$\begin{aligned} f(\vec{\boldsymbol{w}} \mid \mathbf{X}, \vec{\boldsymbol{y}}) &= \frac{f(\vec{\boldsymbol{y}} \mid \mathbf{X}, \vec{\boldsymbol{w}} \times f(\vec{\boldsymbol{w}}')}{f(\vec{\boldsymbol{y}} \mid \mathbf{X})} \\ &= \frac{L(\vec{\boldsymbol{w}})\, f(\vec{\boldsymbol{w}}')}{f(\vec{\boldsymbol{y}} \mid \mathbf{X})} \end{aligned}$$

If we maximize the log posterior, since the probability $f(\vec{\boldsymbol{y}} \mid \mathbf{X})$ is a constant with respect to $\vec{\boldsymbol{w}}$, we have

$$\begin{aligned} \max_{\vec{\boldsymbol{w}}} \ln f(\vec{\boldsymbol{w}} \mid \mathbf{X}, \vec{\boldsymbol{y}}) &= \max_{\vec{\boldsymbol{w}}} \left(\ln L(\vec{\boldsymbol{w}}) + \ln f(\vec{\boldsymbol{w}}') + C\right) \\ &= \max_{\vec{\boldsymbol{w}}} -C_1\left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2 - C_2\left\|\vec{\boldsymbol{w}}'\right\|^2 - C_3 \\ &= \min_{\vec{\boldsymbol{w}}} \left\|\mathbf{X}\vec{\boldsymbol{w}} - \vec{\boldsymbol{y}}\right\|^2 + \lambda\left\|\vec{\boldsymbol{w}}'\right\|^2 \end{aligned}$$

This method of using likelihood but maximizing the posterior is called *maximum a posterior* (MAP) estimation.

13.2 Feature Subset Selection

All features increase variance, but not all features reduce bias; as such, one idea is to identify poorly predictive features, and ignore them (i.e. give them a weight of 0). This means that we have less overfitting, which means that there will be a smaller test error. A second motivation is inference; simpler models usually convey an interpretable wisdom.

However, sometimes it's hard to determine which subset of features to use; different features can partly encode the same information, and it's combinatorially hard to choose the best feature subset.

A naive algorithm to choose the best subset is to try all $2^d - 1$ nonempty subsets of features, and choose the best classifier by cross-validation. Although this guarantees that we choose the best subset, it's extremely slow. If d is large, there is no algorithm that is guaranteed to find the best subset and runs in acceptable time, but heuristics often work well as a compromise.

Forward stepwise selection One heuristic is to start with the *null model* (with zero featuers), and repeatedly add the best feature until validation errors start increasing (due to overfitting) instead of decreasing. At each outer iteration, the inner loop tries every feature and chooses the best by validation. This requires training only $O(d^2)$ models instead of $O(2^d)$.

This heuristic isn't perfect though; for example, we won't find the best 2-feature model if neither of those features yields the best 1-feature model.

Backward stepwise selection Another heuristic is to start with all d features, and repeatedly remove the feature whose removal gives the best reduction in validation error. This also trains $O(d^2)$ models.

An additional heuristic to use is to only try to remove features with small weights. Recall that the variance of least squares regression is proportional to $\sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$. The *z-score* of a weight w_i is $z_i = \frac{w_i}{\sigma\sqrt{v_i}}$, where v_i is the ith diagonal entry of $(\mathbf{X}^T\mathbf{X})^{-1}$. A small z-score usually hints that the "true" w_i could be zero.

The forward stepwise heuristic is usually a better choice when you suspect that only a few features will be good predictors, e.g. spam. Backward stepwise is better when most features are important. If you're lucky, you'll stop early.

13.3 LASSO

LASSO regression has a linear regression function, squared error loss, and an ℓ_1-penalized mean loss; the name stands for "Least absolute shrinkage and selection operator".

LASSO is a regularized regression method similar to ridge regression, but it has the advantage that it often naturally sets some of the weights to zero. The optimization problem is formally

$$\min_{\vec{w}} \|\mathbf{X}\vec{w} - \vec{y}\|^2 + \lambda \|\vec{w}'\|_1, \text{ where } \|\vec{w}'\|_1 = \sum_{i=1}^{d} |w_i|.$$

Again, we don't want to penalize α.

Recall that in ridge regression, the isosurfaces of $\|\vec{w}'\|^2$ are hyperspheres; the isosurfaces of $\|\vec{w}'\|_1$ are *cross-polytopes*, as shown in Fig. 13.2. The unit cross-polytope is the convex hull of all the positive and negative unit coordinate vectors, and you get larger and smaller cross-polytope isosurfaces by scaling these.

In contrast with ℓ_2 regularization, the solution with ℓ_1 regularization usually lies at the tip of the polytope—this means that some dimensions are set to zero, giving us a sparser solution. In higher dimensions, more weights are set to zero, ex. in 3D, if the solution falls on a vertex, then 2 weights are set to zero; if the solution falls on an edge, then 1 weight is set to zero; and if the solution falls on a face, then no weights are set to zero.

Some algorithms for LASSO regression are subgradient descent, least-angle regression (LARS), and forward stage-wise.

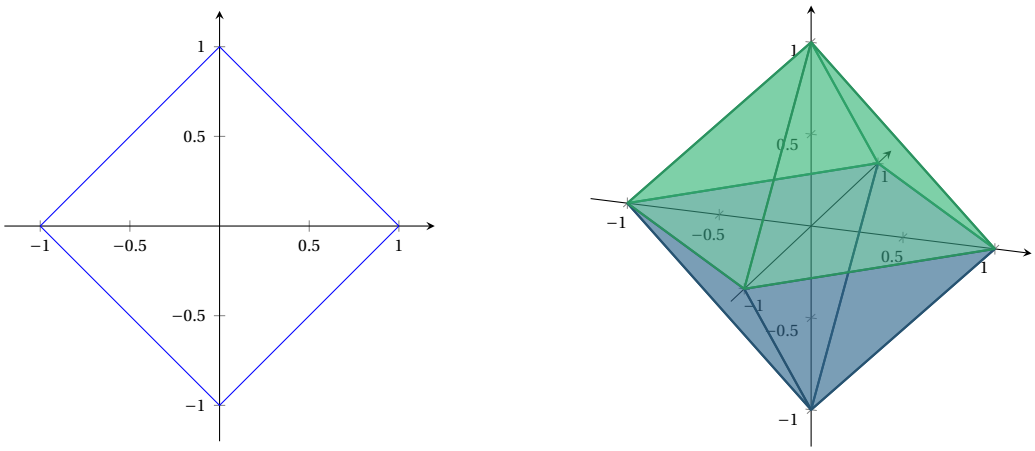


Figure 13.2: Isosurface $\|\vec{w}\|_1 = 1$ for $d = 2$ and $d = 3$

LASSO can also be formalized as a quadratic program, but it has 2^d constraints, since a d-dimensional polytope has 2^d faces. In practice, special-purpose optimization methods have been developed for LASSO.

3/9/2022

Lecture 14

Decision trees

Today we'll talk about *decision trees*, which is a nonlinear method for classification and regression.

With a decision tree, we have two different node types; internal nodes, which test feature values (usually just one value) and branch accordingly, and leaf nodes, which specify a class $h(x)$. An example is shown in Fig. 14.1.

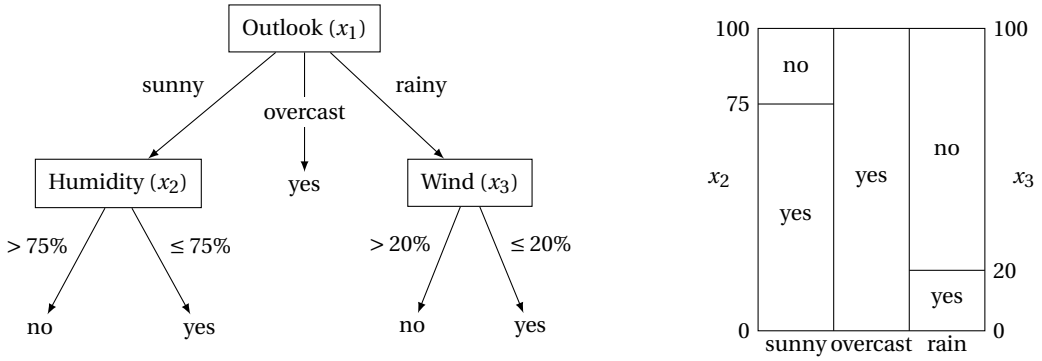


Figure 14.1: An example decision tree for when to go out for a picnic

With a decision tree, we're essentially cutting the x-space into rectangular cells, dividing up the space with an arbitrarily complicated decision boundary. This gives an interpretable result, and works well with both categorical and quantitative features.

A comparison between a linear classifier and a decision tree is shown in Fig. 14.2.

Let's consider how we would use a decision tree to classify data. We'd use a greedy, top-down learning heuristic; this algorithm is relatively straightforward, and has been rediscovered many times. It's naturally recursive, and we'll talk about how it works for classification first; later, we'll talk about regression.

Let $S \subseteq \{1, 2, \ldots, n\}$ be a set of sample points indices. The top-level call has $S = \{1, 2, \ldots, n\}$.

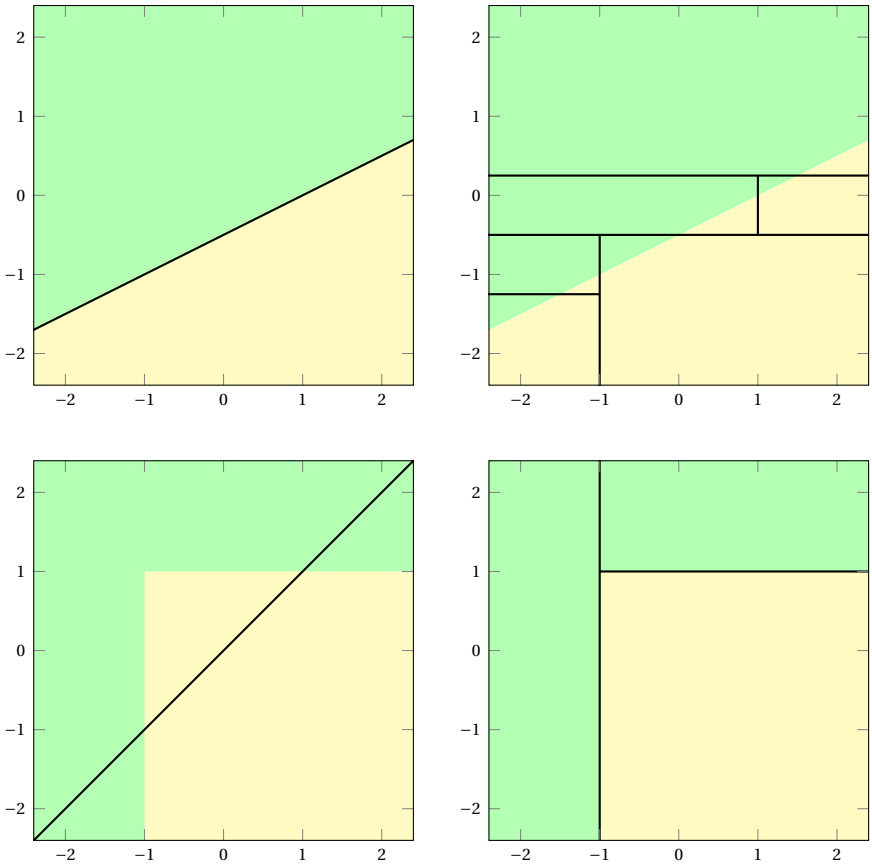


Figure 14.2: Comparison between linear classifiers (left) and decision trees (right) with two examples

```
1 GrowTree(S):
2     if (y_i = C for all i ∈ S and some class C) {
3         return new leaf(C)
4     } else {
5         choose best splitting feature j and splitting value β
6         S_l = {i ∈ S : X_ij < β}
7         S_r = {i ∈ S : X_ij ≥ β}
8         return new node(j, β, GrowTree(S_l), GrowTree(S_r))
9     }
```

Here, we say that the leaves are *pure*; they represent exactly one class.

How would we choose a best split (in Line 5)? One way is to just try all possible splits; for each set S, we let $J(S)$ be the *cost* of S. We then choose the split that minimizes $J(S_l) + J(S_r)$, or the split that minimizes the weighted average $\frac{|S_l|J(S_l)+|S_r|J(S_r)}{|S_l|+|S_r|}$.

This leads us to the next question: how would we choose the cost $J(S)$?

An initial (bad) idea is to label S with the class C that labels the most points in S. That is,

$$J(S) := \#\text{ of points in } S \text{ not in class } C.$$

Two different splits are shown in Fig. 14.3. While intuitively it may seem that a weighted average may be more accurate, it turns out that this is not actually always the case. Here, $J(S_l) + J(S_r) = 10$ for both splits, but the left split is much better. Even worse, the weighted average prefers the right split!

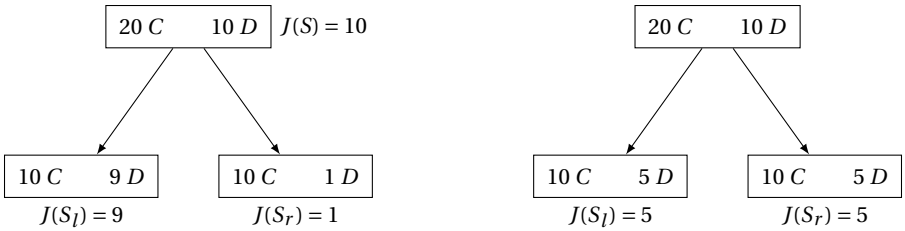


Figure 14.3: Two possible splits of a set S

The main issue here is that there are many splits that all have the same total cost. We want a cost function that better distinguishes between them.

A second idea is to use the *entropy*, an idea from information theory. In particular, we let Y be a random class variable, and suppose $\mathbb{P}(Y = C) = p_C$. We define the *surprise* of Y being class C as $-\log_2 p_C$.

Specifically, and event with probability 1 gives us zero surprise, but an event with probability 0 gives us infinite surprise. In information theory, the surprise is equal to the expected number of bits of information we need in order to transmit which events happened to a recipient who knows the probabilities of the events.

Definition 14.1: Entropy

The *entropy* of an index set S is the average surprise:

$$H(S) = -\sum_C p_C \log_2 p_C, \text{ where } p_C = \frac{\left|\{i \in S : y_i = C\}\right|}{|S|}.$$

That is, p_C is the proportion of points in S that are in class C.

Example 14.2

As a few examples,

- If all points in S belong to the same class, then
$$H(S) = -1 \log_2 1 = 0.$$
- If half of the points belong to class C and half belong to class D, then
$$H(S) = -\frac{1}{2}\log_2 \frac{1}{2} - \frac{1}{2}\log_2 \frac{1}{2} = 1.$$
- If all n points belong to different classes, then
$$H(S) = -\log_2 \frac{1}{n} = \log_2 n.$$

The entropy is the expected number of bits of information we need to transmit in order to identify the class of a sample point S chosen uniformly at random. As such, it makes sense that it only takes 1 bit to specify C or D when each class is equally likely, and it makes sense that it takes $\log_2 n$ bits to specify one of n classes when each class is equally likely. A plot of the binary entropy function $H(p_C)$ is shown in Fig. 14.4.

Using entropy, the weighted average entropy after a split is

$$H_{after} = \frac{|S_l| H(S_l) + |S_r| H(S_r)}{|S_l| + |S_r|}.$$

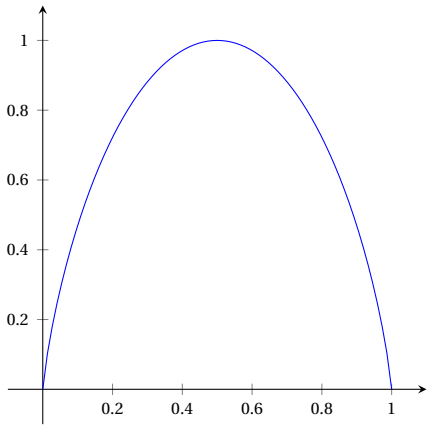


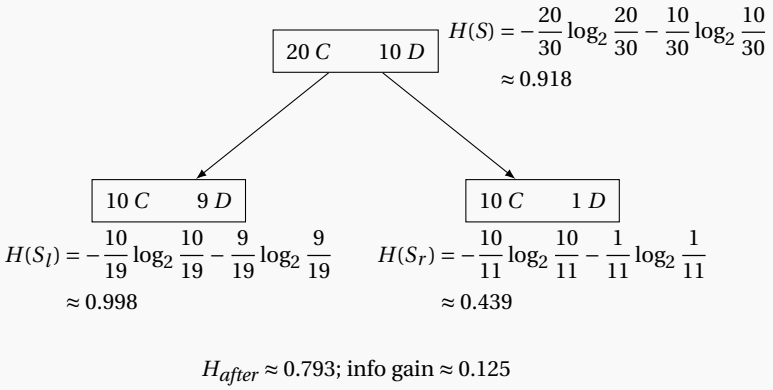
Figure 14.4: Plot of the binary entropy function $H(p_C) = -p_C \log_2 p_C - (1 - p_C) \log_2 p_C$

We'd then choose the split that maximizes *information gain* $H(S) - H_{after}$; this is the same as just minimizing H_{after}, as $H(S)$ is constant with respect to the split.

Notice that information gain is always positive *except* when one child is empty, or if for all C, $\mathbb{P}\big(y_i = C \mid i \in S_i\big) = \mathbb{P}\big(y_i = C \mid i \in S_r\big)$. The second case occurs in the second split in Fig. 14.3.

Example 14.3

Using the same first split as Fig. 14.3, we can calculate the entropy and information gain:



$H_{after} \approx 0.793$; info gain ≈ 0.125

If we compare the graph of the entropy curve and the graph of the earlier percent misclassified cost, we'd have (in the 2-class case) the plots in Fig. 14.5.

Since the entropy curve is strictly concave, the interior of any line segment is strictly below the curve. This means that the weighted average between the two parts of any split will always lie under the curve—the information gain is the difference between the weighted average entropy and the parent entropy (which lies directly above on the curve). As such, the information gain is always positive unless the two child sets both have exactly the same p_C and lie at the same point on the curve.

However, with the percent misclassified cost function, the curve is not strictly concave; the segment connecting two points may lie entirely on the curve—the split does not change the number of misclassified points, nor the percent of points misclassified. However, the biggest problem is that many different splits can get the same weighted average cost—this cost function does not distinguish the *quality* of different splits well.

There are many other different functions that work well as a cost; entropy isn't the only one. Many concave functions would work fine, including the polynomial $p(1 - p)$.

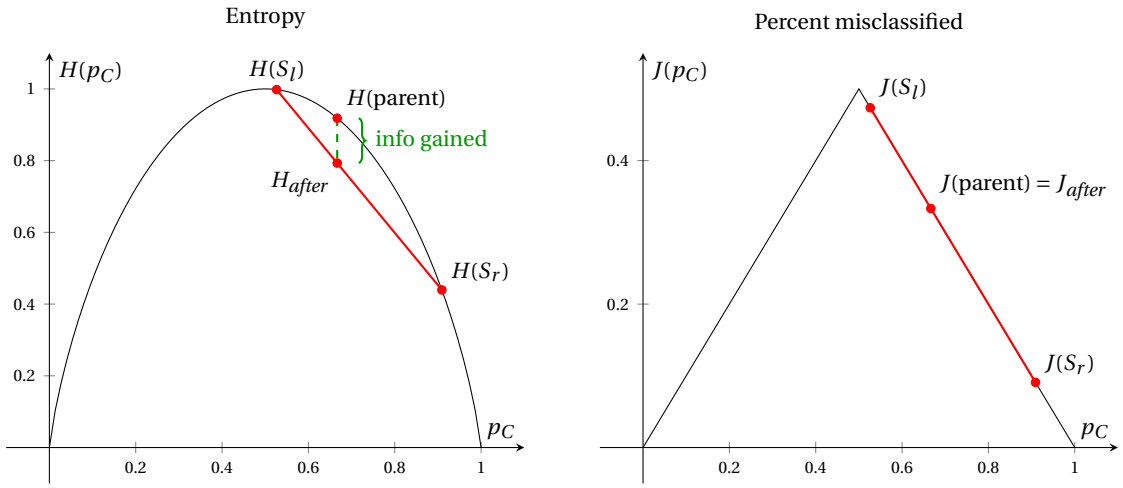


Figure 14.5: Comparison between the entropy function and the percent misclassified cost

Now let's talk a little bit more on choosing a split:

- If x_i is a binary feature, then the children are $x_i = 0$ and $x_i = 1$.
- If x_i has 3 or more discrete values, the split would depend on the application.
- If x_i is quantitative, then we can sort the values of x_i in S, and try to split between each pair of unequal consecutive values, taking the best split at the very end. (This is still efficient, because we can radix sort in linear time—in fact, if n is large, we should be using a radix sort.)

  One clever bit here is that we can update the entropy as we scan through the sorted list in $O(1)$ time (as shown in Fig. 14.6); this is important to obtain a fast tree-building time.

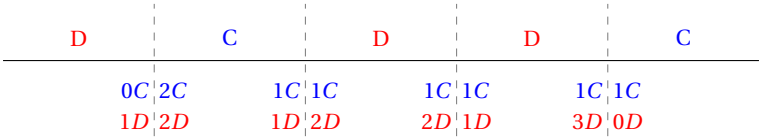


Figure 14.6: An example of scanning through a sorted list of sample points

To actually update the entropies, let C be the number of class C sample points to the left of a potential split and c be the number to the right of the split. Similarly, let D be the number of points not in class C to the left of the split and d be the number to the right of the split. The entropy of the left set is

$$-\frac{C}{C+D}\log_2\frac{C}{C+D}-\frac{D}{C+D}\log_2\frac{D}{C+D},$$

and the entropy of the right set is

$$-\frac{c}{c+d}\log_2\frac{c}{c+d}-\frac{d}{c+d}\log_2\frac{d}{c+d}.$$

(As a note, $\log_2 0$ is undefined, but this formula would still work if we use the convention that $0\log_2 0 = 0$.)

It then follows that the weighted average is

$$-\frac{1}{n'}\left(C\log_2\frac{C}{C+D}+D\log_2\frac{D}{C+D}+c\log_2\frac{c}{c+d}+d\log_2\frac{d}{c+d}\right),$$

where $n' = C+D+c+d$ is the total number of sample points in the tree node. We'd choose the split that minimizes the weighted average.

Let's look at some algorithms and running times. To classify a test point, we'd walk down the tree until we get a leaf, and return its label. The worst case time is $O(\text{depth})$; for binary features, this is always $\leq d$, and usually (not always) it's $O(\log n)$.

To train, we try $O(d)$ splits at each node for binary features, and $O(n'd)$ splits at each node for quantitative features. Specifically, with binary features, all we need to do is try all different features ($O(d)$) and a constant number of splits per feature ($O(1)$), and for quantitative features, we try all features ($O(d)$), and try all $O(n')$ splits for each feature (i.e. scan through the sample points linearly after radix sorting).

In either case, it takes $O(n'd)$ time to find and perform the split (with binary features, we still need to spend $O(n')$ time to split the sample points); the speed of splitting quantitative features comes from the clever way of computing entropy for each split, as described before.

Further, each point participates in $O(\text{depth})$ nodes, and it costs $O(d)$ time in each node; this means that the running time is $O(nd \text{ depth})$.

This is actually a quite reasonable running time; the size of the design matrix $\mathbf{X}$ is nd, and depth is usually logarithmic.

3/14/2022

Lecture 15

Decision Tree Variations, Ensemble Learning

15.1 Decision Tree Variations

Last time, we talked about the vanilla algorithms for building decision trees, and using them to classify test points. There are a lot of different variations on the basic algorithm, which we'll discuss today.

15.1.1 Multivariate Splits

One variation on decision trees is to utilize multivariate splits—that is, we find splits that are not axis-aligned through other classification algorithms, or by generating them randomly.

An example is shown in Fig. 15.1; it's quite clear that an ordinary decision tree needs many splits to approximate a diagonal linear decision boundary, but a single multivariate split does the job much better.

We can use other classification algorithms like SVMs, logistic regression, Gaussian discriminant analysis, etc. to create the multivariate split; decision trees permit these algorithms to find more complicated decision boundaries by making them hierarchical.

As such, we may get a better classifier at the cost of worse interpretability or speed. Specifically, standard decision trees are very fast, since we're only checking one feature at each tree node. If there were instead hundreds of features, and we need to check all of them at every single level of the tree, this slows down the classification a lot.

This means that sometimes it's better to consider methods like forward stepwise selection during training so that we'd only need to check a few features at each node during classification. Along similar lines, we can also utilize LASSO to get a sparser result involving fewer features.

15.1.2 Decision Tree Regression

We can also use a decision tree for regression, creating a piecewise constant regression function; an example is shown in Fig. 15.2.

Here, the cost is

$$J(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \mu_S)^2,$$

where μ_S is the mean label y_i for sample points $i \in S$.

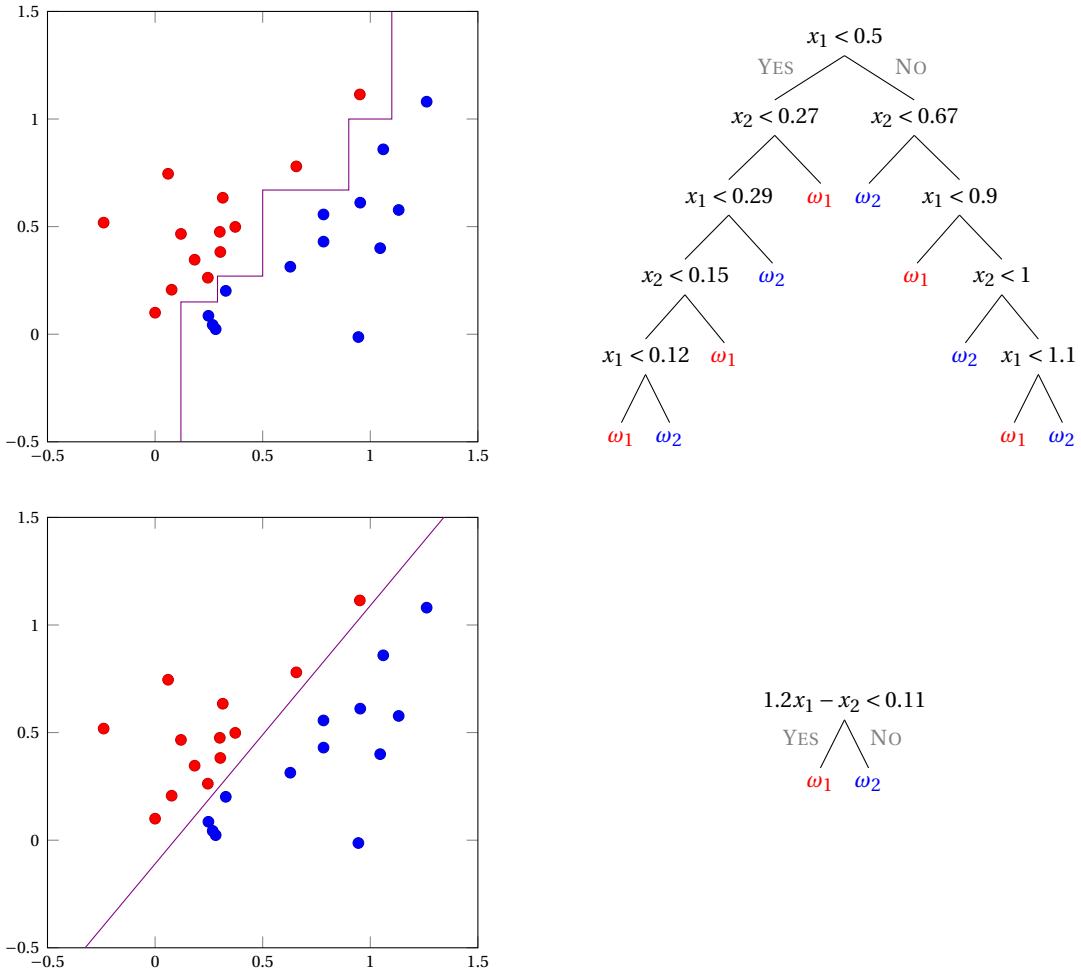


Figure 15.1: Comparison between a vanilla decision tree and a decision tree with a multivariate split

This means that if all points in a node have the same y-value, then the cost is zero for that node; we want to choose the split that minimizes the weighted average of the costs of the children after the split.

15.1.3 Stopping Early

The basic version of the decision tree will keep subdividing nodes until every single leaf is pure (i.e. contains only one class). We don't *always* have to do that; sometimes it's better to stop subdividing nodes earlier.

Why is this the case?

- Limits tree depth (for speed)
- Limits tree size (for big datasets)
- Complete tree is more prone to overfitting (as we're trying to correctly classify every single training point)
- With noise or overlapping distributions, the purity of the leaves is also counterproductive; it's usually better to estimate posterior probabilities

Specifically, with overlapping class distributions, refining the tree down to one sample point per leaf is guaranteed to overfit; as such, it's better to stop early, and then classify each leaf node by taking a vote of its sample points. Alternatively, we can use the points to estimate a posterior probability for each leaf, and then just return the probability. If there are many points in each leaf, the posterior probabilities may be reasonably accurate.

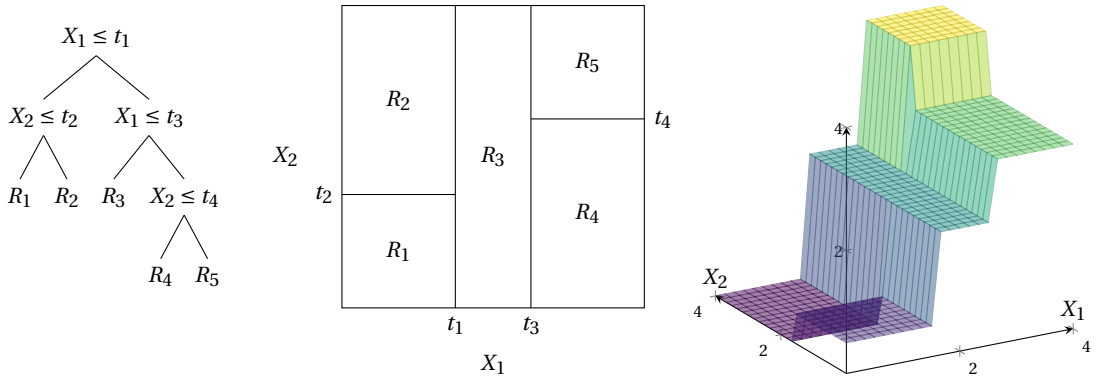


Figure 15.2: Example of regression using a decision tree

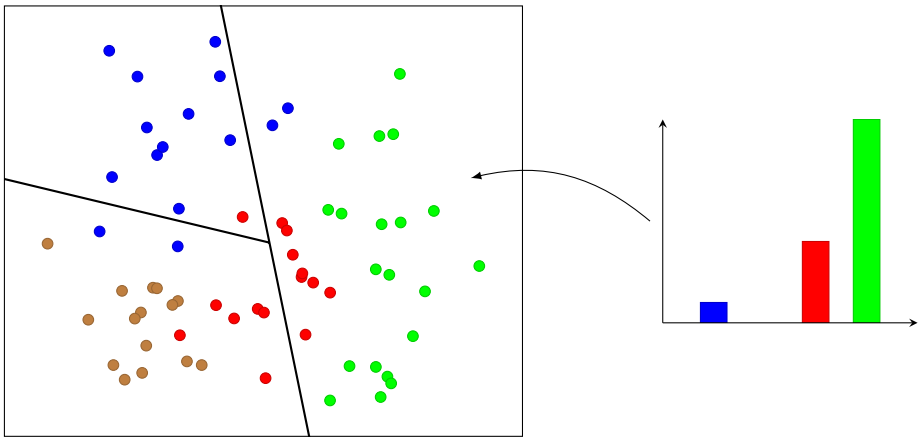


Figure 15.3: Example of a decision tree that has been stopped early, with the corresponding posterior probability distribution for one leaf

When would we stop the decision tree? Here are some possibilities:

- When the next split doesn't reduce the entropy/error enough (this is dangerous though; pruning is better)
- When most of the node's points (ex. $> 95\%$) have the same class (to deal with outliers)
- When the node contains few sample points (ex. < 10)
- When the cell's edges are all tiny
- When the depth is too great (this is risky if there are still many points in the cell)
- We can also use validation to compare errors

The last choice is the slowest but also the most effective way to know when to stop, i.e. to use validation to determine whether splitting the node would lower the validation error. However, if the goal is to avoid overfitting, it's generally even more effective to grow the tree a little too large, and then use validation to prune it back—this is what we'll talk about next.

Another point to talk about is what the leaves would return. One choice is to give a majority vote, and another is to simply return the class posterior probabilities; these would be useful in classification. Another choice is to return an average class, in the case of regression.

15.1.4 Pruning

As mentioned prior, one variation of a decision tree is to grow the tree larger than we want, and then greedily remove each split whose removal improves the validation performance. This is usually more reliable than stopping early.

Here, we have to do validation once for each split that we're considering removing, but we can do this pretty cheaply. Here, we keep track of which points in the validation set end up at which leaf; when deciding whether to remove a split, we can just look at the validation points in the two leaves and see how they'll be reclassified, and how it'll change the error rate—this can be done very quickly.

The reason why pruning usually works better than stopping early is because a split that doesn't seem to make much progress is followed by a split that makes a lot of progress; stopping early means that we'll never find out whether a given split will be followed by one that makes more progress. Pruning is a simple idea, but it's recommended if there's enough time to build and prune the tree.

15.2 Ensemble Learning

Decision trees are fast, simple, interpretable, and easy to explain; they're invariant under scaling and translation, and are also robust to irrelevant features. However, decision trees aren't the best at predicting, compared to the previous methods we've seen, since they have high variance.

For example, if we take a training set, split it into two halves, and train two decision trees on each half, it's not uncommon for the two trees to turn out very differently. Specifically, if the first splits are different, the rest of the trees will likely be completely different.

To fix this, let's turn to an analogy—suppose we're generating random numbers from some unknown distribution. Just generating one number would give us a high variance, but if we generate n random numbers and take their average, the variance of the average is n times smaller. We can apply this same idea to decision trees; we can reduce the variance of decision trees by taking an average answer of a bunch of decision trees.

Insert anecdote about oxen and cows and taking averages of guesses for weight to get very very close to the true weight.

The main idea is that sometimes the average opinion of a bunch of idiots is better than the opinion of one expert; the same applies with learning algorithms. We call a learning algorithm a *weak learner* if it does better than guessing randomly; we can combine a bunch of weak learners to get a strong one.

We can apply this idea in a few ways:

- Average output of several different learning algorithms
- Average output of the same learning algorithm on many training sets (if we have a lot of data)
- *Bagging*: Average output of the same learning algorithm on many random subsamples of one training set
- *Random forests*: Average output of randomized decision trees on random subsamples

The last two methods are the most common ways to use averaging, as usually we don't have enough training data to use fresh data for every learner.

Note that in regression algorithms, we'd take the median or mean output, and in classification algorithms, we'd take the majority vote or an average posterior probability.

The idea here is that we want to use learners with low bias (ex. deep decision trees). High variance and some overfitting are okay here, because averaging will reduce the variance anyways. Specifically, each learner may overfit, but each will overfit in its own unique way, and the averaging will reduce this variance.

Averaging sometimes reduces the bias and increases flexibility, for example creating a nonlinear decision boundary from linear classifiers. But more importantly, averaging learners reduces their variance a lot more than it does bias, so it's better to get the bias nice and small before averaging. The hyperparameter settings will also usually be different from just one learner.

15.2.1 Bagging

Bagging is a randomized method for creating many different learners from the same dataset. It works well with many different learning algorithms, though one exception is k-nearest neighbors—bagging mildly degrades it.

Given an n point training sample, we generate a random subsample of size n' by sampling *with replacement*; that is, some points are chosen multiple times, and some points are not chosen at all. An example is shown in Fig. 15.4. If

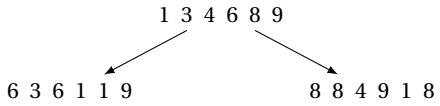


Figure 15.4: Example of drawing two subsamples from a dataset.

we choose subsamples of size $n' = n$, on average $\approx 63.2\%$ of the data points are chosen; this fraction varies randomly.

To account for the duplicates, we treat it as if a point chosen j times have a greater weight:

- In decision trees, a point chosen j times has $j\times$ weight in the entropy calculation
- In SVMs, a point chosen j times has $j\times$ penalty in violating the margin
- In regressions, a point chosen j times incurs $j\times$ loss

We repeat this process of choosing subsamples to train learners until we have T learners. We then have a *metalearner* that takes in a test point, feeds it into all T learners, and returns the average or majority output (depending on if we want classification or regression).

15.2.2 Random Forests

Sometimes, random sampling isn't random enough; for example, with bagging, decision trees often look very similar—why is this? Oftentimes, the existence of a really strong predictor means that this same feature is split at the top of every single tree, no matter what random sample we choose.

If the decision trees are very similar, then taking the average of many similar decision trees doesn't reduce the variance much.

A key idea here is that at each tree node, we take a random sample of m features (out of d), and we choose the best split from these m chosen features; we aren't allowed to split on the other $d-m$ features. We'd choose these random features independently for each tree node.

For classification, $m \approx \sqrt{d}$ works well, and for regression, $m \approx \frac{d}{3}$ works well. These values of m are good starting guesses, but they'd need to be tuned for a particular application, as the value of m is a hyperparameter. A smaller m gives more randomness, less tree correlation, and more bias.

One reason why this works is because if there's a really strong predictor, only a fraction of the trees can choose that predictor as the first split (namely, $\frac{m}{d}$ of them). This tends to "decorrelate" the trees, and taking the average will give less variance than a single tree.

However, we have to be careful not to dumb down the trees *too* much to decorrelate; averaging works best when we have very strong learners that are also diverse, though it is very hard to do so.

The disadvantages of random forests are that it's slow, and loses interpretability/inference; the compensation is that it's more accurate than a single decision tree.

3/28/2022

Lecture 16

The Kernel Trick

16.1 Kernels

Today we'll talk about kernels, and more specifically the kernel trick. It is very similar to features, and we'll get our motivation from polynomials.

Recall that with d input features and degree p polynomials, we have $O(d^p)$ features, which blows up; it gets very slow very fast.

Today, we'll find a way to use these exponentially many polynomial features without computing them!

The key observation here is that in many learning algorithms,

- The weights can be written as a linear combination of the sample points
- We can use inner products of the featured vectors $\Phi(\vec{\boldsymbol{x}})$ only—we do not need to compute the value of each $\Phi(\vec{\boldsymbol{x}})$ individually at all

Going back to the first point, mathematically we have

$$\vec{\boldsymbol{w}} = \mathbf{X}^T \vec{\boldsymbol{a}} = \sum_{i=1}^{n} a_i \mathbf{X}_i,$$

for some $\vec{\boldsymbol{a}} \in \mathbb{R}^n$.

For an algorithm where this is true, we can substitute this identity $\vec{\boldsymbol{w}} = \mathbf{X}^T \vec{\boldsymbol{a}}$ into the algorithm. Here, we'd get rid of the original set of weights $\vec{\boldsymbol{w}}$ and instead optimize the *n dual weights* $\vec{\boldsymbol{a}}$ (also called *dual parameters*) instead of the $d+1$ (or with polynomial features, d^p) *primal weights* $\vec{\boldsymbol{w}}$.

16.1.1 Kernel Ridge Regression

It turns out that it is impossible to write the weights as a linear combination of the sample points if we have a bias term; we'd need to penalize the bias term as well. To work around this, we center $\mathbf{X}$ and $\vec{\boldsymbol{y}}$ so that their means are zero (making sure not to center the fictitious dimension):

$$\begin{aligned} \mathbf{X}_i &\leftarrow \mathbf{X}_i - \vec{\boldsymbol{\mu}}_X \\ y_i &\leftarrow y_i + \mu_y \\ X_{i,d+1} &= 1 \end{aligned}$$

Here, the expected value of the bias is zero—the actual bias won't usually be exactly zero, but it'll usually be small enough that we don't do much harm by penalizing the bias.

This means that the normal equations become

$$\left(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I}\right)\vec{\boldsymbol{w}} = \mathbf{X}^T \vec{\boldsymbol{y}}.$$

Let's look at a slightly different system; suppose $\vec{\boldsymbol{a}}$ is a solution to

$$\left(\mathbf{X}\mathbf{X}^T + \lambda \mathbf{I}\right)\vec{\boldsymbol{a}} = \vec{\boldsymbol{y}}.$$

This means that

$$\begin{aligned} \mathbf{X}^T \vec{\boldsymbol{y}} &= \mathbf{X}^T\mathbf{X}\mathbf{X}^T \vec{\boldsymbol{a}} + \lambda \mathbf{X}^T \vec{\boldsymbol{a}} \\ &= \left(\mathbf{X}^T\mathbf{X} + \lambda \mathbf{I}\right)\mathbf{X}^T \vec{\boldsymbol{a}} \end{aligned}$$

Notice that this means that if we set $\vec{\boldsymbol{w}} = \mathbf{X}^T \vec{\boldsymbol{a}}$, then we have a solution to the normal equations. Further, notice that $\vec{\boldsymbol{w}}$ is now a linear combination of the sample points.

We call $\vec{a}$ a *dual solution*; it solves the *dual form* of the ridge regression

$$\min_{\vec{a}} \left\|\mathbf{X}\mathbf{X}^T\vec{a} - \vec{y}\right\|^2 + \lambda\left\|\mathbf{X}^T\vec{a}\right\|^2.$$

This is obtained by substituting $\vec{w} = \mathbf{X}^T\vec{a}$ into the original ridge regression cost function (which was $\left\|\mathbf{X}\vec{w} - \vec{y}\right\|^2 + \lambda\|\vec{w}\|^2$).

This then leads us to the algorithm; in the training phase, we solve $\left(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}\right)\vec{a} = \vec{y}$ for $\vec{a}$ to get the dual solution.

In the testing phase, we have our regression function

$$h(\vec{z}) = \vec{w}^T\vec{z} = \vec{a}^T\mathbf{X}\vec{z} = \sum_{i=1}^{n} a_i\left(\mathbf{X}_i^T\vec{z}\right).$$

Notice that this is a weighted sum of inner products; this will become important later, as it'll be where we make a lot of the computation disappear (with $\Phi(\vec{x})$ and $\Phi(\vec{z})$).

Let us define $k(\vec{x},\vec{z})$ be the *kernel function*. Here, we have $k(\vec{x},\vec{z}) = \vec{x}^T\vec{z}$; later, we'll replace $\vec{x}$ and $\vec{z}$ with $\Phi(\vec{x})$ and $\Phi(\vec{z})$, and this is where the magic will happen.

We will also define $\mathbf{K} = \mathbf{X}\mathbf{X}^T$ to be the $n \times n$ *kernel matrix*; node that $K_{ij} = k(\mathbf{X}_i, \mathbf{X}_j)$.

$\mathbf{K}$ is singular if $n > d+1$ (and sometimes still singular if this doesn't hold); this is almost always true with real datasets. In this case, there is probably no solution if $\lambda = 0$; this means that we need to choose a positive λ, but that's okay.

With these new terms, we can rewrite the dual ridge regression algorithm:

```
1 Compute the kernel matrix one entry at a time: K_ij ← k(X_i, X_j)
2 Solve (K + λI)a = y for a
3 for each test point z:
4     h(z) = Σ_{i=1}^n a_i k(X_i, z)
```

Looking at the runtimes, Line 1 takes $O(n^2d)$ time, Line 2 takes $O(n^3)$ time, and Line 4 takes $O(nd)$ time.

Notice that this never uses $\mathbf{X}_i$ directly, while still giving us the same predictions as the primal regression.

Comparing the dual and primal running times, we have

- Dual solves $n \times n$ linear system, in $O(n^3 + n^2d)$ time
- Primal solves $d \times d$ linear system, in $O(d^3 + d^2n)$ time

It should be clear here that we'd prefer the dual when $d > n$.

This isn't always true though; the number of features is usually fewer than the number of sample points. However, one big advantage is that when we use polynomial terms as new features, the d in the primal algorithm will increase with the new features we add, but the d in the dual algorithm will *not* increase.

16.1.2 The Kernel Trick (Kernelization)

We'll now talk about the kernel trick; we'll see that we can compute a polynomial kernel that involves many monomial terms without actually computing those terms.

Definition 16.1: Polynomial Kernel

The *polynomial kernel* of degree p is $k(\vec{x},\vec{z}) = (\vec{x}^T\vec{z} + 1)^p$.

Theorem 16.2

$(\vec{x}^T\vec{z}+1)^p = \Phi(\vec{x})^T\Phi(\vec{z})$, where $\Phi(\vec{x})$ is a lifting function that contains every monomial term with respect to $\vec{x}$ of degree $0,\dots,p$.

Example 16.3

We won't prove the above theorem, but here is an example for $d=2$, $p=2$:

$$\begin{aligned}(\vec{x}^T\vec{z}+1)^2 &= x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_1 + 2x_1z_1 + 2x_2z_2 + 1\\ &= \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1\end{bmatrix}\begin{bmatrix} z_1^2\\ z_2^2\\ \sqrt{2}z_1z_2\\ \sqrt{2}z_1\\ \sqrt{2}z_2\\ 1\end{bmatrix}\\ &= \Phi(\vec{x})^T\Phi(\vec{z})\end{aligned}$$

Here, these vectors are how we're defining Φ.

However, notice that we have a few constant stuck in here; we can't control these constants, but it's okay. When we optimize with this kernel, the primal weights will automatically adjust themselves to compensate for these constants.

The key benefit here is that we can compute $\Phi(\vec{x})^T\Phi(\vec{z})$ in $O(d)$ time (where d is the *original* number of features) instead of $O(d^p)$ time, even though $\Phi(\vec{x})$ has length $O(d^p)$.

This leads us to the kernel ridge regression; we replace $\mathbf{X}_i$ with $\Phi(\mathbf{X}_i)$. Our kernel function is then

$$k(\vec{x},\vec{z}) = \Phi(\vec{x})^T\Phi(\vec{z}) = (\vec{x}^T\vec{z}+1)^p.$$

Notice that we define it with $\Phi(\vec{x})$ and $\Phi(\vec{z})$, but we never compute these values; we use the faster $(\vec{x}^T\vec{z}+1)^p$ instead.

With this, we can now do polynomial regression with an exponentially long higher-order polynomial in less time than it would take even to write out the final polynomial. The running time is *sublinear* in the length of the Φ vectors.

16.1.3 Kernel Perceptrons

Revisiting the original perceptron algorithm, we'll now consider the case with features:

```
Set starting point w ← y_1 Φ(X_1) // don't really matter; just can't be 0
while some y_i Φ(X_i) · w < 0: // while there is a misclassified point
    w ← w + ε y_i Φ(X_i)

for each test point z:
    h(z) ← w · Φ(z)
```

To kernelize, let $\Phi(\mathbf{X})$ be the $n \times D$ matrix with rows $\Phi(\mathbf{X}_i)^T$, where D is the length of $\Phi(\cdot)$. The kernel matrix is now $\mathbf{K} = \Phi(\mathbf{X})\Phi(\mathbf{X})^T$.

(An intuition of why we can kernelize in the first place is because the weight updates are all adding some constant multiple of a sample point; this means that the final weight vector will be some weighted combination of the sample points.)

We can dualize with $\vec{w} = \Phi(\mathbf{X})^T\vec{a}$. It's a little hard to conceptualize, but as an example, the code "$a_i \leftarrow a_i + \varepsilon y_i$" has the same effect as "$\vec{w} \leftarrow \vec{w} + \varepsilon y_i \Phi(\mathbf{X}_i)$".

Further, substituting $\vec{\boldsymbol{w}} = \Phi(\mathbf{X})^T \vec{\boldsymbol{a}}$, we have

$$\Phi(\mathbf{X}_i) \cdot \vec{\boldsymbol{w}} = (\Phi(\mathbf{X})\vec{\boldsymbol{w}})_i = \left(\Phi(\mathbf{X})\Phi(\mathbf{X})^T \vec{\boldsymbol{a}}\right)_i = (\mathbf{K}\vec{\boldsymbol{a}})_i.$$

Making all of these substitutions, we have our final dual perceptron algorithm:

```
1 a⃗ ← [y_1  0  ⋯  0]^T
2 Compute the kernel matrix K_ij ← k(X_i, X_j)
3 while some y_i (Ka⃗)_i < 0:
4     a_i ← a_i + ε y_i
5
6 for each test point z⃗
7     h(z⃗) ← Σ_{j=1}^n a_j k(X_j, z⃗)
```

Looking at the runtimes, Line 2 takes $O(n^2 d)$ because of the kernel trick, Line 4 takes $O(1)$ time (but updating $\mathbf{K}\vec{\boldsymbol{a}}$ takes $O(n)$ time to reflect the changes), and Line 7 takes $O(nd)$ time again because of the kernel trick.

16.1.4 Kernel Logistic Regression

The stochastic gradient descent step in kernelized logistic regression becomes

$$a_i \leftarrow a_i + \varepsilon\left(y_i - s((\mathbf{K}\vec{\boldsymbol{a}})_i)\right).$$

This can be done in $O(n)$ time if we update $\mathbf{K}\vec{\boldsymbol{a}}$ each time we update a dual weight a_i.

Or, with batch gradient descent, the step becomes

$$\vec{\boldsymbol{a}} \leftarrow \vec{\boldsymbol{a}} + \varepsilon\left(\vec{\boldsymbol{y}} - s(\mathbf{K}\vec{\boldsymbol{a}})\right).$$

With predictions, for each test point $\vec{\boldsymbol{z}}$, we compute

$$h(\vec{\boldsymbol{z}}) \leftarrow s\left(\sum_{j=1}^{n} a_j k(\mathbf{X}_j \vec{\boldsymbol{z}})\right).$$

If we're just using logistic regression as a classifier, and we don't care about the posterior probabilities, then we don't need the logistic function at the end here, and just compute the summation, much like in the perceptron algorithm—the logistic function is just there to give us posterior probabilities.

16.1.5 The Gaussian Kernel

We've talked about using a polynomial kernel to perform fast computations in a space with exponentially large dimensions—here, we'll discuss how to use feature vectors in an infinite-dimensional space.

Definition 16.4: Gaussian Kernel

The *Gaussian kernel*, or the *radial basis function kernel* is based on the observation that there exists a $\Phi(\vec{\boldsymbol{x}})$ (albeit of infinite dimension) such that

$$k(\vec{\boldsymbol{x}}, \vec{\boldsymbol{z}}) = \exp\left(-\frac{\|\vec{\boldsymbol{x}} - \vec{\boldsymbol{z}}\|^2}{2\sigma^2}\right) = \Phi(\vec{\boldsymbol{x}})^T \Phi(\vec{\boldsymbol{z}}).$$

Example 16.5

For example, with $d = 1$, we have the infinite vector

$$\Phi(\vec{\boldsymbol{x}}) = \exp\left(-\frac{x^2}{2\sigma^2}\right)\left[1 \quad \frac{x}{\sigma\sqrt{1!}} \quad \frac{x^2}{\sigma^2\sqrt{2!}} \quad \frac{x^3}{\sigma^3\sqrt{3!}} \quad \cdots\right]^T.$$

It's not too useful to think about the value of $\Phi(\vec{x})$ directly, since we're now working in an infinite dimensional space. Nobody really cares about this value, and we just use the kernel function k.

As such, at this point it's not too useful to think about the points in a high dimensional space; instead, think of k as a measure of how similar or close together two points are to each other.

The key observation here is that the hypothesis $h(\vec{z}) = \sum_{j=1}^{n} a_i k(\mathbf{X}_j, \vec{z})$ is a linear combination of Gaussians centered at the sample points. The dual weights are the coefficients of the linear combination, and the Gaussians are a basis for the hypothesis.

The Gaussian kernel is very popular in practice; why is this the case?

- It gives a very smooth regression function h; it's infinitely differentiable, or C^∞ continuous.
- The regression function behaves somewhat like the k-nearest neighbors algorithm, but smoother.
- It oscillates less than polynomials (depending on σ)
- $k(\vec{x}, \vec{z})$ can be interpreted as a similarity measure; it has a maximum when $\vec{z} = \vec{x}$, and goes to 0 as the distance increases.
- The sample points "vote" for a value at $\vec{z}$, but closer points get more weight to their vote.

We'd choose σ as a hyperparameter through (cross-)validation; it trades off bias vs. variance. A larger σ makes the Gaussians wider, creating a smoother h; this tends to give more bias (as the function can't adapt as well to the data) but less variance.

3/30/2022

Lecture 17

Neural Networks

Today we'll be talking about neural networks, which can do both classification and regression. They tie together a few different ideas in the course: perceptrons, logistic regression, ensemble learning, and stochastic gradient descent. They also tie in the idea of lifting sample points to a higher-dimensional feature space, except crucially, neural networks can learn the features themselves.

17.1 The XOR Problem

As some motivation, let us go back to the idea of a perceptron. One of the things that a perceptron can't solve is the XOR problem. That is, we want a linear classifier that separates the 1's from the 0's from the output of an XOR function:

		x_1	
XOR		0	1
x_2	0	0	1
	1	1	0

It's impossible to find a linear classifier that separates the 1's and 0's in just 2 dimensions, and as a result, there was very little study and research done on the perceptron. However, a very simple way to get around this problem is to just add a new quadratic feature $x_1 x_2$, and XOR becomes linearly separable in 3D, as shown in Fig. 17.1.

However, there's an even more powerful way to do XOR; the idea is to design linear classifiers whose output is the input to other linear classifiers, as shown in Fig. 17.2. This way, we should be able to emulate arbitrary logical circuits.

However, a linear combination of a linear combination is still a linear combination; this still only works for linearly separable points. The last step to making neural networks is adding some kind of nonlinearity between the linear combinations.

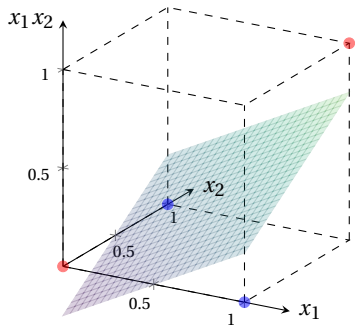


Figure 17.1: XOR lifted into 3D with a quadratic feature

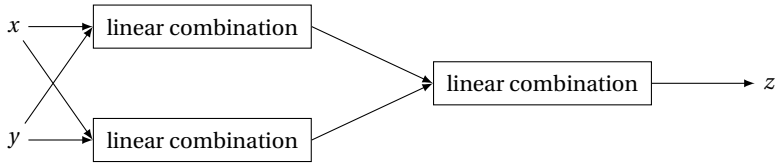


Figure 17.2: An example of nested linear classifiers

Suppose we call these boxes that compute linear combinations *neurons*. If a neuron runs the linear combination through some nonlinear function before sending it to other neurons, then the neurons can act like logic gates. The nonlinearity can be as simple as clamping the output such that it doesn't go below zero, and this is what people usually use in practice nowadays as well.

The traditional choice, however, has been to use the logistic function. The logistic function is nice because it can't go below 0 or above 1, so it can't get huge and oversaturate other neurons; it's also smooth, so it has well-defined gradients and Hessians that we can use for optimization.

Now with the logistic functions between the linear combinations, Fig. 17.3 shows a two-level perceptron that computes the XOR function.

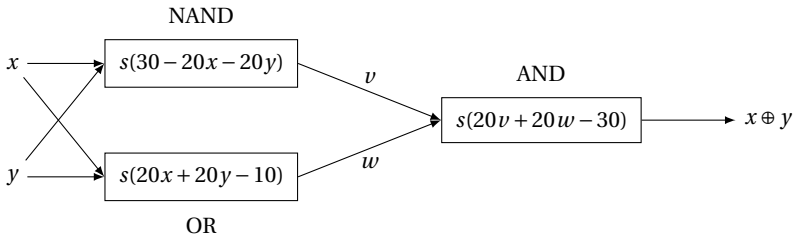


Figure 17.3: A two-level perceptron that computes the XOR function

A natural question that follows is: can an algorithm learn a function like this?

17.2 Neural Network With 1 Hidden Layer

A typical network with one hidden layer is structured as follows:

- Input layer: $x_1, \ldots, x_d$, with $x_{d+1} = 1$
- Hidden units: $h_1, \ldots, h_m$, with $h_{m+1} = 1$
- Output layer: $z_1, \ldots, z_k$

Between each layer, we introduce weights:

- Weights between input and hidden layer: $m \times (d+1)$ matrix $\mathbf{V}$, with $\mathbf{V}_i^T$ denoting row i
- Weights between hidden and output layer: $k \times (m+1)$ matrix $\mathbf{W}$, with $\mathbf{W}_i^T$ denoting row i

Recall the logistic function $s(\gamma) = \frac{1}{1+e^{-\gamma}}$; we use this as an *activation function*, though many other nonlinear functions can be used.

Further, recall that we define $s(\vec{v})$ on a vector $\vec{v}$ as $s(\vec{v}) = \begin{bmatrix} s(v_1) & s(v_2) & \cdots \end{bmatrix}^T$, and let us also define $s_1(\vec{v}) = \begin{bmatrix} s(v_1) & s(v_2) & \cdots & 1 \end{bmatrix}$, with a one appended to the end.

A diagram of a neural network with one hidden layer is shown in Fig. 17.4.

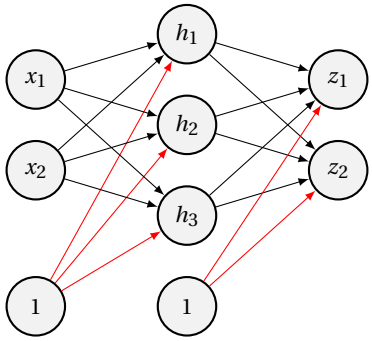


Figure 17.4: An example fully-connected neural network with one hidden layer.

Here, we have

$$\vec{h} = s_1(\mathbf{V}\vec{x})$$
$$\vec{z} = s(\mathbf{W}\vec{h}) = s(\mathbf{W}s_1(\mathbf{V}\vec{x}))$$

Neural networks usually have more than one output; this allows us to build classifiers that share hidden units. An interesting advantage with neural networks is that when training multiple classifiers simultaneously through a single neural network, sometimes some classifiers come out better because they're able to take advantage of useful hidden units that first emerged to support another classifier.

It's common to add more hidden layers, and for image recognition, it's common to have anywhere from 8 to 200 hidden layers. There are many variations to experiment with as well (ex. connections that go forward more than one layer).

17.3 Training

We usually use stochastic or batch gradient descent to train neural networks. We pick a loss function $L(\vec{z}, \vec{y})$, where $\vec{z}$ are the predictions and $\vec{y}$ are the true labels; for example, we can use the squared loss $\left\|\vec{z} - \vec{y}\right\|^2$.

We then use a cost function, ex. $J(h) = \frac{1}{n}\sum_{i=1}^{n} L(h(\mathbf{X}_i), \mathbf{Y}_i)$. Here, we use $\mathbf{Y}$ as a matrix here, because now each row corresponds to a sample point, and each column corresponds to an output unit of the neural network.

Now, we want to find the weight matrices $\mathbf{V}$ and $\mathbf{W}$ that minimize J.

Usually, the cost function is not even close to convex; this means that there are many local minima, and it's possible to wind up in a bad minimum. We'll talk later about some clever ways to coax neural networks into better minima.

There is one issue here though; the neural network has a symmetry, in that there is no difference between one hidden unit and any other hidden unit. This means that gradient descent has no way to break the symmetry between hidden units. We can get stuck in a situation where all of the weights out of an input unit have the same value, and all the weights into an output unit have the same value, and they have no way to become different from each other. To avoid this problem (and in hopes of finding a better local minimum), we start with random weights.

Batch gradient descent would look like the following, with $\vec{w}$ containing all the weights in $\mathbf{V}$ and $\mathbf{W}$:

```
1 w⃗ ← vector of random weights
2 repeat
3     w⃗ ← w⃗ − ε∇J(w⃗)
```

Here, we've rewritten all the weights as a vector for ease of notation; in practice, this should be operating directly on the weight matrices **V** and **W**.

Here, it is important to make sure the random weights aren't too big, since a unit can get "stuck" if its output gets too close to zero or one. This is because the gradient $s'(\cdot)$ of the logistic function is close to zero at these extreme values.

Further, instead of batch gradient descent, we can use stochastic gradient descent, which means that we use gradients of one sample point's loss function at each step. In this case, we'd typically shuffle the points in random order, or just pick one randomly at each step.

The hard part is computing the gradient—simply deriving one derivative for each weight means that it takes time linear in the number of edges in the network to compute a derivative for one weight, and we do this for each weight. This is incredibly inefficient, and we'll spend the rest of the lecture improving this runtime.

Specifically, naive gradient computation takes $O(E^2)$ time, where E is the number of edges; backpropagation takes $O(E)$ time.

17.4 Computing Gradients for Arithmetic Expressions

Let's see what it takes to compute the gradient of an arithmetic expression; it basically turns into repeated application of the chain rule from calculus. A diagram of the computations is shown in Fig. 17.5.

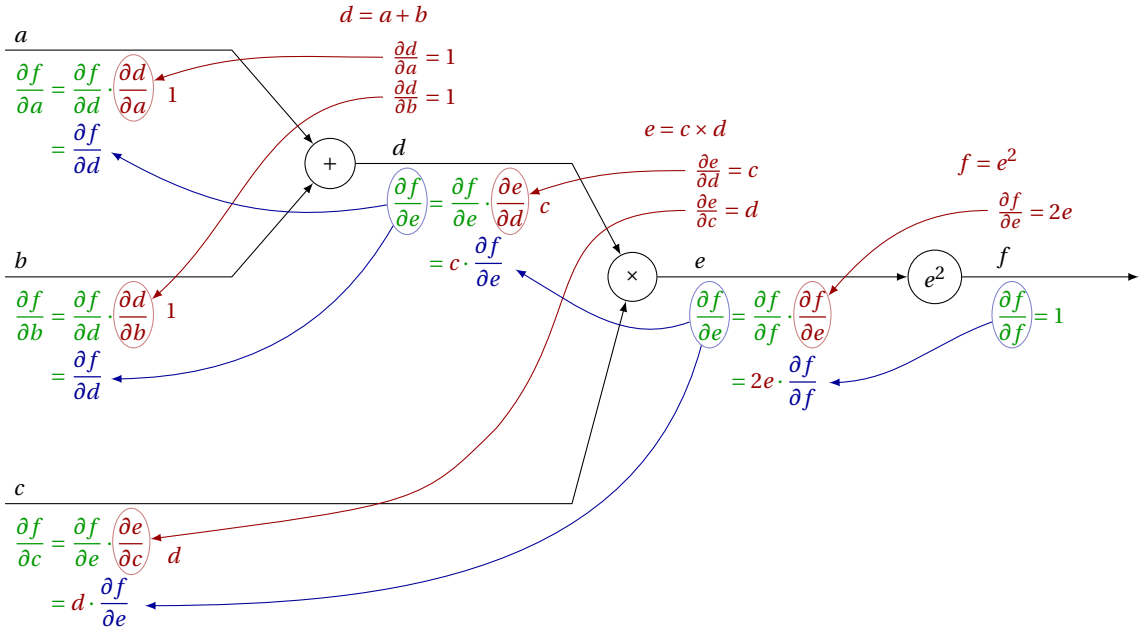


Figure 17.5: Gradient of an arithmetic expression.

Here, the red gradients can be calculated in the forward pass, and the blue gradients can be calculated and passed along in the backward pass after the forward pass. The backward pass gradient calculation is also called *backpropagation*.

What if a unit's output goes to more than one unit? Then, we'd need to understand a more complicated version of the chain rule. Figure 17.6 shows an example from least-squares linear regression.

As before, the red calculations can be done in the forward pass, and the blue calculations are done in the backward pass.

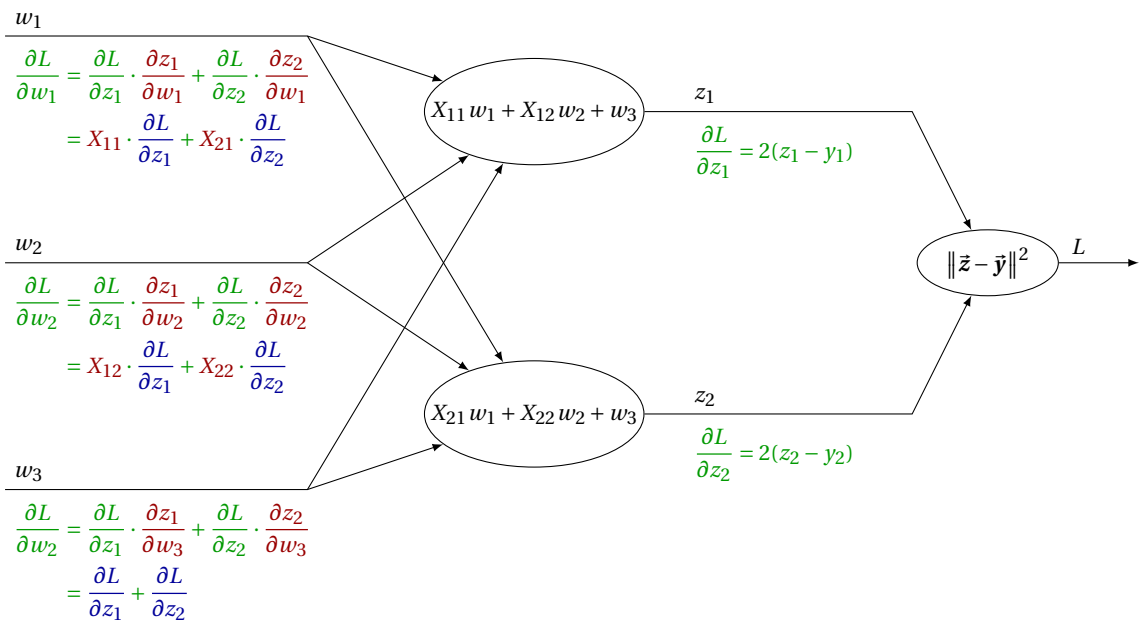


Figure 17.6: Gradient of an expression from least squares linear regression. Backpropagation arrows are omitted here, as they'd make the diagram quite messy.

Further, here we're using another standard rule of multivariable calculus:

$$\frac{\partial}{\partial t} L(z_1(t), z_2(t)) = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial t} + \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial t} = \nabla_{\vec{z}} L \cdot \frac{\partial}{\partial t}\vec{z}.$$

We're doing dynamic programming here; specifically, we're computing the solutions of subproblems, then using each solution to compute the solutions of several bigger problems.

17.5 The Backpropagation Algorithm

Backpropagation is a dynamic programming algorithm for computing the gradients we need in order to perform neural network stochastic gradient descent in linear time with respect to the number of weights.

Here, we let $\mathbf{V}_i^T$ denote row i of the weight matrix $\mathbf{V}$, and likewise for rows of $\mathbf{W}$.

Recall that the gradient of the logistic function is $s'(\gamma) = s(\gamma)\big(1 - s(\gamma)\big)$; computing the gradients, we have

$$\begin{aligned} h_i &= s(\mathbf{V}_i \cdot \vec{\boldsymbol{x}}) & \implies \nabla_{\mathbf{V}_i} h_i &= s'(\mathbf{V}_i \cdot \vec{\boldsymbol{x}})\vec{\boldsymbol{x}} = h_i(1 - h_i)\vec{\boldsymbol{x}} \\ z_j &= s(\mathbf{W}_j \cdot \vec{\boldsymbol{h}}) & \implies \nabla_{\mathbf{W}_j} z_j &= s'(\mathbf{W}_j \cdot \vec{\boldsymbol{h}})\vec{\boldsymbol{h}} = z_j(1 - z_j)\vec{\boldsymbol{h}} \\ & & \nabla_{\vec{\boldsymbol{h}}} z_j &= z_j(1 - z_j)\mathbf{W}_j \end{aligned}$$

With this in mind, we can redraw the diagram in Fig. 17.4 to compute the gradients. This is depicted in Fig. 17.7; note that here we treat the weights $\mathbf{V}$ and $\mathbf{W}$ as inputs, but everything else is exactly the same, just notated differently.

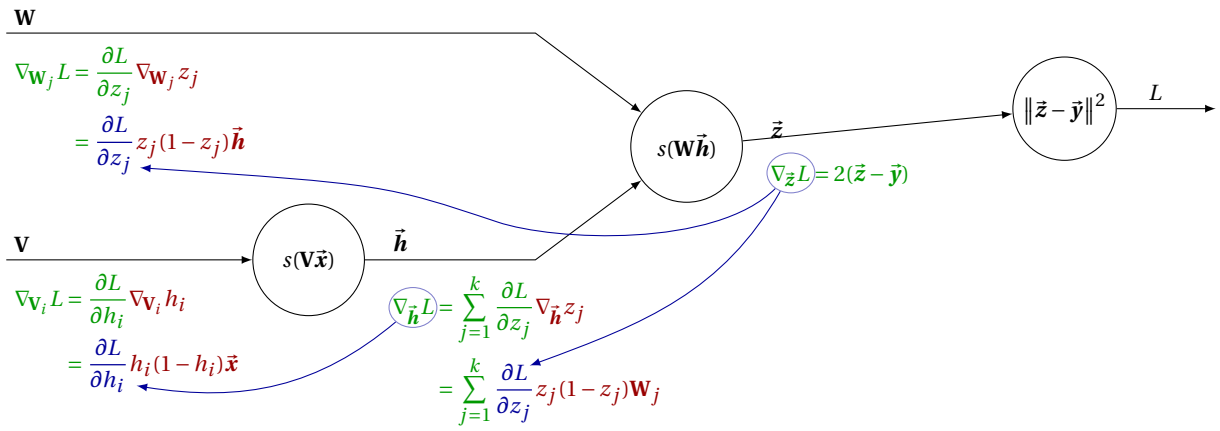


Figure 17.7: Gradient of a neural network with one hidden layer, computing one row at a time

4/4/2022

Lecture 18

Neurobiology; Variations on Neural Networks

18.1 Neurobiology

Neural networks are partly inspired by the workings of actual brains; today, we'll be looking at what we know about biological neurons, and compare/contrast them with neural networks and traditional computation.

Between CPUs and brains, there are a few differences. CPUs are largely sequential, with nanosecond gates, and are fragile if the gate fails. CPUs are superior for arithmetic, logical rules, and perfect key-based memory. In contrast, brains are very parallel, with millisecond neurons, and are fault-tolerant.

In biology, neurons are continually dying. You've probably lost a few since this lecture started, but didn't notice. This is quite interesting, as it points out that memories are stored in our brains in a diffuse representation—there is no single neuron whose death will make you forget that $2+2=4$. Artificial neural networks tend to share that resilience; brains and neural networks store these "memories" all in the same set of weights.

Brains are also superior for vision, speech, and associative memory. Specifically regarding associative memory, brains are good at noticing connections between things, and retrieving a pattern if we specify only a portion of pattern.

Neural networks try to emulate the parallel, associative thinking style of brains, and are among the best techniques we have for many fuzzy problems, including some problems in vision and speech. At the same time, neural networks are also inferior at many traditional computer tasks, ex. multiplying 10-digit numbers or compiling source code.

Here are a few terms to be familiar with:

- **Neuron**: A cell in the brain/nervous system for thinking/communication
- **Action potential** or **spike**: An electrochemical impulse fired by a neuron to communicate with other neurons
- **Axon**: The limb(s) along which the action potential propagates; this is the "output"
- **Dendrite**: Smaller limbs by which the neuron receives info; this is the "input"
- **Synapse**: Connection from one neuron's axon to another's dendrite (some synapses connect axons to muscles or glands)
- **Neurotransmitter**: Chemical released by the axon terminal to stimulate the dendrite

Here are some analogies between artificial neural networks and brains:

- Output of unit $\iff$ firing rate of neuron

  An action potential is "all or nothing", and all action potentials have the same shape and size. The output of a neuron is the frequency at which it fires. Some neurons fire at nearly 1000 times a second, which could be thought of as a strong "1" output; conversely, some can go for minutes without firing, which could be thought of as a strong "0" output.

- Weight of connection $\iff$ synapse strength
- Positive weight $\iff$ excitatory neurotransmitter
- Negative weight $\iff$ inhibitory neurotransmitter

  A typical neuron is either excitatory at all its axon terminals, or inhibitory at all its terminals; it can't switch from one to the other. On the other hand, artificial neural networks have an advantage here in that weights can change freely.

- Linear combination of inputs $\iff$ summation

  A neuron fires when the sum of its inputs, integrated over time, reaches a high enough voltage. However, the neuron body voltage also decays with time, so if the action potentials are coming in slowly enough, the neuron might not fire at all.

- Logistic/sigmoid function $\iff$ firing rate saturation

  A neuron can't fire more than 1000 times a second, nor less than zero times a second. This limits its ability to overpower downstream neurons, and we simulate this behavior with the sigmoid function.

- Weight change/learning $\iff$ synaptic plasticity

  Hebb's rule (1949): "Cells that fire together, wire together."

  Typically, one cell's firing tends to make another cell fire more often, and their excitatory synaptic connection tends to grow stronger. Similarly, there's a reverse rule for inhibitory connections, and there are ways for neurons that aren't even connected to grow connections.

Backpropagation is one part of artificial neural networks for which an analogy is doubtful. There have been some proposals that the brain might do something similar to backpropagation, but it's tenuous.

18.2 Neural Network Variations

Now, we'll look at some basic variations on the standard neural network introduced last lecture, and how some of these variations change backpropagation.

With regression, we usually have a linear output unit; in this case, we omit the sigmoid function.

With classification, to choose from $k \geq 3$ classes, we typically use the softmax function:

$$z_j(t) = \frac{e^{t_j}}{\sum_{i=1}^{k} e^{t_i}} \qquad \frac{\partial z_j}{\partial t_j} = \begin{cases} z_j(1-z_j) & i = j \\ -z_j z_i & i \neq j \end{cases}$$

Here, the output of the softmax $z_j \in (0, 1$, with their sum equal to 1.

We can interpret the output z_j as the probability of the input belonging to class j. With only two classes, we can just use one sigmoid output; it's the equivalent to a 2-way softmax.

18.2.1 Sigmoid Unit Saturation

One problem that occurs with the sigmoid unit is when the output is close to 0 or 1 for most training points (shown in Fig. 18.1). Since $s' = s(1-s) \approx 0$ at those values, gradient descent changes s very slowly; the unit is "stuck". This means that training is slow, and we end up at bad local minima.

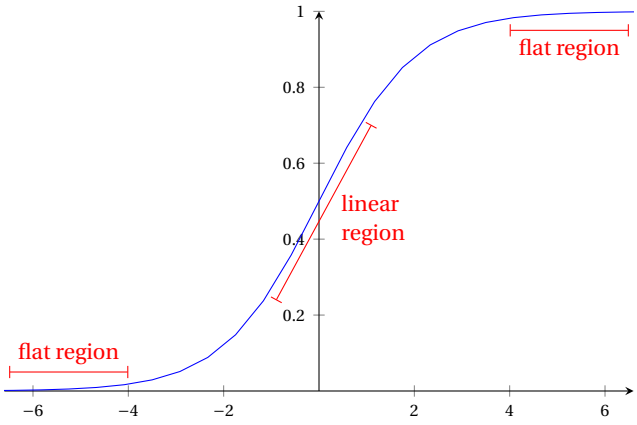


Figure 18.1: The sigmoid function and its flat regions.

This is also called the "vanishing gradient problem". The more layers the network has, the more problematic this problem becomes; most of the early attempts to train deep, many-layered neural networks failed because of this.

Here are some ways that one can mitigate this problem (none of these are complete cures):

1. For a unit with η incoming edges, initialize each incoming edge to a random weight with mean zero, standard deviation $\frac{1}{\sqrt{\eta}}$.

   The bigger the fan-in of a unit (i.e. the more incoming edges), the easier it is to saturate it; this means that we should choose smaller random initial weights for units with a bigger fan-in.

2. Set target values to 0.85 and 0.15 instead of 1 and 0.

   Recall that the sigmoid function can never be 0 or 1; it can only come close. If your target values are 1 and 0, you're pushing the output units into the flat spots; the numbers 0.15 and 0.85 are reasonable, because the sigmoid function achieves its greatest curvature when its output is near 0.21 or 0.79, though one should experiment to find the best values.

3. Modify backpropagation to add a small constant (typically ≈ 0.1) to s'.

   This hacks the gradient so a unit can't get stuck. With this, we aren't doing *steepest* descent anymore, because we're not using the real gradient. However, often we're finding a better descent direction that will get us to a minimum faster. This originates with Scott Fahlman's *Quickprop* algorithm.

4. Use the *cross-entropy* loss function instead of squared error.

   For k-class softmax output, the cross-entropy loss is

   $$L(\vec{\boldsymbol{z}}, \vec{\boldsymbol{y}}) = -\sum_{i=1}^{k} y_i \ln z_i,$$

   where $\vec{\boldsymbol{z}}$ is the k-vector of predictions and $\vec{\boldsymbol{y}}$ is the k-vector of true labels.

   It's strongly recommended to choose labels such that $\sum_{i=1}^{k} y_i = 1$. Typically, people choose one label to be 1 and the others to be 0. However, with idea (2) above, it may be wiser to use less extreme values.

   From here on, we will assume that the target labels sum to 1.

   For a (scalar) sigmoid output, $L(z, y) = -y \ln z - (1 - y) \ln(1 - z)$, i.e. logistic loss. The derivatives for a k-class

softmax backpropagation are:

$$\frac{\partial L}{\partial z_j} = -\frac{y_j}{z_j}$$
$$\nabla_{\mathbf{W}_i} L = \left(\sum_{j=1}^{k} \frac{\partial L}{\partial z_j}\frac{\partial z_j}{\partial t_i}\right)\nabla_{\mathbf{W}_i} t_i = \left(-\frac{y_i}{z_i}z_i + \sum_{j=1}^{k}\frac{y_j}{z_j}z_j z_i\right)\vec{\boldsymbol{h}} = (z_i - y_i)\vec{\boldsymbol{h}}$$
$$\nabla_{\mathbf{W}} L = (\vec{\boldsymbol{z}} - \vec{\boldsymbol{y}})\vec{\boldsymbol{h}}^T$$
$$\nabla_{\vec{\boldsymbol{h}}} L = \sum_{j=1}^{k}\frac{\partial L}{\partial z_j}\sum_{i=1}^{k}\frac{\partial z_j}{\partial t_i}\nabla_{\vec{\boldsymbol{h}}} t_i = \sum_{j=1}^{k} -\frac{y_j}{z_j}\left(z_j\mathbf{W}_j - \sum_{i=1}^{k} z_j z_i \mathbf{W}_i\right) = \mathbf{W}^T\vec{\boldsymbol{z}} - \sum_{j=1}^{k} y_j \mathbf{W}_j = \mathbf{W}^T(\vec{\boldsymbol{z}} - \vec{\boldsymbol{y}})$$

Similar to option (2), cross-entropy loss helps avoid stuck output units, though it does not help stuck hidden units.

Further, cross-entropy loss is only for sigmoid and softmax outputs; with regression, we typically use linear outputs, which don't saturate, so the squared error loss is better for them anyways.

5. Replace sigmoids with ReLUs: *rectified linear units*. The ReLU function is also called the *ramp function*, or the *hinge function*, shown below and in the plot in Fig. 18.2.

$$r(\gamma) = \max(0, \gamma) \qquad r'(\gamma) = \begin{cases} 1 & \gamma \geq 0 \\ 0 & \gamma < 0 \end{cases}$$

The derivative is not defined at zero, but we pretend that it is defined.

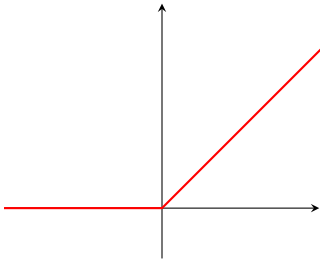


Figure 18.2: ReLU function plot

The ReLU function is popular for networks with many layers and large training sets. One nice thing about the ramp function is that it and its gradient are very fast to compute; exponentials on the other hand are slow to compute.

Even though ReLUs are linear in half of their range, they're still nonlinear enough to easily compute functions like XOR.

Obviously, the gradient is sometimes zero, but fortunately it's rare for a ReLU's gradient to be zero for *all* of the training data; it's usually zero for only some sample points.

The output of a ReLU can be arbitrarily large, which creates the danger that it might overwhelm units downstream. This is called the "exploding gradient problem;" it's not a big problem in shallow networks, but it is a big problem in deep or recurrent networks.

Note that option (5) makes options (2), (3), and (4) irrelevant.

4/6/2022

Lecture 19

Better Neural Network Training, Convolutional Neural Networks

19.1 Heuristics

19.1.1 Heuristics for Faster Training

A big disadvantage of neural networks is that they take a long time to train, compared to other classification methods we've studied. Today, we'll talk about some ways to speed them up. Unfortunately, you'll usually have to experiment with techniques and hyperparameters to find which ones will help with your particular application.

- Fix the vanishing gradient problem (as described in the previous lecture).
- Stochastic gradient descent; it's usually faster than batch gradient descent on large, redundant data sets.

  Batch gradient descent walks downhill on one cost function, whereas stochastic gradient descent takes a very short step downhill on one point's loss function and then another short step on another point's loss function.

  The cost function is the sum of the loss function over all the sample points, so one batch step behaves similarly to n stochastic steps, and takes roughly the same amount of time. However, if you have many different examples of the digit "9", they each contain a lot of redundant information, and stochastic gradient descent learns the redundant information more quickly.

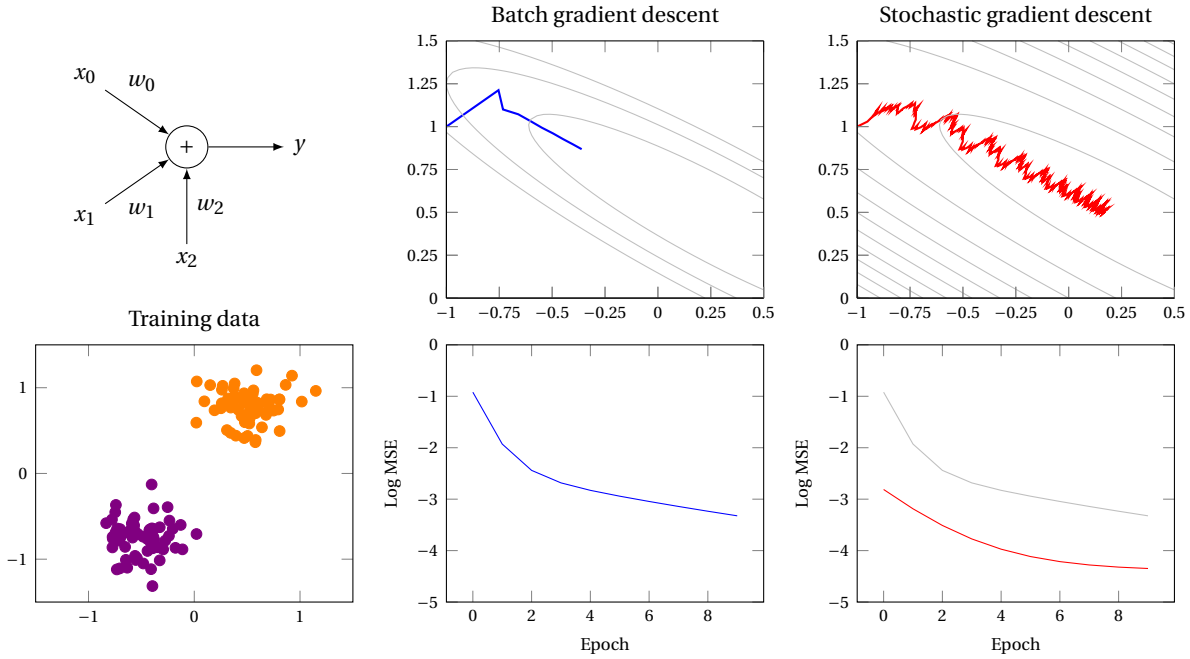


Figure 19.1: Comparison between stochastic gradient descent and batched gradient descent for a simple one-layer neural network

As illustrated for the simple network in Fig. 19.1, stochastic descent tends to decrease error much faster than batch descent.

One *epoch* presents every training point once; training usually takes many epochs, but if the sample size is large (and carries lots of redundant information), it can take less than one.

- Normalizing the data; specifically, center each feature so the mean is zero, and scale each feature so that the variance is ≈ 1. An example is shown in Fig. 19.2.

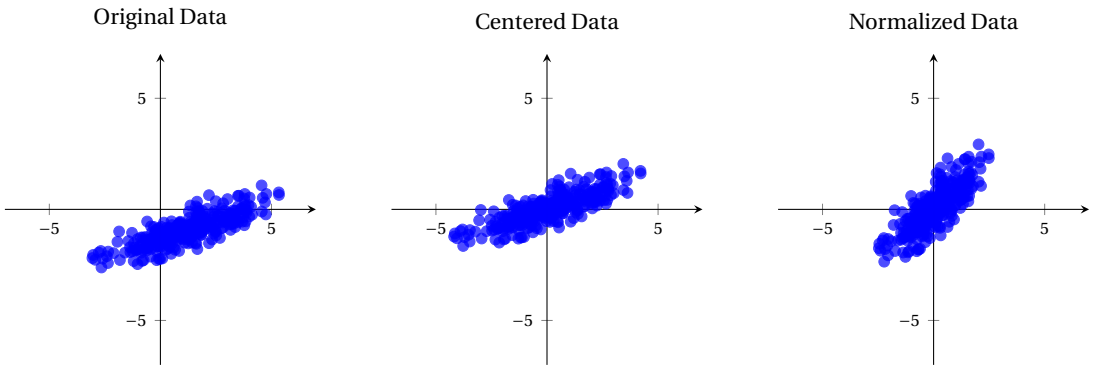


Figure 19.2: Example of centering and normalizing data

The first step of centering the data seems to make it easier for hidden units to get into a good operating region of the sigmoid or ReLU.

The second step of scaling the features makes the objective function much better conditioned, so gradient descent converges faster. Specifically, skewed data often leads to an objective function with an ill-conditioned (i.e. highly eccentric) Hessian. Gradient descent in these functions can be painfully slow; normalizing helps by reducing the eccentricity. Another thing that helps this issue is momentum, which we'll discuss shortly.

- "Centering" the hidden units.

  Specifically, replace sigmoids with

  $$\tanh \gamma = \frac{e^{\gamma} - e^{-\gamma}}{e^{\gamma} + e^{-\gamma}} = 2s(2\gamma) - 1.$$

  This function ranges from -1 to 1 instead of from 0 to 1. If you use tanh units, don't forget that you also need to change backpropagation to replace s' with the derivative $1 - \tanh^2 \gamma$. Good output target values change to roughly 0.7 and -0.7.

- Different learning rates for each layer of weights.

  Earlier layers have smaller gradients, so they'd need larger learning rates.

- Emphasizing schemes; typically, neural networks learn the most redundant examples very quickly, and the most rare examples slowly. This means that we should try to emphasize the uncommon examples.

  To do this, we present examples from rare classes more often, or with a bigger learning rate; we can do the same for misclassified examples.

  As a warning, emphasizing schemes can backfire if there exist very bad outliers.

- Second-order optimization.

  Unfortunately, Newton's method is completely impractical, because the Hessian is too large and expensive to compute. There have been a lot of attempts to incorporate curvature information into neural network learning in cheaper ways, but none of them are popular yet.

  One option is with the *nonlinear conjugate gradient method*; this works well for small networks with small datasets using regression. Further, this should only be used with batched descent; it tends to be too slow with redundant data.

  Another option is with Stochastic Levenberg Marquardt; this approximates a diagonal Hessian. The authors claim that convergence is typically three times faster than well-tuned stochastic gradient descent, though the algorithm is complicated.

- Acceleration schemes; RMSprop, Adam, AMSGrad are the most popular.

19.1.2 Heuristics for Avoiding Bad Local Minima

- Fix vanishing gradients, as described in the previous lecture.
- Stochastic gradient descent; "random" motion tends to get you out of shallow local minima.

  Even if you're at the local minimum of the cost function, the loss function of each sample point will be trying to push you in a different direction. Stochastic gradient descent looks like a random walk or Brownian motion, which can shake you out of a shallow minimum.
- Momentum; gradient descent can be modified to change a "velocity" $\Delta\vec{\boldsymbol{w}}$ slowly. This carries us through shallow local minima into deeper ones.

```
1 Δw ← −ε∇J(w)
2 repeat
3     w ← w + Δw
4     Δw ← −ε∇J(w) + βΔw
```

  This is tends to perform well for both batch and stochastic gradient descent. The hyperparameter $\beta < 1$ specifies how much momentum persists from iteration to iteration.

  There have been conflicting advice on β from researchers; some set it to 0.9, and some set it close to zero. Geoff Hinton suggests starting at 0.5 and slowly increasing to 0.9 as the gradients get small.

  If β is large, you should usually choose ε to be small in order to compensate, though you might still use a large ε in the first line of the algorithm so the initial velocity is reasonable.

  One problem with momentum is that once it gets close to a good minimum, it oscillates around the minimum, but it's more likely to get close to a good minimum in the first place.

19.1.3 Heuristics to avoid Overfitting

1. An ensemble of neural networks, with random initial weights and bagging.

   In a past lecture, we saw how well ensemble learning works for decision trees; it works well for neural networks too. The combination of random initial weights and bagging helps ensure that each neural network comes out differently. However, as one would expect, ensembles of neural networks are slow to train.

2. ℓ_2 regularization, i.e. *weight decay*. Specifically, add $\lambda\|\vec{\boldsymbol{w}}\|^2$ to the cost or loss function, where $\vec{\boldsymbol{w}}$ is the vector of all weights. The effect here is that $-\varepsilon\frac{\partial J}{\partial w_i}$ has an extra $-2\varepsilon\lambda w_i$ term.

   We do this for the same reason we do it in ridge regression; penalizing large weights reduces overfitting by reducing the variance of the method.

   With a neural network, it's not clear whether penalizing the bias terms is good or bad. If you penalize the bias terms, regularization has the effect of drawing each ReLU or sigmoid unit closer to the center of its operating region. It is suggested to try both ways and use validation to decide whether you should penalize the bias terms or not.

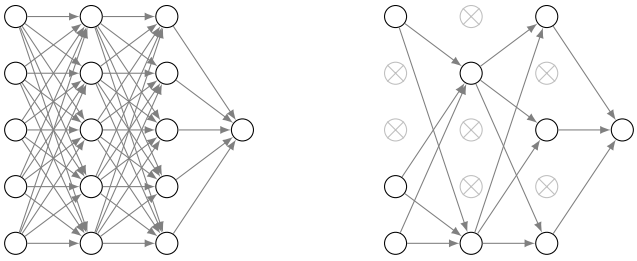
3. *Dropout* emulates an ensemble in one network. An example is shown in Fig. 19.3.

   During training, we temporarily disable a random subset of units, along with all the edges in and out of those units. It seems to work well to disable each hidden unit with probability 0.5, and to disable input units with a smaller probability. Doing stochastic descent, we frequently change which random subset of units is disabled.

   The authors claim that their method gives even better generalization than ℓ_2 regularization. It gives some of the advantages of an ensemble, but it's faster to train.

4. Fewer hidden units.

   The number of hidden units is a hyperparameter you can use to adjust the bias-variance tradeoff. If there's too few, you can't learn well, but if there's too many, you may overfit. ℓ_2 regularization and dropout make it safer to have too many hidden units, so it's less critical to find just the right number.



(a) Network without dropout **(b)** Network with dropout

Figure 19.3: Example of a network with and without dropout

19.2 Convolutional Neural Networks

Convolutional neural networks have caused a big resurgence of interest in neural networks in the last decade.

With vision, we have large images as inputs. For example, a 200×200 image has 40 000 pixels; if we connect to them all to 40 000 hidden units, we have 1.6 billion connections.

This is a lot of parameters to train; in this way, neural networks are often *overparameterized*: there are too many weights, with too little data.

As a rule of thumb, if you have hugely many weights, you want a huge amount of data to train them. A bigger problem with having billions of weights is that the network becomes very slow to train or even to use.

Researchers have addressed these problems by taking inspiration from the neurology of the visual system. Remember that early in the semester, we mentioned that you can get better performance on the handwriting recognition task by using edge detectors. Edge detectors have two interesting properties; first, each edge detector looks at just one small part of the image; second, the edge decision computation is the same no matter which part of the image you apply it to.

As such, let's apply these two properties to neural network design:

- Local connectivity: A hidden unit (in an early layer) connects only to a small patch of units in the previous layer.

  This improves the overparameterization problem, and speeds up both training and classification considerably.

- Shared weights: Groups of hidden units share the same set of input weights, called a *mask*, or a *filter*, or a *kernel*. Each mask operates on every patch of an image.

  The number of units in the first hidden layer is the number of masks, multiplied by the number of patches.

  If a neural network learns to detect edges on one patch, *every* patch has an edge detector, since the mask that detects edges is applied to every patch.

  Convolution: the same linear transformation applied to different parts of the input by shifting.

  Shared weights improve the overparameterization problem even more, because shared weights means fewer weights; it's a kind of regularization.

  Shared weights also have another big advantage. Suppose that gradient descent starts to develop an edge detector; that edge detector is being trained on *every* part of every image, not just on one spot. That's good, because edges appear at different locations in different images; the location no longer matters, as the edge detector can learn from edges in every part of the image.

In a neural network, you can think of hidden units as features that we learn, as opposed to features that you code up yourself. Convolutional neural networks take them to the next level by learning features from multiple patches simultaneously and then applying those features everywhere, not just in the patches where they were originally learned.

4/11/2022

Lecture 20

Unsupervised Learning, Principal Component Analysis

20.1 Unsupervised Learning

With unsupervised learning, we have sample points, but no labels; we have no classes, no y-values, and nothing to predict. The goal here is to discover structure in the data.

Some 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.

  When we use maximum likelihood estimation to fit Gaussians to sample points, that's density estimation, but we can also fit functions more complicated than Gaussians.

20.2 Principal Component Analysis

With principal component analysis (PCA), our goal given sample points in $\mathbb{R}^d$ is to find k directions that capture most of the variation. (This is dimensionality reduction.)

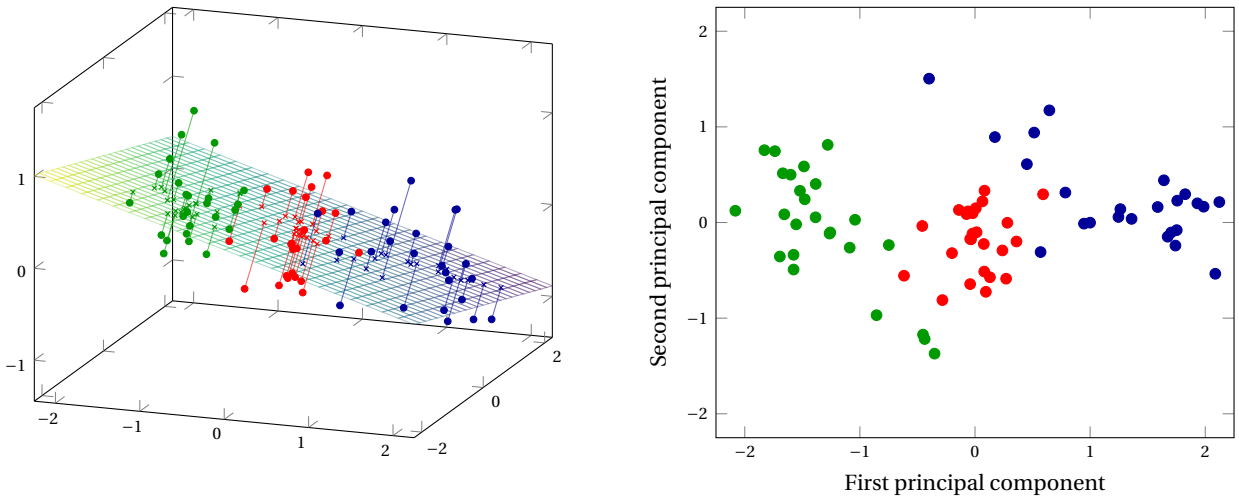


Figure 20.1: An example of PCA applied in a 3D dataset, with two principal components

Figure 20.2 shows the MNIST digits projected into 2D. Two dimensions aren't enough to fully separate the digits, but one thing to note is that the digits 0 (red) and 1 (orange) are well on their way to being separated.

Why perform PCA? Firstly, reducing the number of dimensions makes some computations cheaper, for example, regression. PCA also removes irrelevant dimensions to reduce overfitting in learning algorithms. PCA is much like subset selection, but the "features" aren't axis aligned; they're linear combinations of input features. With PCA, we find a small basis for representing variations in complex things, like faces, genes, etc.

Sometimes PCA is used in preprocessing before regression or classification for the first two reasons (cheaper computations, removing irrelevant dimensions).

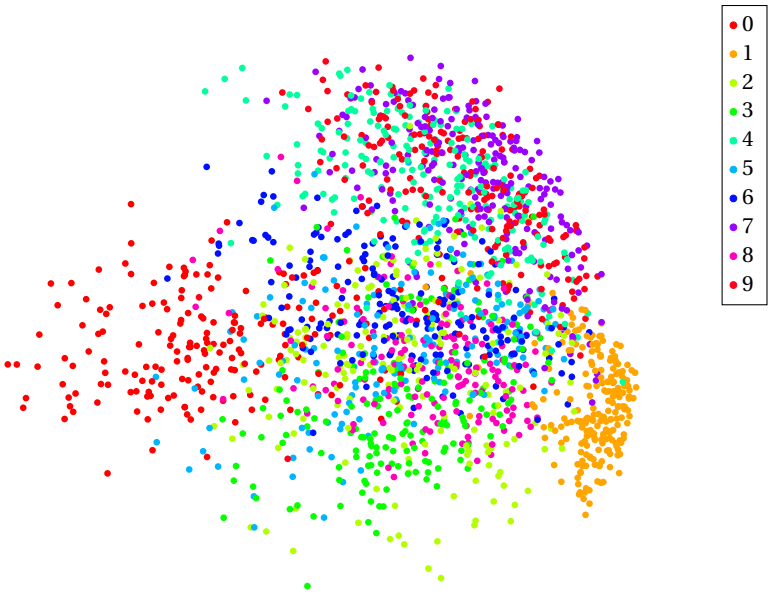


Figure 20.2: The MNIST dataset, projected into 2D (from 784D)

Let's look at the process of computing the PCA. Let $\mathbf{X}$ be an $n \times d$ design matrix (there is no fictitious dimension here). From now on, let us assume that $\mathbf{X}$ is centered, i.e. $\mu_{\mathbf{X}} = 0$. As usual, we can center the data by computing the mean $\vec{\boldsymbol{x}}$-value, then subtracting the mean from each sample point.

First, we can start by looking at what happens if we pick just one principal direction.

Let $\vec{\boldsymbol{w}}$ be a unit vector. The *orthogonal projection* of point $\vec{\boldsymbol{x}}$ onto vector $\vec{\boldsymbol{w}}$ is

$$\vec{\tilde{\boldsymbol{x}}} = (\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{w}})\, \vec{\boldsymbol{w}}.$$

If $\vec{\boldsymbol{w}}$ is not a unit vector, then instead

$$\vec{\tilde{\boldsymbol{x}}} = \frac{\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{w}}}{\|\vec{\boldsymbol{w}}\|^2}\, \vec{\boldsymbol{w}}.$$

The idea here is that we pick the best direction $\vec{\boldsymbol{w}}$, then project all the data down onto $\vec{\boldsymbol{w}}$ so we can analyze it in a one-dimensional space. Of course, we lose a lot of information when we project down from d dimensions to just one, so suppose we pick several directions.

These directions span a subspace, and we want to project points orthogonally onto the subspace. This is easy *if* the directions are orthogonal to each other.

Given orthonormal directions $\vec{\boldsymbol{v}}_1, \ldots, \vec{\boldsymbol{v}}_k$, we have

$$\vec{\tilde{\boldsymbol{x}}} = \sum_{i=1}^{k} (\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{v}}_i)\, \vec{\boldsymbol{v}}_i.$$

Oftentimes, we want just the k *principal coordinates* $\vec{\boldsymbol{x}} \cdot \vec{\boldsymbol{v}}_i$ in the principal component space; we don't usually want the actual projected point in $\mathbb{R}^d$.

$\mathbf{X}^T\mathbf{X}$ is square symmetric, positive semidefinite $d \times d$ matrix. Since it's symmetric, its eigenvalues are real; let $0 \le \lambda_1 \le \lambda_2 \le \cdots \le \lambda_d$ be its eigenvalues.

Let $\vec{\boldsymbol{v}}_1, \vec{\boldsymbol{v}}_2, \ldots, \vec{\boldsymbol{v}}_d$ be the corresponding orthogonal *unit* eigenvectors; these are the *principal components*. The most important principal components will be the ones with the greatest eigenvalues. We'll show this in three different ways.

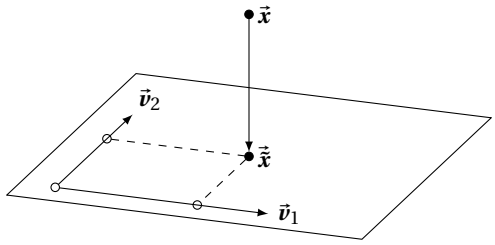


Figure 20.3: Example of projection onto a 2D subspace

20.2.1 Fitting Gaussian via MLE

In the first derivation of PCA, we fit a Gaussian to data with maximum likelihood estimation. Specifically, we choose the k Gaussian axes of greatest variance.

Recall that MLE estimates a covariance matrix $\hat{\Sigma} = \frac{1}{n}\mathbf{X}^T\mathbf{X}$, presuming $\mathbf{X}$ is centered.

This gives us our PCA algorithm:

- Center $\mathbf{X}$.
- Optionally, normalize $\mathbf{X}$.

  If the units of measurement are different between features, then you should normalize, since it's bad for principal components to depend on an arbitrary choice of scaling.

  Otherwise, we usually don't; if several features have the same unit of measurement, but some of them have smaller variance than others, that difference is usually meaningful.
- Compute the unit eigenvectors and eigenvalues of $\mathbf{X}^T\mathbf{X}$.
- Optionally, choose k based on the eigenvalue sizes.
- For the best k-dimensional subspace, pick the eigenvectors $\vec{v}_{d-k+1}, \ldots, \vec{v}_d$

  That is, choose the k eigenvectors corresponding to the largest eigenvalues.
- Compute the principal coordinates $\vec{x} \cdot \vec{v}_i$ of the training/test data.

  When we do this projection, we have two choices: we can un-center the input training data before projecting it, *or*, we can translate the test data by the same vector we used to translate the training data when we centered it.

20.2.2 Maximizing Sample Variance

A second derivation of PCA comes from finding the direction $\vec{w}$ that maximizes the sample variance of the projected data. In other words, when we project the data down, we don't want it all to bunch up together; we want to keep it as spread out as possible.

Formally, we want to find the $\vec{w}$ that maximizes

$$\text{Var}(\{\tilde{\mathbf{X}}_1, \tilde{\mathbf{X}}_2, \ldots, \tilde{\mathbf{X}}_n\}) = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{X}_i \cdot \frac{\vec{w}}{\|\vec{w}\|}\right)^2 = \frac{1}{n}\frac{\|\mathbf{X}\vec{w}\|^2}{\|\vec{w}\|^2} = \frac{1}{n}\underbrace{\frac{\vec{w}^T\mathbf{X}^T\mathbf{X}\vec{w}}{\vec{w}^T\vec{w}}}_{\textit{Rayleigh quotient}}.$$

The last fraction is a well-known construction called the Rayleigh quotient of $\mathbf{X}^T\mathbf{X}$ and $\vec{w}$. How do we maximize this?

If $\vec{w}$ is an eigenvector $\vec{v}_i$, the Rayleigh quotient would be equal to λ_i:

$$\frac{\|\mathbf{X}\vec{v}_i\|^2}{\|\vec{v}_i\|^2} = \frac{\|\lambda_i\vec{v}_i\|^2}{\|\vec{v}_i\|^2} = \lambda_i\frac{\|\vec{v}_i\|^2}{\|\vec{v}_i\|^2} = \lambda_i.$$

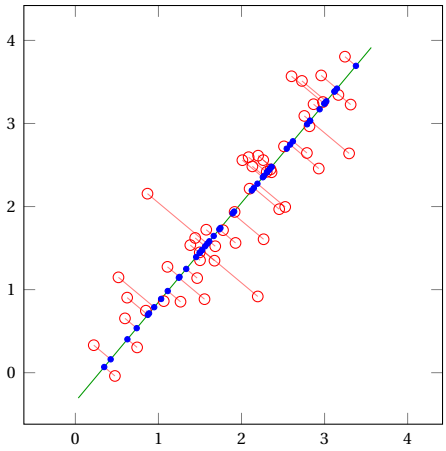


Figure 20.4: Points projected onto a line that maximizes the sample variance of the projected points

This means that out of all eigenvectors, $\vec{\boldsymbol{v}}_d$ achieves the maximum variance $\frac{1}{n}\lambda_d$.

One can also show that $\vec{\boldsymbol{v}}_d$ beats every other vector too, since every vector $\vec{\boldsymbol{w}}$ is a linear combination of eigenvectors. This means that its Rayleigh quotient will be a convex combination of eigenvalues.

All of this means that the top eigenvector gives us the best direction, but we typically want k directions. This means that after we've picked one direction, we need to pick a direction that is orthogonal to the best direction; subject to this constraint, we again pick the direction that maximizes the sample variance.

20.2.3 Minimizing Mean Squared Projection Distance

A third derivation is to find the vector $\vec{\boldsymbol{w}}$ that minimizes the mean squared projection distance.

You can think of this as a kind of least-squares linear regression, except with one subtle but important change. Instead of measuring the error in a fixed vertical direction, we're measuring the error in a direction orthogonal to the principal component direction we choose. This is shown in Fig. 20.5.

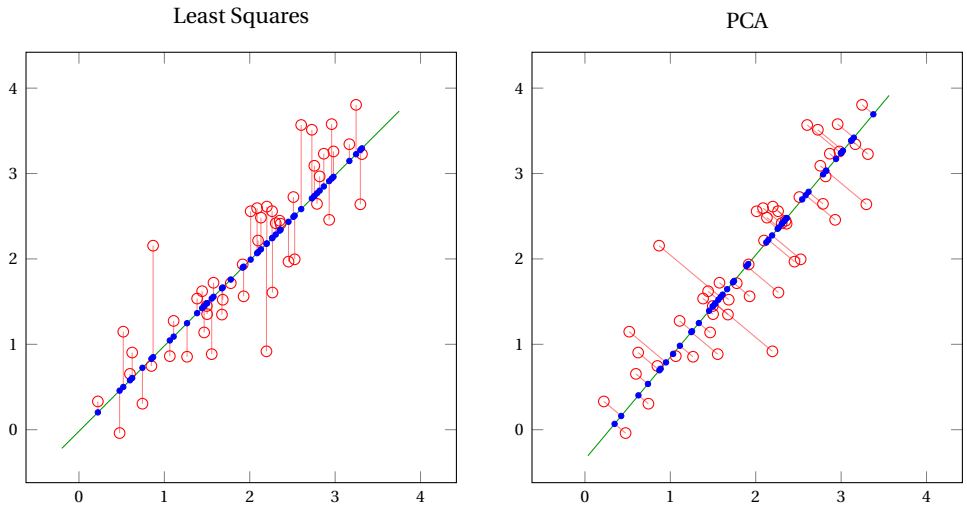


Figure 20.5: Comparison between least squares and PCA

Formally, we want to find the $\vec{w}$ that minimizes

$$\begin{aligned}\sum_{i=1}^{n} \left\|\mathbf{X}_i - \tilde{\mathbf{X}}_i\right\|^2 &= \sum_{i=1}^{n} \left\|\mathbf{X}_i - \frac{\mathbf{X}_i \cdot \vec{w}}{\|\vec{w}\|^2}\vec{w}\right\|^2 \\ &= \sum_{i=1}^{n} \left(\|\mathbf{X}_i\|^2 - \left(\mathbf{X}_i \cdot \frac{\vec{w}}{\|\vec{w}\|}\right)^2\right) \\ &= \sum_{i=1}^{n} \|\mathbf{X}_i\|^2 - \sum_{i=1}^{n} \left(\mathbf{X}_i \cdot \frac{\vec{w}}{\|\vec{w}\|}\right)^2 \\ &= (\text{constant}) - n \cdot (\text{variance from derivation 2})\end{aligned}$$

This suggests that minimizing the mean-squared projection distance is equivalent to minimizing the variance.

From this point onward, we carry on with the same reasoning as derivation 2.

20.3 Eigenfaces

One application of PCA is with face recognition. Here, the data matrix $\mathbf{X}$ contains n images of faces, d pixels each. In particular, a 200×200 image would be represented as a vector of length 40 000.

With face recognition, we're given a query face, and our goal is to find the nearest neighbor in $\mathbb{R}^d$, comparing with all training faces.

The problem is that each query takes $\Theta(nd)$ time, since we'd need to iterate through every sample, and perform a comparison with each feature. To speed this up, we can run PCA on the faces, and reduce it to a much smaller dimension d'.

Now, nearest neighbor takes $O(nd')$ time, and possibly even less. We'll talk about speeding up nearest-neighbor search in a future lecture. If the dimension is small enough, you can sometimes do better than linear time.

4/13/2022

Lecture 21

Singular Value Decomposition, Clustering

21.1 Singular Value Decomposition

Recall that for PCA, we need to compute $\mathbf{X}^T\mathbf{X}$ and its eigenvectors and eigenvalues. A problem with this is that the computation takes $\Theta(nd^2)$ time, and further, if $\mathbf{X}^T\mathbf{X}$ is poorly conditioned, then we get numerically inaccurate eigenvectors. The SVD improves on both of these problems.

Earlier this semester, we learned about the eigendecomposition of a square, symmetric matrix. Unfortunately, non-symmetric matrices don't eigendecompose as nicely, and non-square matrices don't have eigenvectors at all. However, the SVD is a decomposition that works for all matrices, even if they're not symmetric and not square.

Definition 21.1: Singular Value Decomposition

We can always find a *singular value decomposition* $\mathbf{X} = \mathbf{UDV}^T$ such that

$$\mathbf{X} = \underbrace{\begin{bmatrix} | & | & & | \\ \vec{u}_1 & \vec{u}_2 & \cdots & \vec{u}_d \\ | & | & & | \end{bmatrix}}_{\mathbf{U}} \underbrace{\begin{bmatrix} \delta_1 & & & 0 \\ & \delta_2 & & \\ & & \ddots & \\ 0 & & & \delta_4 \end{bmatrix}}_{\mathbf{D}} \underbrace{\begin{bmatrix} — & \vec{v}_1 & — \\ — & \vec{v}_2 & — \\ & \vdots & \\ — & \vec{v}_d & — \end{bmatrix}}_{\mathbf{V}^T}$$

Specifically, $\mathbf{U}$ is the orthonormal matrix of *left singular vectors* of $\mathbf{X}$, $\mathbf{D}$ is the diagonal matrix of nonnegative

singular values of $\mathbf{X}$, and $\mathbf{V}$ is the orthonormal matrix of *right singular vectors* of $\mathbf{X}$.

The singular value decomposition can also be rewritten as a sum of outer products:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T = \sum_{i=1}^{d} \delta_i \vec{\boldsymbol{u}}_i \vec{\boldsymbol{v}}_i^T.$$

If $n \geq d$, then $\mathbf{U}$ is $n \times d$, $\mathbf{D}$ is $d \times d$, and $\mathbf{V}^T$ is $d \times d$. If $n < d$, then $\mathbf{U}$ is $n \times n$, $\mathbf{D}$ is $n \times n$, and $\mathbf{V}^T$ is $n \times d$.

Some of the singular values might be zero, and the number of nonzero singular values is equal to the rank of $\mathbf{X}$. That is, if $\mathbf{X}$ is a centered design matrix for sample points that all lie on a line, there is only one nonzero singular value; if the centered sample points span a subspace of dimension r, then there are r nonzero singular values.

Theorem 21.2

Consider the SVD $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$. The vectors $\vec{\boldsymbol{v}}_i$ (i.e. rows of $\mathbf{V}^T$) are eigenvectors of $\mathbf{X}^T\mathbf{X}$, with corresponding eigenvalue δ_i^2.

Proof. We have

$$\mathbf{X}^T\mathbf{X} = \mathbf{V}\mathbf{D}\mathbf{U}^T\mathbf{U}\mathbf{D}\mathbf{V} = \mathbf{V}\mathbf{D}^2\mathbf{V}^T,$$

and the last expression is an eigendecomposition of $\mathbf{X}^T\mathbf{X}$. □

PCA requires the eigenvectors of $\mathbf{X}^T\mathbf{X}$, and it turns out that the columns of $\mathbf{V}$ are exactly what we want. The SVD also tells us their eigenvalues, which are the squares of the singular values. This is also why SVD is more numerically stable; the ratios between singular values are smaller than the ratios between eigenvalues.

If $n < d$, then $\mathbf{V}$ will omit some of the eigenvectors that have eigenvalue zero, but those are useless for PCA anyways.

It turns out that we can find the k greatest singular values and the corresponding singular vectors in $O(ndk)$ time. This means that we can save time by computing only some of the singular vectors. There are approximate, randomized algorithms that are even faster, producing an approximate SVD in $O(nd \log k)$ time. These are starting to become popular in algorithms for very big data.

One important note is that row i of $\mathbf{U}\mathbf{D}$ gives the principle coordinates of sample point $\mathbf{X}_i$; i.e. for all j, $\mathbf{X}_i \cdot \vec{\boldsymbol{v}}_j = \delta_j \mathbf{U}_{ij}$.

This means that we don't need to explicitly compute the inner products $\mathbf{X}_i \cdot \vec{\boldsymbol{v}}_j$; the SVD has already done it for us.

21.2 Clustering

The goal of clustering is to partition data into clusters such that points in a cluster are more similar than across clusters.

Why is this useful? Here are some examples of where we'd apply clustering:

- Discovery: Find songs similar to songs you like; determine market segments
- Hierarchy: Find good taxonomy of species from genes
- Quantization: Compress a dataset by reducing choices
- Graph Partitioning: Image segmentation; find groups in social networks

21.2.1 k-means Clustering

With k-means clustering, the goal is to partition n points into k disjoint clusters.

We assign each input point $\mathbf{X}_i$ a cluster label $y_i \in [1, k]$, and denote cluster i's mean as

$$\vec{\boldsymbol{\mu}}_i = \frac{1}{n_i} \sum_{y_j = i} \mathbf{X}_j,$$

given that there are n_i points in cluster i.

The formal optimization problem is to find $\vec{\boldsymbol{y}}$ that minimizes

$$\sum_{i=1}^{k} \sum_{y_j = i} \left\| \mathbf{X}_j - \vec{\boldsymbol{\mu}}_i \right\|^2.$$

This problem is NP-hard, and can be solvable in $O(nk^n)$ time by trying every partition.

However, we can use heuristic to get an approximate solution; this is Lloyd's algorithm. Specifically, we alternate between (1) fix y_j's, update $\vec{\boldsymbol{\mu}}_i$'s, and (2) fix $\vec{\boldsymbol{\mu}}_i$, update y_j's. We halt when step (2) changes no assignments.

One can show through calculus that in step (1) that the optimal $\vec{\boldsymbol{\mu}}_i$ is the mean of the points in cluster i. One can also show in step (2) that the optimal $\vec{\boldsymbol{y}}$ assigns each point $\mathbf{X}_j$ to the closest center $\vec{\boldsymbol{\mu}}_i$. If there is a tie, and one of the choices is for $\mathbf{X}_j$ to stay in the same cluster as the previous iteration, always take that choice.

Visually, a few steps of the algorithm are shown in Fig. 21.1. Here, the odd steps relabel the data points, and the even steps compute the new means.

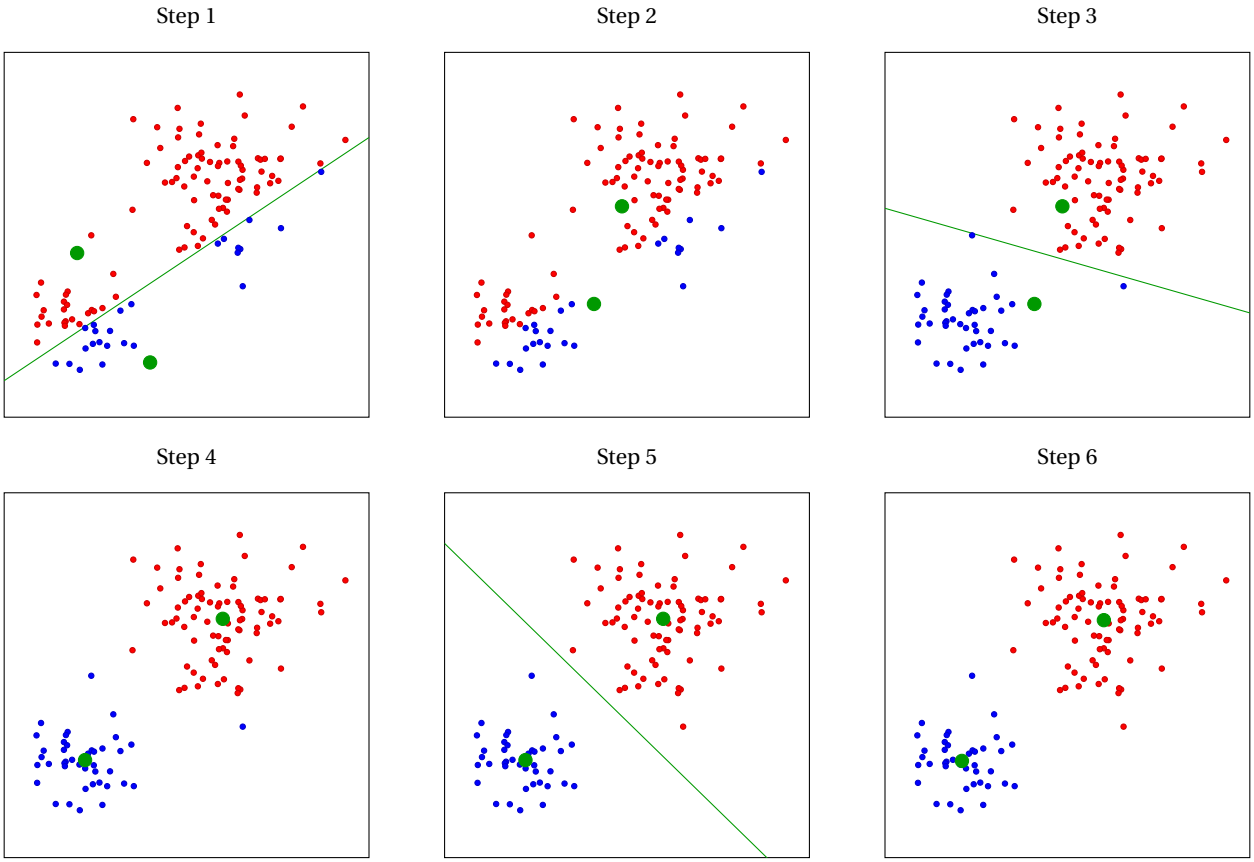


Figure 21.1: An example of 2-means clustering

Both steps will always decrease the objective function, unless they change nothing. This means that the algorithm never returns to a previous assignment; it must terminate, as there are only finitely many assignments.

This means that Lloyd's algorithm never loops forever, but it doesn't say anything optimistic about the runtime, as we may see $O(k^n)$ different assignments before we halt. In theory, one can construct point sets in the plane that take an exponential number of iterations, but those don't come up in practice; Lloyd's algorithm is usually very fast in practice.

Lloyd's algorithm also usually finds a local minimum, not a global minimum—this should be expected, as the problem is NP-hard.

There are also a few ways to initialize the means in k-means clustering:

- Forgy method: choose k random sample points to be the initial $\vec{\boldsymbol{\mu}}_i$'s, and start with step (2); update the labels.
- Random partition: randomly assign each sample point to a cluster, and start with step (1); update the means.
- k-means++: like Forgy, but we use a biased distribution instead. Here, each center is chosen with a preference for points far from previous centers.

The last choice, k-means++, is a little bit more work, but it works well in practice and in theory. Forgy seems to be better than random partition, though there are some variants of k-means for which random partition is better. For the best results, you should run k-means multiple times with random starts.

An equivalent objective function is to minimize the *within-cluster variation*:

$$\sum_{i=1}^{k} \frac{1}{n_i} \sum_{y_i=i} \sum_{y_m=i} \left\| \mathbf{X}_j - \mathbf{X}_m \right\|^2.$$

At the minimizer, it turns out that this objective function is equal to twice the previous one. The nice thing about this expression is that it doesn't include the means; it's a function purely of the input points and the clusters we assign them to.

Should we normalize the data before applying k-means? It's the same as with PCA; sometimes yes, sometimes no. If some features are much larger than others, they will tend to dominate the Euclidean distance; this means that if you have features with different units of measurement, you should probably normalize them. Otherwise, if you have features in the same unit of measurement, you usually shouldn't, but it depends on the context.

21.2.2 k-medoids Clustering

k-medoids clustering generalizes k-means beyond the typical Euclidean distance; means aren't optimal for other distance metrics.

Here, we specify a distance function $d(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}})$ between points $\vec{\boldsymbol{x}}$, $\vec{\boldsymbol{y}}$, also known as *dissimilarity*. This distance function can be arbitrary, and ideally satisfies the triangle inequality: $d(\vec{\boldsymbol{x}}, \vec{\boldsymbol{y}}) \leq d(\vec{\boldsymbol{x}}, \vec{\boldsymbol{z}}) + d(\vec{\boldsymbol{z}}, \vec{\boldsymbol{y}})$.

Sometimes people use the ℓ_1 norm or the ℓ_∞ norm, and sometimes people specify a matrix of pairwise distances between the input points.

An example of a situation where Euclidean distance is not a good measure of dissimilarity is with market analysis, where we want to cluster customers who buy similar products. Here, if we use Euclidean distance, we'd get a big cluster of all the customers who have only ever bought one thing; instead, it makes more sense to treat each customer as a vector and measure the *angle* between two customers. If there's a large angle between customers, they're dissimilar.

We generalize the means with Euclidean distance with the *medoid*, i.e. the sample point that minimizes the total distance to other points in the same cluster. This means that the medoid of a cluster is *always* one of the input points.

21.2.3 Hierarchical Clustering

One difficulty with k-means (or k-medoids) is that we have to choose the number of clusters (k) before starting, and there isn't any reliable way of guessing how many clusters will best fit the data (though there are methodologies behind choosing a good k).

Hierarchical clustering has an advantage in this respect; here, we create a tree, and every subtree is a cluster. This means that some clusters contain smaller clusters.

There are two ways of constructing this tree; bottom-up and top-down. Bottom-up is *agglomerative clustering*; we start with each point as its own cluster, and repeatedly fuse pairs together. Top-down is *divisive clustering*; we start with all points in one cluster, and repeatedly split it up.

When the input is a point set, agglomerative clustering is used much more in practice; on the other hand, when the input is a graph, divisive clustering is more common.

To perform hierarchical clustering, we need a distance function for two clusters A and B. There are a few distance functions (*linkages*) commonly used:

- Complete linkage:
$$d(A,B) = \max d(\vec{\boldsymbol{w}}, \vec{\boldsymbol{x}}) : \vec{\boldsymbol{w}} \in A, \vec{\boldsymbol{x}} \in B.$$
- Single linkage:
$$d(A,B) = \min d(\vec{\boldsymbol{w}}, \vec{\boldsymbol{x}}) : \vec{\boldsymbol{w}} \in A, \vec{\boldsymbol{x}} \in B.$$
- Average linkage:
$$d(A,B) = \frac{1}{|A||B|} \sum_{\vec{\boldsymbol{w}} \in A} \sum_{\vec{\boldsymbol{x}} \in B} d(\vec{\boldsymbol{w}}, \vec{\boldsymbol{x}}).$$
- Centroid linkage:
$$d(A,B) = d(\vec{\boldsymbol{\mu}}_A, \vec{\boldsymbol{\mu}}_B),$$
where here $\vec{\boldsymbol{\mu}}_S$ is the mean of S.

The first three of these linkages work for any distance function, even if the input is just a matrix between all pairs of points. The centroid linkage really only makes sense if we're using the Euclidean distance, though there's a version of it that uses medoids instead of means (it turns out that medoids are also more robust to outliers than means).

The greedy agglomerative algorithm repeatedly fuses the two clusters that minimizes $d(A,B)$; this naively takes $O(n^3)$ time. However, for complete and single linkage, there are more sophisticated algorithms called CLINK and SLINK, which run in $O(n^2)$ time.

21.2.4 Dendrograms

A *dendrogram* is an illustration of the cluster hierarchy (tree) in which the vertical axis encodes all of the linkage distances. The horizontal axis is arbitrary, and has no meaning.

To cut a dendrogram into clusters, we draw a horizontal line according to the choice of clusters or intercluster distance. This is visualized in Fig. 21.2.

Figure 21.3 shows the various dendrograms produced with different linkage functions. Notably, complete linkage gives the best balanced dendrogram, whereas single linkage gives a very unbalanced dendrogram that is sensitive to outliers (especially near the top).

As such, single linkage is perhaps the worst; it tends to give a very unbalanced tree. In particular, if you cut the example in Fig. 21.3c into three clusters, notice that two of them have only one sample point.

The complete linkage tends to be the best balanced, as shown with the example in Fig. 21.3b, because when a cluster gets large, the furthest point in the cluster is always far away. This means that large clusters are more resistant to growth than small ones; if balanced clusters are your goal, this is your best choice. In most applications, you'll probably want the average or complete linkage.

One warning here is that centroid linkage can cause *inversions*, where a parent cluster is fused at a lower height than its children. As such, statisticians don't like it, but centroid linkage is popular in genetics.

As one final note, all the clustering algorithms we've studied so far are unstable—deleting a few input points can sometimes give very different results. However, these unstable heuristics are still the most commonly used clustering algorithms; it's unclear whether a truly stable clustering algorithm is even possible.

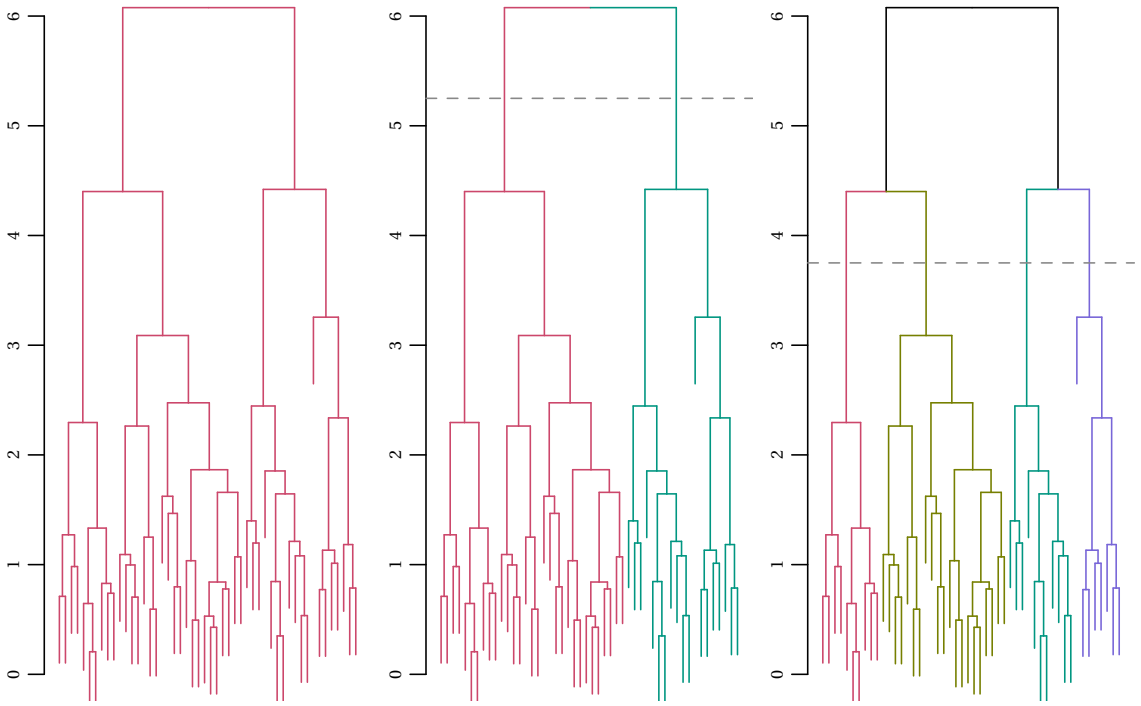


Figure 21.2: Partitioning a dendrogram into clusters

4/18/2022

Lecture 22

High Dimensional Spaces, Random Projection, Pseudoinverse

22.1 Geometry of High Dimensional Spaces

Consider a random point $\vec{\boldsymbol{p}} \sim \mathcal{N}(0, \mathbf{I}) \in \mathbb{R}^d$. What is the distribution of its length?

With the one-dimensional normal distribution, we'd expect it to be very common that the length is close to zero, and less common the farther we are from zero. However, in high dimensions, this intuition is completely wrong.

If the dimension is high, it turns out that the vast majority of the random points are at approximately the same distance from the mean; they lie in a thin shell. To see why, let us study the square of the distance:

$$\left\|\vec{\boldsymbol{p}}\right\|^2 = p_1^2 + p_2^2 + \cdots + p_d^2.$$

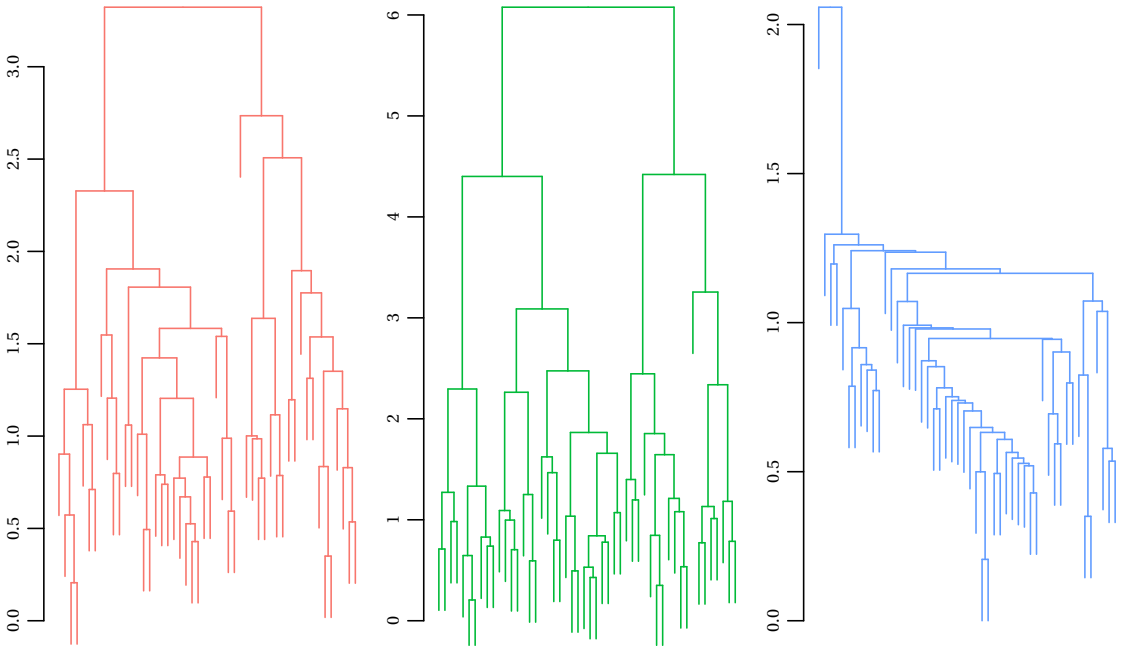
Here, each component p_i is sampled independently from a univariate standard normal distribution. The square of a component, p_i^2, then comes from a *chi-squared distribution*:

$$p_i \sim \mathcal{N}(0,1) \qquad p_i^2 \sim \chi^2(1) \qquad \mathbb{E}\left[p_i^2\right] = 1 \qquad \operatorname{Var}(p_i^2) = 2$$

This means that we have

$$\begin{aligned}
\mathbb{E}\left[\left\|\vec{\boldsymbol{p}}\right\|^2\right] &= d\,\mathbb{E}\left[p_1^2\right] = d \\
\operatorname{Var}(\left\|\vec{\boldsymbol{p}}\right\|^2) &= d\operatorname{Var}(p_1^2) = 2d \\
\sigma(\left\|\vec{\boldsymbol{p}}\right\|^2) &= \sqrt{2d}
\end{aligned}$$

Looking at these statistics, for large d, $\left\|\vec{\boldsymbol{p}}\right\|$ is concentrated in a thin shell with radius $\approx \sqrt{d}$ with a thickness proportional to $\sqrt[4]{2d}$. Here, the mean value of $\left\|\vec{\boldsymbol{p}}\right\|$ isn't exactly $\sqrt{d}$, but it is close, because the mean of $\left\|\vec{\boldsymbol{p}}\right\|^2$ is d and the standard deviation is much smaller. Likewise, the standard deviation isn't exactly $\sqrt[4]{2d}$, but it's close.



(a) Average Linkage **(b)** Complete Linkage **(c)** Single Linkage

Figure 21.3: Comparison between dendrograms of various linkage functions

What about a uniform distribution? Consider two concentric spheres of radii r and $r-\varepsilon$. The volume of the outer ball is proportional to r^d, and the volume of the inner ball is proportional to $(r-\varepsilon)^d$.

The ratio of the two volumes is

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

This quantity is small for large d; this means that most of the volume comes from a very slim shell of the outer sphere. For example, if $\frac{\varepsilon}{r} = 0.1$ and $d = 100$, then the inner ball has $0.9^{100} \approx 0.0027\%$ of the volume.

Together, we've shown that random points from a uniform distribution and from a Gaussian distribution in high dimensions will almost always fall in some outer shell.

This means that in high dimensions, sometimes the nearest neighbor and the 1000th-nearest neighbor don't differ by much; further this means that, k-means clustering and nearest neighbor classifiers are less effective for large d.

22.1.1 Angles between Random Vectors

What is the angle θ between a random $\vec{\boldsymbol{p}} \in \mathcal{N}(0, \mathbf{I}) \in \mathbb{R}^d$ and an arbitrary $\vec{\boldsymbol{q}} \in \mathbb{R}^d$?

Without loss of generality, suppose $\vec{\boldsymbol{q}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^T$. The value of $\vec{\boldsymbol{q}}$ doesn't really matter here, because the direction that $\vec{\boldsymbol{p}}$ points in is uniformly distributed overall possible directions.

From earlier this semester, the angle θ can be described with

$$\cos\theta = \frac{\vec{\boldsymbol{p}} \cdot \vec{\boldsymbol{q}}}{\|\vec{\boldsymbol{p}}\| \|\vec{\boldsymbol{q}}\|} = \frac{p_1}{\vec{\boldsymbol{p}}}.$$

This means that $\mathbb{E}[\cos\theta] \approx \frac{1}{\sqrt{d}}$.

If d is large, $\cos\theta$ is almost always very close to zero; this suggests that θ is almost always very close to $90°$.

Intuitively, this means that in high dimensional spaces, two random vectors are almost always very close to orthogonal. To put it another way, an arbitrary vector is almost orthogonal to the vast majority of all other vectors.

22.2 Random Projection

An alternative to PCA as preprocessing for clustering, classification, and/or regression is *random projection*. This process approximately preserves distances between points.

Here, we project onto a random subspace instead of the PCA subspace, but sometimes this preserves distances better than PCA. It works best when you project a very high-dimensional space into a medium-dimensional space. Since it roughly preserves the distances, algorithms like k-means clustering and nearest neighbor classification will give similar results to what they would give in high dimensions, but they run much faster.

to perform random projection, we pick a small ε, a small δ, and a random subspace $S \subset \mathbb{R}^d$ of dimension k, where

$$k = \left\lceil \frac{2\ln\left(\frac{1}{\delta}\right)}{\frac{\varepsilon^2}{2} - \frac{\varepsilon^3}{3}} \right\rceil.$$

For any point $\vec{\boldsymbol{q}}$, we then let $\hat{\vec{\boldsymbol{q}}}$ be the orthogonal projection of $\vec{\boldsymbol{q}}$ onto S, multiplied by $\sqrt{\frac{d}{k}}$. This multiplication helps preserve the distances between points after your project.

Theorem 22.1: Johnson-Lindenstrauss Lemma

For any two points $\vec{\boldsymbol{q}}, \vec{\boldsymbol{w}} \in \mathbb{R}^d$,

$$(1-\varepsilon)\left\|\vec{\boldsymbol{q}} - \vec{\boldsymbol{w}}\right\|^2 \leq \left\|\hat{\vec{\boldsymbol{q}}} - \hat{\vec{\boldsymbol{w}}}\right\|^2 \leq (1+\varepsilon)\left\|\vec{\boldsymbol{q}} - \vec{\boldsymbol{w}}\right\|^2,$$

with probability at least $1 - 2\delta$.

Typical values of ε are in $[0.02, 0.5]$, and typical values of δ are in $[\frac{1}{n^3}, 0.05]$. These values should be chosen according to your needs though.

With these ranges, the squared distance between two points after projecting might change by 2% to 50%. In practice, you can experiment with k to find the best speed-accuracy tradeoff; if you want all inter-sample-point distances to be accurate, you should set δ smaller than $\frac{1}{n^2}$, so you need a subspace of dimension $\Theta(\log n)$.

Reducing δ doesn't cost much, but reducing ε costs more. With random projection, you can bring 1 000 000 sample points down to a 10 000-dimensional space with at most a 6% error in the distances. What is remarkable is that the dimension d of the input points does not matter!

Why does this work? A random projection of $\vec{\boldsymbol{q}} - \vec{\boldsymbol{w}}$ is like taking a random vector and selecting k components. The mean of squares of those k components approximates the mean for the whole population.

How do we get a uniformly distributed random projection direction? We can choose each component from a univariate Gaussian distribution, then normalize the vector to unit length.

How do we get a random subspace? We can choose k random directions,then use Gram–Schmidt orthogonalization to make them mutually orthonormal. Interestingly, Indyk and Motwani show that if you skip the expensive normalization and Gram–Schmidt steps, random projection still works almost as well, because random vectors in a high-dimensional space are nearly equal in length and nearly orthogonal to each other with high probability.

22.3 The Pseudoinverse and the SVD

As of now, we're done with unsupervised learning. For the rest of the semester, we'll go back to supervised learning.

The singular value decomposition can give us insight into the pseudoinverse and its use in least-squares linear regression. Let's understand the pseudoinverse of a diagonal matrix first, and then we'll look at the pseudoinverse of a general matrix.

Suppose $\mathbf{D}$ is a (not necessarily square) diagonal $n \times d$ matrix. We find its pseudoinverse $\mathbf{D}^\dagger$ by transposing $\mathbf{D}$ and replacing every nonzero entry with its reciprocal.

This means that we have

$$\mathbf{D} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{D}^\dagger = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix} \qquad \mathbf{D}\mathbf{D}^\dagger = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{D}^\dagger\mathbf{D} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If $\mathbf{D}$ were a square matrix with no zeroes on the diagonal, then $\mathbf{D}^\dagger$ would be the inverse of $\mathbf{D}$, and $\mathbf{D}\mathbf{D}^\dagger$ and $\mathbf{D}^\dagger\mathbf{D}$ would be the identity matrix.

In general, $\mathbf{D}\mathbf{D}^\dagger$ and $\mathbf{D}^\dagger\mathbf{D}$ are always diagonal matrices with 0's and 1's only.

Observe that $\mathbf{D}\mathbf{D}^\dagger\mathbf{D} = \mathbf{D}$ and $\mathbf{D}^\dagger\mathbf{D}\mathbf{D}^\dagger$ and $\mathbf{D}^2\mathbf{D}^\dagger = \mathbf{D}$.

This is because the 0's and 1's in $\mathbf{D}\mathbf{D}^\dagger$ line up with the 0's and nonzeros in $\mathbf{D}$ and $\mathbf{D}^\dagger$. This is as close to an "inverse" as a rank-deficient matrix can get.

Now let's consider the pseudoinverse of an arbitrary matrix. Let $\mathbf{X}$ be any $n \times d$ matrix. Let $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ be its SVD. Recall that $\operatorname{rank}\mathbf{D} = \operatorname{rank}\mathbf{X}$.

The *Moore-Penrose pseudoinverse* of $\mathbf{X}$ is $\mathbf{X}^\dagger = \mathbf{V}\mathbf{D}^\dagger\mathbf{U}^T$.

Observe the following properties:

1. We have
$$\mathbf{X}\mathbf{X}^\dagger = \mathbf{U}\mathbf{D}\mathbf{V}^T\mathbf{V}\mathbf{D}^\dagger\mathbf{U}^T = \mathbf{U}\Big(\mathbf{D}\mathbf{D}^\dagger\Big)\mathbf{U}^T,$$
which is symmetric and positive semidefinite.
2. We have
$$\mathbf{X}^\dagger\mathbf{X} = \mathbf{V}\mathbf{D}^\dagger\mathbf{U}^T\mathbf{U}\mathbf{D}\mathbf{V}^T = \mathbf{V}\Big(\mathbf{D}^\dagger\mathbf{D}\Big)\mathbf{V}^T,$$
which is symmetric and positive semidefinite as well.
3. All of the following have the same rank: $\mathbf{D}$, $\mathbf{D}^\dagger$, $\mathbf{D}\mathbf{D}^\dagger$, $\mathbf{D}^\dagger\mathbf{D}$, $\mathbf{X}$, $\mathbf{X}^\dagger$, $\mathbf{X}\mathbf{X}^\dagger$, $\mathbf{X}^\dagger\mathbf{X}$.
4. If $\mathbf{X}$ has rank n, then $\mathbf{X}\mathbf{X}^\dagger = \mathbf{I}_{n\times n}$ and $\mathbf{X}^\dagger$ is a *right inverse*.
5. If $\mathbf{X}$ has rank d, then $\mathbf{X}^\dagger\mathbf{X} = \mathbf{I}_{d\times d}$ and $\mathbf{X}^\dagger$ is a *left inverse*.
6. $\mathbf{X}\mathbf{X}^\dagger\mathbf{X} = \mathbf{X}$. To see this, we have
$$\mathbf{X}\mathbf{X}^\dagger\mathbf{X} = \mathbf{U}\Big(\mathbf{D}\mathbf{D}^\dagger\Big)\mathbf{U}^{\mathbf{U}}\mathbf{D}\mathbf{V}^T = \mathbf{U}\Big(\mathbf{D}\mathbf{D}^\dagger\mathbf{D}\Big)\mathbf{V}^T = \mathbf{U}\mathbf{D}\mathbf{V}^T = \mathbf{X}.$$
7. $\mathbf{X}^\dagger\mathbf{X}\mathbf{X}^\dagger = \mathbf{X}^\dagger$. This proof is symmetric to the previous one.

We can also show that the pseudoinverse always gives a good solution in least-squares linear regression, even when $\mathbf{X}^T\mathbf{X}$ is singular.

To see this, we just need to show that $\vec{\boldsymbol{w}} = \mathbf{X}^\dagger\vec{\boldsymbol{y}}$ is a solution to the normal equations $\mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} = \mathbf{X}^T\vec{\boldsymbol{y}}$.

$$\mathbf{X}^T\mathbf{X}\vec{\boldsymbol{w}} = \mathbf{X}^T\mathbf{X}\mathbf{X}^\dagger\vec{\boldsymbol{y}} = \mathbf{V}\mathbf{D}\mathbf{U}^T\mathbf{U}\Big(\mathbf{D}\mathbf{D}^\dagger\Big)\mathbf{U}^T\vec{\boldsymbol{y}} = \mathbf{V}\Big(\mathbf{D}^2\mathbf{D}^\dagger\Big)\mathbf{U}^T\vec{\boldsymbol{y}} = \mathbf{V}\mathbf{D}\mathbf{U}^T\vec{\boldsymbol{y}} = \mathbf{X}^T\vec{\boldsymbol{y}}.$$

If the normal equations have multiple solutions, $\vec{\boldsymbol{w}} = \mathbf{X}^\dagger\vec{\boldsymbol{y}}$ is the *least-norm solution*. That is, it minimizes $\|\vec{\boldsymbol{w}}\|$ among all solutions.

This way of solving the normal equations is very helpful when $\mathbf{X}^T\mathbf{X}$ is singular, because $n < d$ or the sample points lie on a subspace of the feature space. However, if $\mathbf{X}$ has a very small singular value, the reciprocal of that singular value will be very large and have a very large effect on $\vec{\boldsymbol{w}}$; when that singular value is exactly zero, it has no effect on $\vec{\boldsymbol{w}}$.

This provokes the question: if we have a very tiny singular value, should we pretend it is zero? Ridge regression implements this policy to some degree.

4/20/2022

Lecture 23

Learning Theory

One thing humans do well is generalize; learning theory tries to explain how machine learning algorithms generalize, so they can classify data they've never seen before. It also tries to derive mathematically how much training data we need to generalize well.

Learning theory starts with the observation that if we want to generalize, we must constrain what hypotheses we allow our learner to consider.

A *range space* (also known as a *set system*) is a pair (P, H), where

- P is the set of all possible test/training points (this can be infinite)
- H is the *hypothesis class*, i.e. a set of *hypotheses* (also known as *ranges*, or *classifiers*)

  Each hypothesis is a subset $h \subseteq P$ that specifies which points hypothesis h predicts are in class C. This means that each hypothesis h is a 2-class classifier, and H is ta set of sets of points.

Here are a few examples:

- Power set classifier: P is a set of k numbers, H is the *power set* of P, containing all 2^k subsets of P. For example, $P = \{1,2\}$ and $H = \{\varnothing, \{1\}, \{2\}, \{1,2\}\}$.
- Linear classifier: $P = \mathbb{R}^d$, and H is the set of all *halfspaces*; each half space has the form $\{\vec{\boldsymbol{x}} : \vec{\boldsymbol{w}} \cdot \vec{\boldsymbol{x}} \geq -\alpha\}$.

  In this example, both P and H are infinite; in particular, H contains every possible halfspace, i.e. every possible linear classifier in d dimensions.

The power set classifier sounds very powerful, because it can learn every possible hypothesis. However, the reality is that this classifier can't generalize at all. Imagine that we have three training points and three test points in a row:

? C C ? N ?

The power set classifier can classify these three test points in any way it likes; unfortunately, that means it has learned nothing about the test points from the training points.

In contrast, the linear classifier can learn only two hypotheses that fit this training data. The leftmost test point must be classified as class C, and the rightmost test point must be classified as class Not-C (i.e. N). Only the test point in the middle can swing either way.

This means that the linear classifier has a big advantage: it can generalize from a few training points. This is also a big disadvantage if the data isn't close to linearly separable, but that's another issue for another time.

Now, we will investigate how well the training error predicts the test error, and how that differs for these two classifiers.

Suppose all of the training and test points are drawn independently from the same probability distribution $\mathcal{D}$ defined on the domain P. Here, $\mathcal{D}$ also determines each point's label; classes C and N may have overlapping distributions.

Let $h \in H$ be a hypothesis (i.e. a classifier). Here, h predicts a point x is in class C if $x \in h$.

Definition 23.1: Risk

The *risk*, or the *generalization error* $R(h)$ of h is the probability that h misclassifies a random point x drawn from $\mathcal{D}$. That is, the probability that $x \in C$ but $x \notin h$ or vice versa.

Risk is almost the same as test error. To be precise, the risk is the average test error for test points drawn randomly from $\mathcal{D}$. For a particular test set, sometimes the test error is higher or lower, but on average it is $R(h)$. If you have an infinite amount of test data, the risk and the test error would be the same.

Definition 23.2: Empirical Risk

Let $X \subseteq P$ be a set of n training points drawn from $\mathcal{D}$. The *empirical risk*, or the *training error* $\hat{R}(h)$ is the percent of X misclassified by h.

This matches the definition of empirical risk from Definition 12.2 if you use the 0-1 loss function.

h misclassifies each training point with probability $R(h)$, so the total number of misclassified training points follows a binomial distribution. As $n \to \infty$, $\hat{R}(h)$ better approximates $R(h)$.

This can be seen in Fig. 23.1; with $n = 20$, we can get lucky and have a very low training error, whereas with $n = 500$ this is a lot less likely.

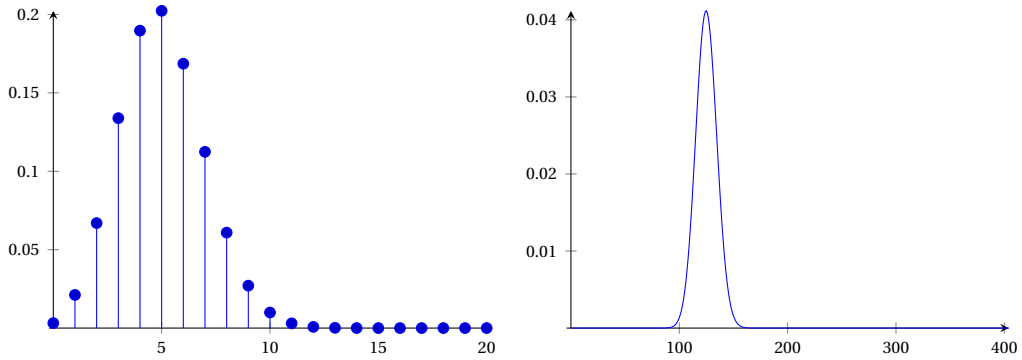


Figure 23.1: Comparison between binomial distributions with $n = 20$ and $n = 500$, both with $p = 0.25$

If we had infinite training data, this distribution would become infinitely narrow, and the training error would always be equal to the risk. However, we can't have infinite training data, so how well does the training error approximate the risk?

Hoeffding's inequality gives us the probability of a bad estimate:

$$\mathbb{P}\left(\left|\hat{R}(h) - R(h)\right| > \varepsilon\right) \leq 2e^{-2\varepsilon^2 n}.$$

This tells us that if n is big enough, it's very unlikely for a binomial random variable to be far from its mean.

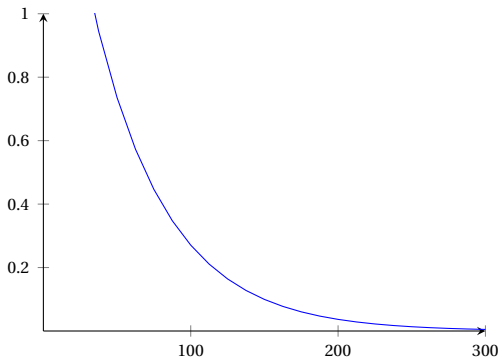


Figure 23.2: Hoeffding's bound for $\varepsilon = 0.1$, plotting how n affects $\mathbb{P}\left(\left|\hat{R}(h) - R(h)\right| > \varepsilon\right)$

As one can see in Fig. 23.2, it takes at least 200 training points to have high confidence of attaining the (rather unambitious) error bound of $\varepsilon = 0.1$.

One idea for a learning algorithm is to choose the $\hat{h} \in H$ that minimizes $\hat{R}(\hat{h})$; this is called *empirical risk minimization*.

None of the classification algorithms we've studied actually do this; mainly because it's computationally infeasible to pick the best hypothesis. SVMs can find a linear classifier with zero training error when the training data is linearly separable, but when it isn't, SVMs try to find a linear classifier with low training error, but don't necessarily find the one with *minimum* training error (it's NP-hard to do so).

If we nevertheless pretend that we do have the computational power to try every hypothesis and pick the one with lowest training error, we run into a problem. If there are too many hypotheses, there could be some h with high $R(h)$ that gets lucky and has a very low $\hat{R}(h)$.

This brings us to the central idea of learning theory. It is perhaps natural to think that the ideal learning algorithm would have the largest class of hypotheses, so it can find the perfect one to fit the data. However, in reality, you can have so many hypotheses that some of them just get lucky and score far lower training error than their actual risk. This is another way of understanding what "overfitting" is.

23.1 Dichotomies

More precisely, the issue is not that we have too many *hypotheses*; usually we have infinitely many hypotheses, and it's okay. The problem is that we can have too many *dichotomies*.

Definition 23.3: Dichotomy

A *dichotomy* of X is $X \cap h$, where $h \in H$.

A dichotomy picks out the training points that h predicts are in class C. We can think of each dichotomy as a function assigning each training point to class C or N.

For example, with a linear classifier, we could have dichotomies CCN, or CCC, but not CNC.

For n training points, there could be up to 2^n dichotomies. The more dichotomies there are, the more likely it is that one of them will get lucky and have misleadingly low empirical risk.

In an extreme case, if H allows all 2^n dichotomies, then there exists some $\hat{h}$ such that $\hat{R}(\hat{h}) = 0$ even if every $h \in H$ has high risk. Specifically, if our hypothesis permits all 2^n possible assignments of the n training points to classes, then one of them will have zero training error. However, this is always true regardless of the actual performance of the hypotheses; since the hypothesis class imposes no structure, we overfit the training points.

Given Π dichotomies, we have

$$\mathbb{P}\left(\text{at least one dichotomy has } \left|\hat{R} - R\right| > \varepsilon\right) \le 2\Pi e^{-2\varepsilon^2 m} = \delta.$$

If we fix δ and solve for ε, we have with probability $\ge 1 - \delta$ that for every $h \in H$,

$$\left|\hat{R}(h) - R(h)\right| \le \sqrt{\frac{1}{2n} \ln \frac{2\Pi}{\delta}} = \varepsilon.$$

This tells us that the smaller we make Π (the number of possible dichotomies), and the larger we make n (the number of training points), the more accurately the training error will approximate how well the classifier performs on test data.

Small Π means that we're less likely to overfit; we have less variance but more bias. This doesn't necessarily mean that the risk will be small though; if our hypothesis class H doesn't fit the data well, both training and test error would be large. Ideally, we'd want a hypothesis class that fits the data well, and yet doesn't have many hypotheses.

Suppose we define $h^* \in H$ be the minimizer of $R(h^*)$. This is the "best" classifier.

Remember we picked the classifier $\hat{h}$ that minimizes the empirical risk; we really want the classifier h^* that minimizes the *actual* risk, but we can't know what h^* is. However, if Π is small and n is large, the hypothesis $\hat{h}$ is probably nearly as good as h^*.

Quantifying this, we can say with probability $\geq 1-\delta$ that our chosen $\hat{h}$ has nearly optimal risk:

$$R(\hat{h}) \leq \hat{R}(\hat{h}) + \varepsilon \leq \hat{R}(h^*) + \varepsilon \leq R(h^*) + 2\varepsilon,$$

where

$$\varepsilon = \sqrt{\frac{1}{2n} \ln \frac{2\Pi}{\delta}}.$$

This means that with enough training data and a limit on the number of dichotomies, empirical risk minimization usually chooses a classifier close to the best one in the hypothesis class.

For a given δ and ε, the *sample complexity* is the number of training points needed to achieve this ε with probability $\geq 1-\delta$:

$$n \geq \frac{1}{2\varepsilon^2} \ln \frac{2\Pi}{\delta}.$$

If Π is small, then we won't need too many training points to choose a good classifier. Unfortunately, if $\Pi = 2^n$, the inequality then says that n has to be bigger than n. This means that the power set classifier can't learn much or generalize at all. We'd need to severely reduce Π; one way to do this is to use a linear classifier.

23.2 The Shatter Function and Linear Classifiers

We define $\Pi_H(X)$ to count the number of dichotomies in X:

$$\Pi_H(X) = |\{X \cap h : h \in H\}|.$$

The *shatter function* is defined as

$$\Pi_H(n) = \max_{|X|=n,\ X \subseteq P} \Pi_H(X).$$

That is, the most dichotomies out of any point set of size n.

For example, suppose we consider linear classifiers in a plane with H as the set of all half-planes.

With three training points, $\Pi_H(3) = 8$. This means that linear classifiers can induce all 8 dichotomies of three points in a plane.

With four training points, however, $\Pi_H(4) = 14$. This means that no four points can ever have 16 dichotomies; the full standard proof uses a result called Radon's Theorem.

Theorem 23.4

For all range spaces, either $\Pi_H(n)$ is a polynomial in n, or $\Pi_H(n) = 2^n$ for all $n \geq 0$.

This is a surprising fact with deep implications. Suppose you have m points, with some training points and some test points. It is then either the case that a range space permits every possible dichotomy of the points, and the training points don't help classify the test points at all; or the range space permits only a polynomial subset of the 2^m possible dichotomies, so once you have labelled the training points, you've cut down the number of ways you can classify the test points dramatically.

No shatter function ever occupies the no-man's land between polynomial and 2^m.

For linear classifiers, we can do slightly better; we always know exactly how many dichotomies there can be.

Theorem 23.5: Cover's Theorem

Linear classifiers in $\mathbb{R}^d$ allow up to

$$\Pi_H(n) = 2\sum_{i=0}^{d} \binom{n-1}{i}$$

dichotomies of n points.

Specifically:

- For $n \le d+1$,
$$\Pi_H(n) = 2^n.$$
- For $n \ge d+1$,
$$\Pi_H(n) \le 2\left(\frac{e(n-1)}{d}\right)^d.$$
Note that this is polynomial in d, with exponent d.
This also means that the sample complexity needed to achieve $R(\hat{h}) \le \hat{R}(\hat{h}) + \varepsilon \le R(h^*) + 2\varepsilon$ with probability $\ge 1-\delta$ satisfies
$$n \ge \frac{1}{2\varepsilon^2}\left(d \ln \frac{n-1}{d} + d + \ln \frac{4}{\delta}\right).$$

A corollary is that linear classifiers need only $n \in O(d)$ training points for training error to accurately predict risk or test error.

On the other hand, with a classifier with 2^n possible dichotomies, no amount of training data will ever guarantee that the training error of the hypothesis we choose approximates the true risk.

23.3 VC Dimension

Definition 23.6: VC Dimension

The *Vapnick–Chervonenkis dimension* of (P, H) is
$$\mathrm{VC}(h) = \max\{n : \Pi_H(n) = 2^n\}.$$
Note that this quantity can be ∞.

We say that H *shatters* a set X of n points if $\Pi_H(X) = 2^n$. $\mathrm{VC}(H)$ is the size of the largest X that H can shatter. Such an X is a point set for which all 2^n dichotomies are possible.

The VC dimension is motivated by an observation that sometimes makes it easy to bound the polynomial if the shatter function isn't 2^n for all n.

Theorem 23.7

We have
$$\Pi_H(n) \le \sum_{i=0}^{\mathrm{VC}(H)} \binom{n}{i}.$$
This means that for $n \ge \mathrm{VC}(H)$, then
$$\Pi_H(n) \le \left(\frac{en}{\mathrm{VC}(H)}\right)^{\mathrm{VC}(H)}.$$

This means that the VC dimension is an upper bound on the exponent of the polynomial.

A corollary is that $O(\mathrm{VC}(H))$ training points suffices for accuracy, though the hidden constant is big.

For example, with linear classifiers in a plane, we know that $\Pi_H(3) = 8$, but $\Pi_H(4) = 14$.

This means that $\mathrm{VC}(H) = 3$, $\Pi_H(n) \le \frac{e^3}{27} n^3$, and we have $O(1)$ sample complexity.

The VC dimension isn't always the tightest bound, but that's not a big deal—the sample complexity and the accuracy bound are both based on the logarithm of the shatter function; getting the exponent wrong only changes a constant in the sample complexity.

The important thing is simply to show that there is some polynomial bound on the shatter function at all; the VC dimension is not the only way to do that, but it's often the easiest.

The main point here is that to get generalization, you need to limit the expressiveness of your hypothesis class, so that you limit the number of possible dichotomies of a point set. This may or may not increase the bias, but no limiting the number of dichotomies results in very bad overfitting.

4/25/2022

Lecture 24

Boosting, Nearest Neighbor Classification

24.1 AdaBoost

AdaBoost ("adaptive boosting") is an ensemble method for classification (or regression) that trains multiple learners on *weighted* sample points (much like bagging). It uses different weights for each learner, increases weights of misclassified training points, and gives bigger votes to more accurate learners.

The input is an $n \times d$ design matrix $\mathbf{X}$, and a vector of labels $\vec{y} \in \mathbb{R}^n$ with $y_i = \pm 1$.

Here are the main ideas:

- We train T classifiers $G_1, \dots, G_T$ (here, T stands for "trees")
- The weight of sample point $\mathbf{X}_i$ in G_t grows according to how many of $G_1, \dots, G_{t-1}$ misclassified it.

  Further, if $\mathbf{X}_i$ is misclassified by very accurate learners, its weight grows even more. On the other hand, the weight shrinks every time $\mathbf{X}_i$ is correctly classified.
- Train G_t to try harder to correctly classify sample points with larger weights.
- The metalearner is a linear combination of the learners. That is, for a test point $\vec{z}$,

$$M(\vec{z}) = \sum_{t=1}^{T} \beta_t G_t(\vec{z}).$$

  Each G_t is ± 1, but M is continuous; at the very end, we'd return the sign of $M(\vec{z})$.

Boosting works with most learning algorithms, but it was originally developed for decision trees, and boosted decision trees are very popular and successful. To weight points in decision trees, we use a weighted entropy where instead of computing the proportion of points in each class, we instead compute the proportion of *weight* in each class.

In iteration T, what classifier G_T and coefficient β_T should we choose? We pick a loss function $L(\text{prediction}, \text{label})$, and we find G_T and β_T to minimize

$$\text{Risk} = \frac{1}{n} \sum_{i=1}^{n} L(M(\mathbf{X}_i), y_i) \qquad\qquad M(\mathbf{X}_i) = \sum_{t=1}^{T} \beta_t G_t(\mathbf{X}_i)$$

The AdaBoost metalearner uses the *exponential loss function*

$$L(\rho, \ell) = e^{-\rho\ell} = \begin{cases} e^{-\rho} & \ell = +1 \\ e^{\rho} & \ell = -1 \end{cases}$$

This loss function is for the metalearner only. The individual learners G_t usually use other loss functions, if they use a loss function at all.

It's important to note that the label ℓ is binary, G_t is binary, but $\rho = M(\mathbf{X}_i)$ is continuous.

The exponential loss function has the advantage that it pushes hard against badly misclassified points. That's one reason why it's usually better than the squared error loss function for classification in a metalearner. It's similar to why in neural networks we often prefer the cross-entropy loss function to the squared error.

If we plug in the loss function and simplify, we have

$$\begin{aligned} n \times \text{Risk} &= \sum_{i=1}^{n} L(M(\mathbf{X}_i), y_i) \\ &= \sum_{i=1}^{n} e^{-y_i M(\mathbf{X}_i)} \\ &= \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \beta_t G_t(\mathbf{X}_i)\right) \\ &= \sum_{i=1}^{n} \prod_{t=1}^{T} e^{-\beta_t y_i G_t(\mathbf{X}_i)} \\ &= \sum_{i=1}^{n} w_i^{(T)} e^{-\beta_T y_i G_T(\mathbf{X}_i)} \qquad \left(\text{where } w_i^{(T)} = \prod_{t=1}^{T-1} e^{-\beta_t y_i G_t(\mathbf{X}_i)}\right) \end{aligned}$$

If we split the summation into two parts, depending on whether any given point $\mathbf{X}_i$ is classified correctly (i.e. $y_i = G_T(\mathbf{X}_i)$), or classified incorrectly (i.e. $y_i \neq G_T(\mathbf{X}_i)$), we have

$$\begin{aligned} &= e^{-\beta_T} \sum_{y_i = G_T(\mathbf{X}_i)} w_i^{(T)} + e^{\beta_T} \sum_{y_i \neq G_T(\mathbf{X}_i)} w_i^{(T)} \\ &= e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + \left(e^{\beta_T} - e^{-\beta_T}\right) \sum_{y_i \neq G_T(\mathbf{X}_i)} w_i^{(T)} \end{aligned}$$

What G_T minimizes the risk? It's the same as the learner who minimizes the sum of $w_i^{(T)}$ over all misclassified points X_i.

This is an interesting result; we've discovered what weight we should assign to each sample point. If we want to minimize the risk, we should find the classifier that minimizes the total weight of the misclassified points for this weight function $w_i^{(T)}$. It's a complicated function, but we can compute it. The main observation is that we can express each learner's weights in terms of the previous learner's weights recursively:

$$w_i^{(T+1)} = w_i^{(T)} e^{-\beta_T y_i G_T(\mathbf{X}_i)} = \begin{cases} w_i^{(T)} e^{-\beta_T} & y_i = G_T(\mathbf{X}_i) \\ w_i^{(T)} e^{\beta_T} & y_i \neq G_T(\mathbf{X}_i) \end{cases}$$

This recursive formulation is a nice benefit of choosing the exponential loss function. Notice that a weight shrinks if the point was classified correctly by learner T, and grows if the point was misclassified.

It's not always possible to pick a learner that classifies all the training points correctly (ex. a linear classifier can't possibly classify data that is not linearly separable); it's also NP-hard to find the optimal linear classifier, so in practice G_T will be an approximate best learner, not the true minimizer of training error.

Now, let's derive the optimal value of β_T; we can set the gradient of risk (with respect to β_T) equal to zero and solve:

$$0 = -e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + \left(e^{\beta_T} + e^{-\beta_T}\right) \sum_{y_i \neq G_T(\mathbf{X}_i)} w_i^{(T)}$$

Dividing both sides by the first term, we have

$$\begin{aligned} &= -1 + \left(e^{2\beta_T} + 1\right) \text{err}_T \qquad \left(\text{where } \text{err}_T = \frac{\sum_{y_i \neq G_T(\mathbf{X}_i)} w_i^{(T)}}{\sum_{i=1}^{n} w_i^{(T)}}\right) \\ \beta_T &= \frac{1}{2} \ln\left(\frac{1 - \text{err}_T}{\text{err}_T}\right) \end{aligned}$$

This is the optimal metalearner. Looking a little closer, if $\text{err}_T = 0$, then $\beta_T = \infty$; a perfect learner gets an infinite vote. We also have if $\text{err}_T = \frac{1}{2}$, then $\beta_T = 0$; a learner with 50% weighted training accuracy gets no vote at all.

This means that more accurate learners get bigger votes in the metalearner. Interestingly, a learner with training accuracy worse than 50% gets a negative vote. This is because a learner with 40% accuracy is just as useful as a learner with 60% accuracy; the metalearner just reverses the sign of its votes.

Now, we can state the AdaBoost algorithm.

1. Initialize weights $w_i \leftarrow \frac{1}{n}$ for all $i \in [1, n]$
2. For all $t = 1$ to T:
   (a) Train G_t with weights w_i
   (b) Compute weighted error rate
   $$\text{err} \leftarrow \frac{\sum_{\text{misclassified}} w_i}{\sum_{\text{all}} w_i}$$
   and the coefficient
   $$\beta_T \leftarrow \ln\left(\frac{1-\text{err}}{\text{err}}\right).$$
   (c) Re-weight points:
   $$w_i \leftarrow w_i \cdot \begin{cases} e^{\beta_t} & G_t \text{ misclassifies } \mathbf{X}_i \\ e^{-\beta_t} & \text{otherwise} \end{cases} = w_i \cdot \begin{cases} \sqrt{\frac{1-\text{err}}{\text{err}}} & G_t \text{ misclassifies } \mathbf{X}_i \\ \sqrt{\frac{\text{err}}{1-\text{err}}} & \text{otherwise} \end{cases}$$
3. Return metalearner
   $$h(\vec{z}) = \text{sign}\left(\sum_{t=1}^{T} \beta_t G_t(\vec{z})\right).$$

An example of AdaBoost with two learners is shown in Fig. 24.1. The left plot is the original data, colored by label. The middle plot shows the first learner of AdaBoost, training a depth 1 decision tree on the data all with equal weights. The green line is the resulting decision boundary. AdaBoost then proceeds to re-weight the points, putting more weight on misclassified points; the size of the points are proportional to the weight given to the point. The right plot shows a second learner of AdaBoost, proceeding in a similar manner, except the decision tree is now trained with the previous weights in mind.

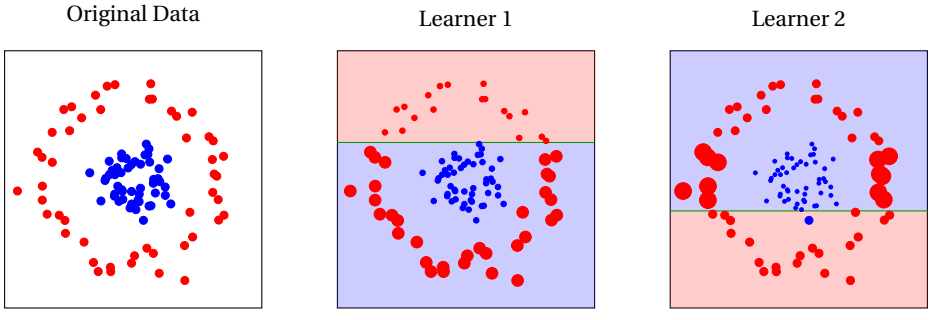


Figure 24.1: Example of AdaBoost with two learners

Why boost decision trees, as opposed to other learning algorithms? Why boost *short* trees?

- It's fast. We're training many learners, and running many learners at classification time too; short decision trees that only look at a few features are fast at both training and testing.
- No hyperparameter search is needed.
- Easy to make a tree beat 55% training accuracy (or any other threshold) consistently.

- Easy bias-variance control. Boosting can overfit, and as such AdaBoost trees are usually short to reduce overfitting.

  With more learners, the bias of AdaBoost decreases. Variance is more complicated though—it often decreases at first, since successive trees focus on different features; later, it often increases. Sometimes boosting overfits after many iterations, and sometimes it doesn't.

- AdaBoost and short trees is a form of subset selection.

  Features that don't improve the metalearner aren't used at all; this helps reduce overfitting and running time, especially if there are a lot of irrelevant features.

- Linear decision boundaries don't boost well.

  Boosting linear classifiers gives you an approximately linear classifier, so SVMs aren't a great choice to use. Methods with nonlinear decision boundaries benefit more from boosting, because they allow boosting to reduce the bias faster.

  Sometimes AdaBoost is used with depth-one decision trees, but it's not ideal, since depth-one decision trees are linear. Even depth-two decision trees boost substantially better.

Some more facts about AdaBoost:

- The posterior probability can be approximated as $\mathbb{P}(Y = 1 \mid \vec{x}) \approx \frac{1}{1+e^{-2M(\vec{x})}}$.
- Exponential loss if vulnerable to outliers; for corrupted data, it is often better to use another loss (though better loss functions usually have more complicated weight computations).
- If every learner beats accuracy μ for $\mu > 50\%$, then the metalearner training accuracy will eventually be 100%. (This result will be proven in homework.)

24.2 Nearest Neighbor Classification

With nearest neighbor classification, we are given a query point q, and we want to find the k sample points nearest to q. Here, we use the distance metric of your choice.

Regression can give us the average label of the k points, and classification can return the class with the most votes from the k points, or return a histogram of class probabilities (with limited precision); neither of these options are quite what we want.

Theorem 24.1: Cover and Hart (1967)

As $n \to \infty$, the 1-NN error rate is $< 2B - B^2$, where B is the Bayes risk.

If there are only two classes, then the 1-NN error rate is $\leq 2B - 2B^2$.

This is one of the theorems that shows that if we have a lot of data, then nearest neighbors can work quite well. There are a few technical requirements for this theorem though; the most important requirement see that the training points and test points must all be drawn independently from the same probability distribution. The theorem also applies to any separable metric space, so it's not just for the Euclidean metric.

Here's another theorem along the same lines, showing that k-NN can also work quite well with a lot of data.

Theorem 24.2: Fix and Hodges (1951)

As $n \to \infty$, $k \to \infty$, and $\frac{k}{n} \to 0$, the k-NN error rate converges to B, where B is the Bayes risk. This means that k-NN converges to be Bayes optimal.

4/27/2022

Lecture 25

Nearest Neighbor Algorithms: Voronoi Diagrams and k-d Trees

25.1 Exhaustive k-NN Algorithm

Given a query point q, the exhaustive k-N algorithm is as follows:

- Scan through all n sample points, and compute the (squared) distances to q.
- Maintain a max-heap with the k shortest distances seen so far. Whenever we encounter a sample point closer to q than the point at the top of the heap, you remove the top point and insert the better point.

Here, it should be clear that a heap isn't necessary if k is small, but with larger k, a heap will substantially speed up keeping track of the kth shortest distance.

The time to train this algorithm is 0; all of the computation is done during the query. This is the only $O(0)$ time algorithm we'll learn this semester.

The query time is $O(nd + n\log k)$, and for a random point order, the expected query time is $O(nd + k\log n\log k)$.

Notice that we can slightly improve the expected running time by randomizing the point order, so that only $O(k\log n)$ expected heap operations occur. In practice, though, it isn't recommended, as you'd probably lose more from cache misses than you'll gain from fewer heap operations.

Is there an algorithm to preprocess training points to obtain a sublinear query time?

- For 2–5 dimensions, Voronoi diagrams allow us to do this.
- For a medium dimension (up to ≈ 30), k-d trees are better.
- With larger dimensions, it turns out that exhaustive k-NN is best, but we can also use PCA or random projection.

  Locality sensitive hashing can also be used with large dimensions, but it's still under research and not widely adopted.

25.2 Voronoi Diagrams

Let X be a point set. The *Voronoi cell* of $\vec{\boldsymbol{w}} \in X$ is

$$\operatorname{Vor}(\vec{\boldsymbol{w}}) = \left\{\vec{\boldsymbol{p}} \in \mathbb{R}^d : (\forall \vec{\boldsymbol{v}} \in X)\left(\left\|\vec{\boldsymbol{p}} - \vec{\boldsymbol{w}}\right\| \le \left\|\vec{\boldsymbol{p}} - \vec{\boldsymbol{v}}\right\|\right)\right\}.$$

Here, notice that a Voronoi cell is always a convex polyhedron or polytope.

The *Voronoi diagram* of X is the set of X's Voronoi cells. An example is shown in Fig. 25.1.

The size of a Voronoi diagram (ex. the number of vertices) is $O(n^{\lceil d/2\rceil})$ (here, d is the dimension of the points). This upper bound is tight when d is a small constant. As d grows, the tightest asymptotic upper bound is somewhat smaller than this, but the complexity still grows exponentially with d.

However, in practice, the size is $O(n)$, though this leaves out a constant that may grow exponentially with d.

Utilizing Voronoi diagrams, given a query point $\vec{\boldsymbol{q}} \in \mathbb{R}^d$, we want to find the point $\vec{\boldsymbol{w}} \in X$ for which $\vec{\boldsymbol{q}} \in \operatorname{Vor}(\vec{\boldsymbol{w}})$. We'd need a second data structure that can perform this search on a Voronoi diagram efficiently.

In 2D, we take $O(n\log n)$ time to compute the Voronoi diagram and a *trapezoidal map* for the point location. With this, the query time is $O(\log n)$ because of the trapezoidal map. This is a great improvement from the linear query time of exhaustive search.

In d dimensions, we can use a *binary space partition tree* (BSP tree) for the point location. It's difficult to characterize the running time of this strategy, but it's likely to be reasonably fast in 3–5 dimensions.

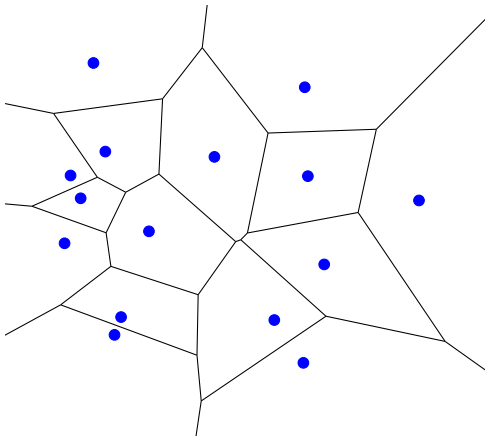


Figure 25.1: Example Voronoi diagram for 15 points

Note that Voronoi diagrams supports only 1-nearest neighbor queries. If we want the k-nearest neighbors, there is an order-k Voronoi diagram that has a cell for each possible k-nearest neighbors. However, nobody uses these: the size of an order-k Voronoi diagram is $O(k^2 n)$ in 2D, and gets worse in higher dimensions; there's also no software available to compute one.

There are also Voronoi diagrams for other distance metrics, like the ℓ_1 and ℓ_∞ norms.

Voronoi diagrams are good for 1-nearest neighbors in 2 or 3 dimensions, maybe 4 or 5, but k-d trees are much simpler and probably faster in 6 or more dimensions.

25.3 k-d Trees

k-d trees are essentially decision trees for nearest neighbor search. Here are some differences compared to decision trees:

- We choose the splitting feature with the greatest width; that is, feature i in $\max_{i,j,k}\left(X_{ji} - X_{ki}\right)$

  With nearest neighbor search, we don't care about the entropy. Instead, we'd like a sphere around the query point to not intersect very many boxes of the decision tree. This means that it helps if the boxes are nearly cubical, rather than long and thin.

  One cheap alternative is to rotate through the features; we split on the first feature at depth 1, the second feature at depth 2, and so on. This builds the tree faster, by a factor of $O(d)$.

- The splitting value is the median point for feature i, or $\frac{X_{ji}+X_{ki}}{2}$.

  The median guarantees $\lfloor \log_2 n \rfloor$ tree depth, and $O(nd \log n)$ tree building time.

  If we rotate through the features, this gives $O(n \log n)$ time. One alternative to the median is splitting at the box center, which improves the aspect ratios of the boxes, but it could make the tree unbalanced. A compromise strategy is to alternate between medians at odd depths and centers at even depths, which also guarantees an $O(\log n)$ depth.

- Each internal node of a k-d tree stores a sample point that lies in the node's box. This sample point is usually the splitting point.

  Some k-d tree implementations have points only at the leaves, but it's better to have points in internal nodes to, so when we search the tree, we often stop searching earlier.

With k-d trees, given a query point $\vec{\boldsymbol{q}}$, we want to find a sample point $\vec{\boldsymbol{w}}$ such that $\left\|\vec{\boldsymbol{q}} - \vec{\boldsymbol{w}}\right\| \le (1+\varepsilon)\left\|\vec{\boldsymbol{q}} - \vec{\boldsymbol{s}}\right\|$, where $\vec{\boldsymbol{s}}$ is the closest sample point.

If $\varepsilon = 0$, this is known as *exact nearest neighbors*, and with $\varepsilon > 0$, this is known as *approximate nearest neighbors*.

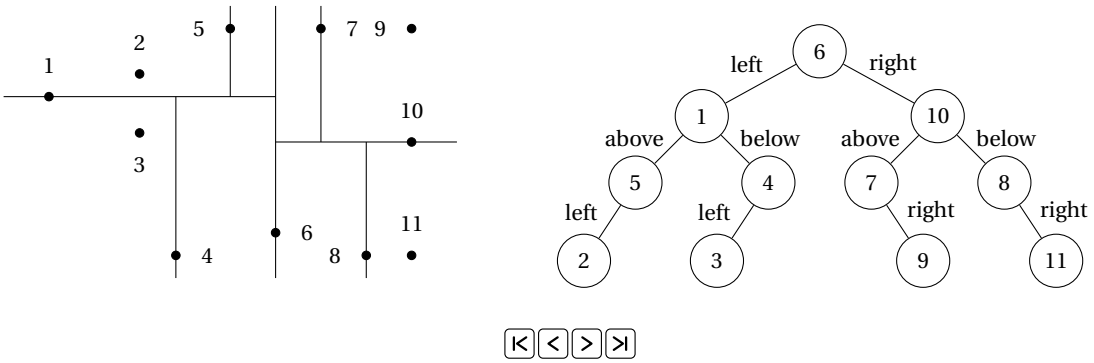


Figure 25.2: Example of a k-d tree. To step through the animation, you must use a PDF reader that supports animations, eg. Adobe Acrobat or Okular.

The query algorithm maintains two things. First, it maintains the nearest neighbor found so far (or k nearest), and successive improvements decrease this distance. It also maintains a binary min-heap of unexplored subtrees, keyed by distance from $\vec{q}$; successive pops increase the minimum distance in the heap.

Each subtree represents an axis-aligned box. The query tries to avoid searching unnecessary boxes/subtrees by searching boxes close to $\vec{q}$ first.

We measure the distance from $\vec{q}$ to a box, and use it as a key for the subtree in the heap. The search stops when the distance from $\vec{q}$ to the kth nearest neighbor found so far is less than the distance from $\vec{q}$ to the nearest unexplored box (times $1+\varepsilon$ if approximate NN).

Formally, the algorithm for an 1-NN query is as follows:

```
Q = heap containing root node with key zero
r = ∞
while Q is not empty and (1 + ε) * min(Q) < r
    // pop min and update nearest neighbor
    B = Q.removemin()
    w = sample point at B
    r = min(r, dist(q, w))

    // add child boxes to heap
    B1, B2 = child boxes of B
    if (1 + ε) * dist(q, B1) < r then
        Q.insert(B1, dist(q, B1)) // key for B1 is dist(q, B1)
    if (1 + ε) * dist(q, B2) < r then
        Q.insert(B2, dist(q, B2)) // key for B2 is dist(q, B2)
```

For k-NN, we just replace r with a max-heap holding the k nearest neighbors, much like in the exhaustive search algorithm.

See Fig. 25.3 for an animation of an example 1-NN query.

To summarize and explain the animation a little bit, we start with a heap initialized with the root of the k-d tree.

We pop from the min-heap, and compare the distance from $\vec{q}$ to the point in the current node, updating the nearest neighbor if necessary. Then, we look at the shortest distances from $\vec{q}$ to the children *boxes* of the current node. If these distances to the boxes are less than our current nearest neighbor, then there is a possibility that there is a node within the box that is our new nearest neighbor, so we add the child to the heap.

We repeat this process until the minimum element in the heap (i.e. the next element we'd pop off) is too far away—that is, the shortest distance from $\vec{q}$ to the closest unvisited box is greater than the distance from $\vec{q}$ to the current nearest neighbor. At this point, we're guaranteed that there are no other boxes we need to visit, as there are no possible points closer than what we have already.

In the animation, the query point $\vec{q}$ is marked by the bright green dot. When we pop an element from the heap, the arrow from $\vec{q}$ to the point is colored red if it's longer than the current nearest neighbor, or green if we've found a new nearest neighbor. The distance from $\vec{q}$ to the current nearest neighbor is marked by a green circle as well as the arrow.

When we check the children boxes of the current element, the boxes are colored cyan and magenta, with corresponding arrows pointing to the boxes, marking the shortest distance from $\vec{q}$ to the respective boxes. If $\vec{q}$ lies *within* one of the boxes, no arrow is drawn, and instead a circle with the corresponding box color is drawn around $\vec{q}$.

Light blue arrows are also drawn for each (old) element in the heap, marking the distances they represent—the length of the arrows represent the corresponding keys in the heap.

At the very end, we see that the closest box is too far away, so we stop.

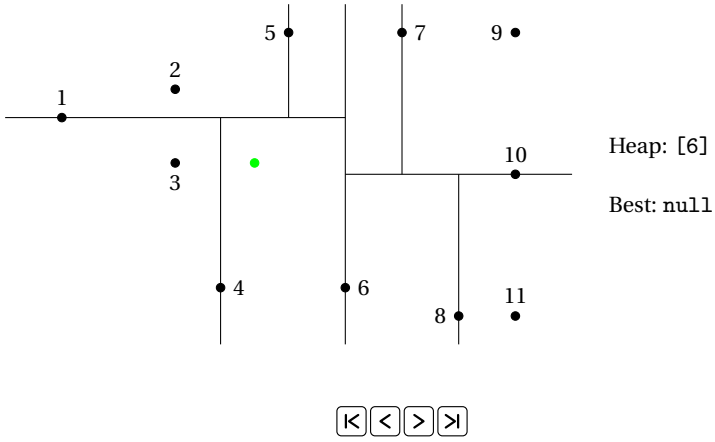


Figure 25.3: Example of the query algorithm for 1-NN in a k-d tree. To step through the animation, you must use a PDF reader that supports animations, eg. Adobe Acrobat or Okular.

This algorithm works with any ℓ_p norm for $p \in [1, \infty)$; k-d trees are not limited to the Euclidean (ℓ_2) norm.

Why would we want to use an ε-approximate NN? In the worst case, we may have to visit every single node in the k-d tree to find the exact nearest neighbor; in this case, the k-d tree is slower than simple exhaustive search, and an *approximate* nearest neighbor search can be much faster.

In practice, settling for an approximate nearest neighbor sometimes improves the speed by a factor of 10 or even 100, because you don't need to look at most of the tree to do a query. This is especially true in high dimensions—remember that in high-dimensional spaces, the nearest point often isn't much closer than *a lot* of other points.

Lecture A

Spectral Graph Clustering

Note: this was not an actual lecture in Spring 2022; it was included in Shewchuk's lecture note PDF, but was not actually covered. I only realized this after I had finished writing up most of the lecture, so I've left the full bonus lecture note here anyways.

Today, we'll be talking about *spectral graph clustering*. We're given as input a weighted undirected graph $G = (V, E)$, with no self-edges. We define w_{ij} as the weight of edge $(i, j) = (j, i)$, with value zero if $(i, j) \notin E$.

Here, we can think of the edge weights as a similarity measure. A large weight means that the two vertices want to be in the same cluster.

This means that the circumstances are the opposite of last lecture on clustering; before, we had a distance/dissimilarity function.

Our goal is to cut G into 2 (or more) pieces, G_i, of similar sizes, but we don't want to cut too much edge weight. There are several ways to make this last bit more precise; a typical goal is to minimize *sparsity*, or the *cut ratio*:

$$\frac{\text{Cut}(G_1, G_2)}{\text{Mass}(G_1) \cdot \text{Mass}(G_2)}.$$

Here, $\text{Cut}(G_1, G_2)$ is the total weight of the cut edges, and $\text{Mass}(G_1)$ is the number of vertices in G_1 (we can also assign masses to vertices). The denominator of this fraction penalizes imbalanced cuts as well.

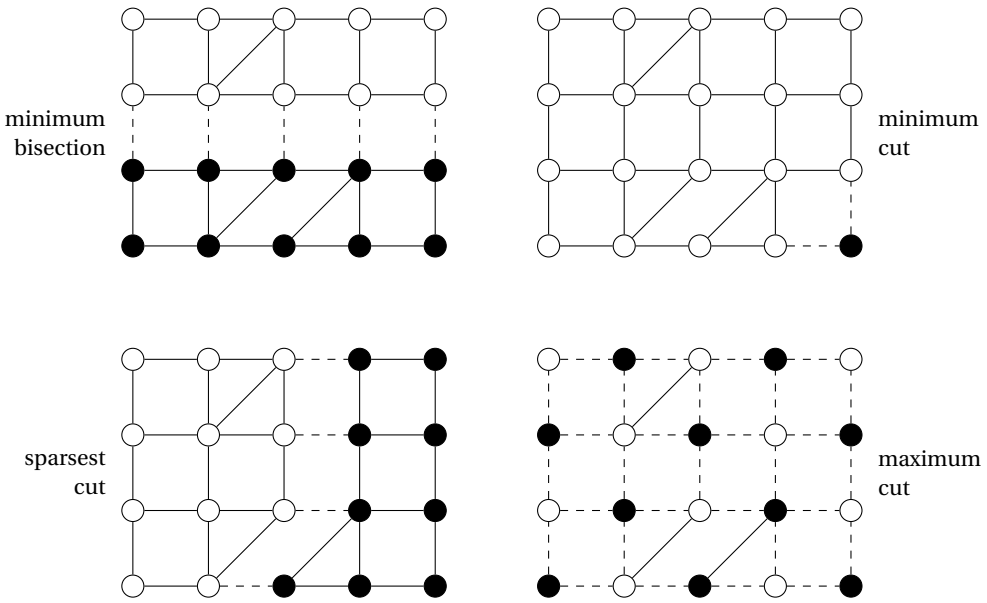


Figure A.1: Four different kinds of cuts of a graph

Figure A.1 shows four different kinds of cuts, with an example graph where all edges have weight 1.

- The *minimum bisection* is the minimum cut such that the two resulting graphs are perfectly balanced.
- The *minimum cut* is the cut with the minimum total weight of cut edges; the minimum cut is usually very unbalanced, which is typically not what we want.
- The *sparsest cut* was described earlier; it's usually good for many applications.
- The *maximum cut* is the cut with the maximum total weight of cut edges; in the example, it's also the maximum bisection.

Sparsest cut, minimum bisection, and maximum cut are all NP-hard. Today, we'll look at an approximate solution to the sparsest cut problem. To do so, we'll first turn this combinatorial graph cutting problem into algebra.

Let $n = |V|$, and let $\vec{y} \in \mathbb{R}^n$ be an *indicator vector*:

$$y_i = \begin{cases} 1 & \text{vertex } i \in G_1 \\ -1 & \text{vertex } i \in G_2 \end{cases}$$

This means that we have

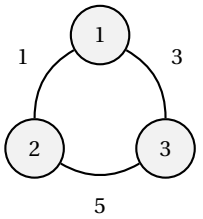
$$w_{ij}\frac{(y_i - y_j)^2}{4} = \begin{cases} w_{ij} & (i,j) \text{ is cut} \\ 0 & (i,j) \text{ is not cut} \end{cases}$$

Calculating the value of the cut, we have

$$\begin{aligned} \text{Cut}(G_1, G_2) &= \sum_{(i,j)\in E} w_{ij}\frac{(y_i - y_j)^2}{4} \\ &= \frac{1}{4}\sum_{(i,j)\in E}\left(w_{ij}y_i^2 - 2w_{ij}y_iy_j + w_{ij}y_j^2\right) \\ &= \frac{1}{4}\Bigg(\underbrace{\sum_{(i,j)\in E} -2w_{ij}y_iy_j}_{\text{off-diagonal terms}} + \underbrace{\sum_{i=1}^{n} y_i^2 \sum_{k\neq i} w_{ik}}_{\text{diagonal terms}}\Bigg) \\ &= \frac{\vec{y}^T\mathbf{L}\vec{y}}{4} \end{aligned}$$

Here, $L_{ij} = \begin{cases} -w_{ij} & i \neq j \\ \sum_{k\neq i} w_{ik} & i = j \end{cases}$.

Note that $\mathbf{L}$ is a symmetric $n \times n$ matrix, called the *Laplacian matrix* for G.



$$\mathbf{L} = \begin{bmatrix} 4 & -1 & -3 \\ -1 & 6 & -5 \\ -3 & -5 & 8 \end{bmatrix}$$

Figure A.2: Example Laplacian matrix of a graph

$\mathbf{L}$ is essentially a matrix representation of a graph G; an example is shown in Fig. A.2.

We can see that minimizing the weight of the cut is equivalent to minimizing the *Laplacian quadratic form* $\vec{y}^T\mathbf{L}\vec{y}$; this lets us turn graph partitioning into a problem in matrix algebra.

Usually we assume that there are no negative weights, in which case $\text{Cut}(G_1, G_2)$ can never be negative, so it follows that $\mathbf{L}$ is positive semidefinite.

We can see that $\mathbf{L}\vec{\mathbf{1}} = \vec{\mathbf{0}}$, since each row of $\mathbf{L}$ sums to zero. This means that $\vec{\mathbf{1}}$ is an eigenvector of $\mathbf{L}$, with eigenvalue 0.

If G is a connected graph and all edge weights are positive, then this is the only zero eigenvalue; however, if G is not connected, then L has one zero eigenvalue for each connected component of G.

A.1 Minimum Bisection

A *bisection* is a cut such that there are exactly $\frac{n}{2}$ vertices in G_1, and exactly $\frac{n}{2}$ vertices in G_2.

Here, we can write $\vec{\mathbf{1}}^T\vec{y} = 0$. This means that we have reduced graph bisection to the following constrained optimization problem:

$$\begin{aligned} \min_{\vec{y}} \quad & \vec{y}^T\mathbf{L}\vec{y} \\ \text{s.t.} \quad & \forall i,\ y_i = \pm 1, \quad \text{(binary constraint)}, \\ & \vec{\mathbf{1}}^T\vec{y} = 0 \qquad \text{(balance constraint)} \end{aligned}$$

However, this optimization problem is NP-hard, because of the binary constraint. We can relax the binary constraint to allow for fractional vertices.

This is a very common approach in combinatorial optimization algorithms, so that the discrete problem becomes a continuous problem. This continuous problem is much easier to solve than the combinatorial problem; after solving the continuous problem, we can round the vertex values to ± 1, and we'll hope that our solution is still close to optimal.

We still can't just drop the binary constraint either; we still need *some* constraint to rule out the solution $\vec{y} = \vec{\mathbf{0}}$.

Our new constraint can be that $\vec{y}$ must lie on a hypersphere of radius $\sqrt{n}$; this is a relaxation of the binary constraint, which forces $\vec{y}$ to lie on the unit hypercube.

This means that the relaxed problem is

$$\begin{aligned} \min_{\vec{y}} \quad & \vec{y}^T\mathbf{L}\vec{y} \\ \text{s.t.} \quad & \vec{y}^T\vec{y} = n, \\ & \vec{\mathbf{1}}^T\vec{y} = 0 \end{aligned}$$

This is equivalent to minimizing the Rayleigh quotient $\frac{\vec{y}^T\mathbf{L}\vec{y}}{\vec{y}^T\vec{y}}$, subject to the same two constraints.

The $\vec{y}$ that minimizes the Rayleigh quotient is the eigenvector with the smallest eigenvalue; we already know that this eigenvector is $\vec{\mathbf{1}}$, but this violates our balance constraint.

This means that $\vec{v}_2$ solves the relaxed problem, with corresponding second-smallest eigenvalue λ_2 of $\mathbf{L}$. This vector is also called the *Fiedler vector* of $\mathbf{L}$.

The simplest way to make $\vec{v}_2$ also satisfy the integer constraint is to round all positive entries to 1 and all negative entries to -1. However, in both theory and in practice, it's better to choose the threshold as follows, with the spectral partitioning algorithm:

- Compute the Fiedler vector $\vec{v}_2$ of $\mathbf{L}$.
- Round $\vec{v}_2$ with a *sweep cut*.

  That is, sort the components of $\vec{v}_2$, and try the $n-1$ cuts between successive components. We choose the minimum sparsity cut. If we're clever about updating the sparsity, we can try all of these cuts in linear time relative to the number of edges in G.

One consequence of relaxing the binary constraint is that the balance constraint no longer forces an exact bisection, but that's okay. We'd prefer a slightly unbalanced cut if it means we cut fewer edges. Even though our discrete problem was the minimum bisection problem, our relaxed continuous problem will be an approximation of the sparsest cut problem (this is a bit counterintuitive).

A.2 Vertex Masses

Sometimes we want the notion of balance to give more prominence to some vertices compared to others. To do this, we can assign masses to vertices.

Let $\mathbf{M}$ be a diagonal matrix with vertex masses on the diagonal. The new balance constraint is then $\vec{\mathbf{1}}^T\mathbf{M}\vec{y} = 0$.

This new balance constraint says that G_1 and G_2 should each have the same total mass. It turns out that this new balance constraint is easier to satisfy if we also revise the sphere constraint a little bit.

Instead of the sphere constraint, we can use a new ellipsoid constraint: $\vec{y}^T \mathbf{M} \vec{y} = \text{Mass}(G) = \sum \mathbf{M}_{ii}$, i.e. we now constrain $\vec{y}$ to lie on an axis-aligned ellipsoid.

The new solution is the Fiedler vector of the *generalized eigensystem* $\mathbf{L}\vec{v} = \lambda \mathbf{M}\vec{v}$.

Most algorithms for computing eigenvectors and eigenvalues of symmetric matrices can easily be adapted to compute eigenvectors and eigenvalues of symmetric *generalized* eigensystems too.

Theorem A.1: Cheeger's Inequality

Sweep cut finds a cut with sparsity at most $\sqrt{2\lambda_2 \max_i \frac{\mathbf{L}_{ii}}{\mathbf{M}_{ii}}}$.

The optimal cut has sparsity $\geq \frac{\lambda_2}{2}$.

This means that the spectral partitioning algorithm is an approximation algorithm, though not one with a constant factor of approximation.

A.3 The Normalized Cut

With a normalized cut, we set vertex i's mass $\mathbf{M}_{ii} = \mathbf{L}_{ii}$ That is, the mass is the sum of edge weights incident to vertex i.

The normalized cut is popular for image segmentation, which is the problem of looking at a photograph and separating it into different objects. To do that, we define a graph on the pixels.

For pixels with coordinate $\vec{p}_i$ and brightness b_i, we use the graph weights

$$w_{ij} = \exp\left(-\frac{\left\|\vec{p}_i - \vec{p}_j\right\|^2}{\alpha} - \frac{\left|b_i - b_j\right|^2}{\beta}\right),$$

or zero if $\left\|\vec{p}_i - \vec{p}_j\right\|$ is large.

We choose a distance threshold, typically less than 4 to 10 pixels apart. Pixels that are far apart from each other aren't connected.

α and β are empirically chosen constants; it often makes sense to choose β proportional to the variance of the brightness values.

A.4 Clustering with Multiple Eigenvectors

When we use the Fiedler vector for spectral graph clustering, it tells us how to divide a graph into two graphs. If we want more than two clusters, we can use divisive clustering: we repeatedly cut the subgraphs into smaller subgraphs by computing their Fiedler vectors.

However, there are several other methods to subdivide a graph into k clusters in one shot that use multiple eigenvectors rather than just the Fiedler vector $\vec{v}_2$; these methods sometimes give better results. They use k eigenvectors in a natural way to cluster a graph into k subgraphs.

For k clusters, we compute the first k eigenvectors $\vec{v}_1 = \vec{\mathbf{1}}, \vec{v}_2, \ldots, \vec{v}_k$ of the generalized eigensystem $\mathbf{L}\vec{v} = \lambda \mathbf{M}\vec{v}$.

We then scale them so that $\vec{v}_i^T \mathbf{M} \vec{v}_i = 1$. For example, $\vec{v}_1 = \frac{1}{\sqrt{\sum_i \mathbf{M}_{ii}}} \vec{\mathbf{1}}$.

After scaling, we have $\mathbf{V}^T \mathbf{M} \mathbf{V} = \mathbf{I}$. That is, the eigenvectors are $\mathbf{M}$-orthogonal.

The columns of $\mathbf{V}$ are the eigenvectors with the k smallest eigenvalues; we do also include the all-1's vector $\vec{v}_1$ as one of the columns of $\mathbf{V}$.

The rows of $\mathbf{V}$ are the *spectral vectors* for vertex i, which we'll label $\mathbf{V}_i$. We'll call the k-dimensional space the row vectors lie in the "spectral space". With more than one eigenvector, it makes sense to cluster vertices together if their spectral vectors point in similar directions.

If we normalize each row $\mathbf{V}_i$ to unit length, the spectral vectors are now points on the unit sphere centered at the origin. We then k-means cluster these vectors; because of the normalizing, this clusters together vectors that are separated by small angles.

A comparison between k-means clustering (i.e. k-means by itself) and spectral clustering is shown in Fig. A.3.

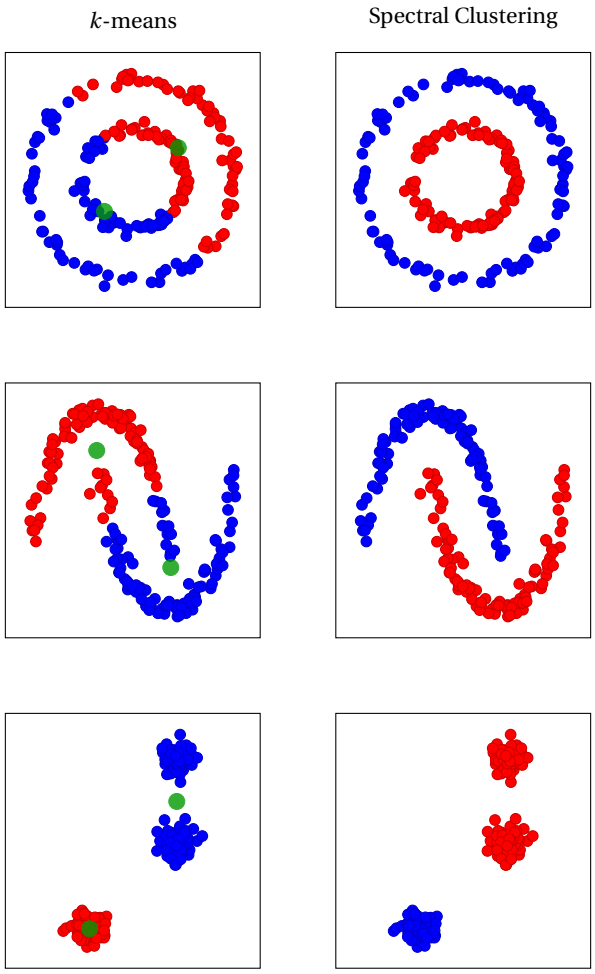


Figure A.3: Comparison between k-means and spectral clustering